Advanced Topics in Complexity Theory

Daniel Borchmann

October 4, 2016

Contents

1 Approximation Complexity **3**
1.1 Function Problems . . . 3
1.2 Approximation Algorithms . . . 6
1.3 Approximation and Complexity . . . 10
1.4 Non-Approximability . . . 14

2 Interactive Proofs **18**
2.1 Interactive Proofs Systems with Deterministic Verifiers . . . 18
2.2 Probabilistic Verifiers . . . 20
2.3 Public Coin Protocols . . . 23
2.4 IP = PSpace . . . 27
2.5 Outlook: Multi-Prover Systems . . . 32

3 Counting Complexity **33**
3.1 Counting problems and the class #P . . . 33
3.2 #P-completeness and Valiant's Theorem . . . 36

1 Approximation Complexity

Dealing with NP-complete problems in practice may sometimes be infeasible. On the other hand, it is often enough to have *approximate* solutions to those problems, provided these are not "too bad". However, for this to make sense, it is no longer possible to only consider decision problems.[1]

1.1 Function Problems

1.1 Example Let us consider the following "computation problem" associated to SAT: given a propositional formula φ, return a satisfying assignment of φ if one exists, and "no" otherwise. Let us call this problem FSAT ("F" for "function"; note however that for a satisfiable formula φ there may be more than one satisfying assignment – the "functional" aspect thus does not really correspond to the mathematical notion of a function).

It is clear that when we can solve FSAT in polynomial time, we can also solve SAT in polynomial time. The converse is also true: if $\mathsf{SAT} \in \mathsf{P}$, then FSAT can be solved in polynomial time as well. This can be done as follows: let $x_1, \ldots, x_n$ be the variables in φ. We first check whether φ is satisfiable, and return "no" if it is not. Otherwise, we set $x_1 = \text{true}$ and ask whether φ is still satisfiable. If it is, we fix $x_1 = \text{true}$, and otherwise $x_1 = \text{false}$. In any case, we continue with $x_2, x_3, \ldots, x_n$. It is easy to see that when we have fixed x_n, we have obtained a satisfying assignment of φ. Moreover, the procedure runs in polynomial time in $|\varphi|$, provided that we can solve SAT in polynomial time. ♢

The previous example illustrates a property of SAT that is called "self-reducibility": we can solve FSAT by considering instances of SAT of equal or smaller size. The same technique also works for other problems.

1.2 Example Let us consider the *traveling salesperson problem (*TSP*)*: given an undirected (complete) graph G with non-negative edge weights, find a Hamiltonian cycle in G, called a *tour*, with minimal total weight. We shall show that we can solve this optimization problem, provided we have access to an oracle for the decision variant $\mathsf{TSP_D}$.

Let $G = (V, E, w)$ be an edge-weighted graph and let n be the length of the encoding $\langle G \rangle$ of G. The cost of a shortest tour on G is then between 0 and 2^n. Using binary search, we can find the cost C of the optimal tour with at most $\mathcal{O}(n)$ calls to $\mathsf{TSP_D}$.

Once C is fixed, we create instances of $\mathsf{TSP_D}$ where the budget is fixed to be C, but where the weights of the underlying graph G have been altered. More precisely, for all edges

[1] The content of this part of the lecture is based on: Christos H. Papadimitriou: *Computational Complexity*, Addison-Wesley, 1995, Section 10.3 and Chapter 13, and: Sanjeev Arora and Boaz Barak: *Computational Complexity A Modern Approach*, Cambridge University Press, 2009, Chapter 11.

$\{ i, j \} \in E$ we successively set its weight to $C + 1$. For each such modification we ask whether G still has a tour of length C or less. If yes, the edge $\{ i, j \}$ is not part of some optimal tour, and we can freeze its cost at $C + 1$. Otherwise, we restore the weight of $\{ i, j \}$ to the previous value. It is easy to see that after $\mathcal{O}(n^2)$ calls to $\mathsf{TSP_D}$, the only edges with weights below $C + 1$ form an optimal tour in G. ♢

We can formalize the relationship between decision and function problems in the following way: let $L \in \mathsf{NP}$. Then there exists a polynomial-time TM $\mathcal{K}$ and some polynomial $p(n)$ such that

$$L = \{ x \in \Sigma^* \mid \exists y. |y| \leq p(|x|) \wedge (x, y) \in \mathcal{K} \}.$$

The *function problem* associated to L, denoted $\mathsf{F}L$, is then the following computation problem:

> Given x, find y such that $|y| \leq p(|x|)$ and $(x, y) \in \mathcal{K}$. If no such y exists, return "no".

We next provide a more formal definition of function problems.

1.3 Definition A *function problem* over Σ is a pair $Q = (I, R)$, where $I \subseteq \Sigma^*$ is called the set of *instances* of Q and $R \subseteq \Sigma^* \times \Sigma^*$. Both I and R must be decidable in polynomial time. For $x \in I$, define $F(x)$ to be the set of *feasible solutions* of x, consisting of all y such that $(x, y) \in R$. A TM M *solves* the function problem Q if it returns for each instances $x \in I$ some $y \in F(x)$ if $F(x) \neq \emptyset$, and rejects x otherwise.

Call a function problem (I, R) *polynomially balanced* if there exists some polynomial p such that $(x, y) \in R$ implies $|y| \leq p(|x|)$. The class FNP consists of all polynomially balanced function problems (I, R). The set FP is the subset of FNP consisting of all function problems solvable in polynomial time. ♢

We have

- $\mathsf{FSAT} \in \mathsf{FNP}$, but is is neither known nor expected that $\mathsf{FSAT} \in \mathsf{FP}$.
- $\mathsf{FHornSAT} \in \mathsf{FP}$.
- Finding perfect matchings in bipartite graphs is in FP.
- $\mathsf{FTSP_D} \in \mathsf{FNP}$.

Note that, on the other hand, it is not known whether TSP is in FNP. The reason is that a given tour cannot be checked easily for being optimal (for all we know). Moreover, TSP is complete for the class $\mathsf{FP}^{\mathsf{NP}}$ of all function problems computable in polynomial time with access to an NP oracle. We shall discuss this in some future exercise.

Note that each problem in FNP can be solved in non-deterministic polynomial time. Conversely, for each function problem $L = (I, R)$ that is solvable in non-deterministic polynomial time, there exists a function problem $L' = (I, R') \in \mathsf{FNP}$ such that

- each solution to some instance of L' is a solution for the same instance of L;

- if some instance of L' does not have a solution, neither has the same instance of L.

Indeed, if M is some polynomial-time NTM solving L, let p be some bounding polynomial of the runtime of M. Define R'

$$R' = \{ (x, y) \mid |y| \leq p(|x|) \wedge (x, y) \in R \}.$$

Because of this, we can identify problems in FNP with function problems solvable in nondeterministic polynomial time.

1.4 Definition Let A, B be two function problems. We say that A *reduces* to B (in logarithmic space) if and only if the following is true: there exist logspace-computable functions f and g such that

1. if x is an instance of A, then $f(x)$ is an instance of B;
2. if $f(x)$ does not have a solution, then x does not have a solution;
3. if z is a solution for $f(x)$, then $g(z)$ is a solution for x.

The problem A is *complete* for a class $\mathsf{F}\mathcal{C}$ of function problems if and only if $A \in \mathsf{F}\mathcal{C}$ and all problems in $\mathsf{F}\mathcal{C}$ reduce to A. ♢

It is not hard to see that FP and FNP are closed under reductions (using the convention for FP and FNP as discussed above), and that reductions compose. It is also not hard to show that FSAT is FNP-complete. From Example 1.1 we know that FSAT can be solved in polynomial time if and only if SAT can be. Thus we have obtained the following result.

1.5 Theorem FP = FNP if and only if P = NP.

Because of this close relationship between FNP/FP and NP/P, one could think that studying function problems does not add any value. However, among the problems in FNP there is a special class of function problems that are guaranteed to never say "no", corresponding to $L = \Sigma^*$.

1.6 Example Consider the problem Factoring: given $N \in \mathbb{N}$, find its *prime decomposition*

$$N = p_1^{k_1} \dots p_m^{k_m}.$$

Since Primes $\in$ P, checking whether a tuple $(p_1, k_1, \dots, p_m, k_m)$ denotes the prime decomposition of N can be done in polynomial time. Alternatively, one could also supply *primality certificates* for $p_1, \dots, p_m$ (such certificates of polynomial length exist; see exercises). ♢

It is not known whether Factoring can be solved in polynomial time. However, in contrast to FSAT, Factoring never returns no on valid instances.

1.7 Definition A function problem $Q = (I, R)$ is called *total* if and only if for each $x \in I$ there exists a $y \in \Sigma^*$ such that $(x, y) \in R$. The class of all total function problems in FNP is denoted TFNP. ♢

1.2 Approximation Algorithms

Function problems in TFNP are always guaranteed to have a solution. However, for practical concerns, some of these solutions may be preferred over others, e.g., if they minimize costs.

1.8 Definition An *optimization problem* is a tuple $(Q, \text{cost}, \text{opt})$, where Q is a total function problem, $\text{cost}(y) \in \mathbb{N} \setminus \{ 0 \}$ for each solution y of Q, and $\text{opt} \in \{ \min, \max \}$. For an instance x of Q the *optimal cost* $\text{OPT}(x)$ of x is defined as

$$\text{OPT}(x) = \text{opt}_{s \in F(x)} \text{cost}(s).$$

A Turing machines *solves* the optimization problem $(Q, \text{cost}, \text{opt})$ if for each instance x of Q the Turing machine accepts and returns a solution $y \in F(x)$ such that $\text{OPT}(x) = \text{cost}(y)$.

An optimization problem $(Q, \text{cost}, \text{opt})$ is called an NP-*optimization problem* if $Q \in$ TFNP and cost is computable in polynomial time. The class of all NP-optimization problems is denoted NPO. ♢

Examples of NP-optimization problems are not hard to find:

- MinVertexCover: given an undirected graph, find a minimal vertex cover;
- TSP (on complete graphs);
- Knapsack: given weights $w_1, \dots, w_n \in \mathbb{N}$, a weight limit $W \in \mathbb{N}$ and $v_1, \dots, v_n \in \mathbb{N}$, find $S \subseteq \{ 1, \dots, n \}$ such that $\Sigma_{i \in S} w_i \leq W$ and $\Sigma_{i \in S} v_i$ is maximal.

It is clear that solving optimization problems is at least as hard as solving the *corresponding decision problem*: if $A = (Q, \text{cost}, \text{opt})$ is an optimization problem, then the corresponding decision problem A_D is

Input:	An instance x of Q and some $k \in \mathbb{N}$
Question:	Does there exist a solution $y \in F(x)$ such that $\text{cost}(y) \leq k$ (if $\text{opt} = \min$) or $\text{cost}(y) \geq k$ (if $\text{opt} = \max$)?

Indeed, if $A \in$ NPO, then $A_D \in$ NP. Furthermore, it is easy to see that if A_D can be solved in polynomial time, then $\text{OPT}(x)$ can be computed in polynomial time for each instance x of Q. In some cases (as in TSP), this allows to compute an optimal solution in polynomial time as well.

An advantage of optimization problems is that we can ask for *approximate* solutions.

1.9 Definition Let $(Q, \text{cost}, \text{opt})$ be an optimization problem. Let M be a Turing machine that for each instance x of Q returns a solution $M(x) \in F(x)$. We say that M is an ε-*approximation algorithm* for some $\varepsilon \in [0, 1]$ if for all $x \in I$ we have

$$\frac{|\text{cost}(M(x)) - \text{OPT}(x)|}{\max(\text{cost}(M(x)), \text{OPT}(x))} \leq \varepsilon.$$ ♢

Note that since $\text{cost}(M(x))$ is positive, the fraction is well-defined for each instance x.

The intuition behind the above definition is that an ε-approximation algorithm always returns a solution whose *relative error* is at most ε. Indeed, for maximization problems ($\text{opt} = \max$), a solution returned by an ε-approximation algorithm will never be smaller than $(1 - \varepsilon) \cdot \text{OPT}(x)$. On the other hand, for minimization problems the returned solution will never be larger than $\frac{1}{1-\varepsilon} \cdot \text{OPT}(x)$. In other words, if C is the cost of the returned solution, then $(1 - \varepsilon) \cdot C \leq \text{OPT}(x)$.

Question: given an NP-optimization problem, what is the smallest ε for which a *polynomial-time* ε-approximation algorithm for this problem exists?

1.10 Definition The *approximation threshold* for an optimization problem A is the infimum over all $\varepsilon \geq 0$ such that there exists a polynomial-time ε-approximation algorithm for A. $\diamondsuit$

Clear: if $\mathsf{P} = \mathsf{NP}$, then each NP-optimization problem has approximation threshold 0. Furthermore, the approximation threshold of an optimization problem can be anywhere between 0 (arbitrary close approximation) and 1 (no approximation). We shall see examples of problems that exhibit a wide range of different approximation thresholds.

1.11 Example A very simple algorithm for finding a vertex cover is the following: let $G = (V, E)$ be an undirected graph.

C = $\emptyset$
while $E \neq \emptyset$ **do**
  choose $\{ i, j \} \in E$
  add i, j to C
  delete i, j **from** G

It is clear that this returns a vertex cover of G. But how good is this cover in terms of approximation?

Note that C contains $\frac{1}{2}|C|$ edges of G, each two of them being disjoint (i.e., C is a *matching* in G). Every vertex cover (including the optimal one) must contain at least one vertex from each edge of a matching. Therefore, $\text{OPT}(G) \geq \frac{1}{2}|C|$ and thus

$$\frac{|C| - \text{OPT}(G)}{|C|} \leq \frac{1}{2}.$$

Hence the approximation ratio of MinVertexCover is at least $\frac{1}{2}$. Surprisingly, no better approximation algorithm is known, and there are results indicating that none exists. $\diamondsuit$

MinVertexCover is a problem that allows for non-trivial approximation algorithms. However, there are problems in NPO for which this is not the case, unless $\mathsf{P} = \mathsf{NP}$.

1.12 Theorem Unless $\mathsf{P} = \mathsf{NP}$, the approximation threshold for TSP is one.

Proof Assume that there exists a polynomial-time ε-approximation algorithm for TSP for some $\varepsilon < 1$. We shall show that in this case HamiltonianCycle can be solved in polynomial time. As this problem is NP-complete, we obtain $\mathsf{P} = \mathsf{NP}$.

Let $G = (V, E)$ be an undirected graph. We construct an instance of TSP as follows: each vertex of G is considered as a city, and the distance between two cities $i, j \in V$ is 1 if $\{ i, j \} \in E$, and $\frac{1}{1-\varepsilon} \cdot |V|$ otherwise.

Then

- if there exists a tour of total cost $|V|$, then this tour will only visit unit edges and thus constitutes a Hamiltonian cycle in G.
- if the shortest tour has length greater than $|V|$, then G does not have a Hamiltonian cycle.

We can decide which of these cases is true by employing our approximation algorithm:

- if a tour of length $|V|$ is returned, then G has a Hamiltonian cycle.
- if a tour is returned that contains an edge with weight $\frac{1}{1-\varepsilon} \cdot |V|$, then the cost of the overall tour is strictly greater than $\frac{1}{1-\varepsilon} \cdot |V|$. Since our approximation algorithm is guaranteed to return a solution y such that the optimum is at least $(1 - \varepsilon) \cdot \text{cost}(y)$, we obtain that the optimal tour has length greater than $|V|$. Thus, G does not have a Hamiltonian cycle. □

The other extreme is that a problem has ε-approximation algorithms for any $\varepsilon > 0$. This is the case for Knapsack.

1.13 Theorem The approximation threshold for Knapsack is 0, i.e., for each $\varepsilon > 0$ there exists a polynomial-time ε-approximation algorithm for Knapsack.

Indeed, approximating an optimization problem containing numbers suggest an immediate approach, namely to trade *accuracy* for *speed*: when ignoring the trailing b bits of some of the numbers involved in the problem instance, we may obtain a problem that is easier to solve. This is just what it happens in the case of Knapsack.

Proof Let an instance of Knapsack be given, that is some positive value $v_1, \ldots, v_n$, some non-negative values $w_1, \ldots, w_n$, and some maximal weight W. We are then seeking a set $S \subseteq \{ 1, \ldots, n \}$ such that $\sum_{i=1}^{n} w_i \leq W$ and $\sum_{i=1}^{n} v_i$ is maximal.

From the introductory course on complexity theory we know that Knapsack can be solved in *pseudopolynomial* time: if all the numbers in the instances of Knapsack are spelled out in unary, then the problem is in P. To see this, one can give an algorithm based on *dynamic programming*. We shall recall this algorithm here.

Let $V = \max\{ v_1, \ldots, v_n \}$ and denote with $W(i, v)$, for $1 \leq i \leq n$ and $1 \leq v \leq nV$, the minimal weight attainable when choosing among the first i items such that the overall value is *exactly* v. We then have the following connection:

$$W(i + 1, v) = \min\{ W(i, v), W(i, v - v_{i+1}) + w_{i+1} \}.$$

Setting $W(0, v) = \infty$ and $W(0, 0) = 0$ for all feasible v we obtain an algorithm that solves Knapsack in time $\mathcal{O}(n^2 V)$: simply choose the maximal v such that $W(n, v) \leq W$.

Now let us put the idea into practice of truncating b bits of the involved numbers. More precisely, let us introduce for our instance $(v_1, \ldots, v_n, W, w_1, \ldots, w_n)$ an *approximate* instance $(v'_1, \ldots, v'_n, W, w_1, \ldots, w_n)$ where v'_i is the same value as v_i with the last b bits set to zero, i.e.,

$$v'_i = 2^b \left\lfloor \frac{v_i}{2^b} \right\rfloor .$$

Then we can solve the approximate instance in time $\mathcal{O}(n^2 V/2^b)$, because we can apply our dynamic programming algorithm to the approximate instance where we just ignore the last b bits, and re-adding them to the final solution.

Suppose we obtain an approximate solution S' in this way, and let S be an optimal solution for the original instance. We then have the following chain of inequalities:

$$\sum_{i \in S} v_i \geq \sum_{i \in S'} v_i \geq \sum_{i \in S'} v'_i \geq \sum_{i \in S} v'_i \geq \sum_{i \in S} (v_i - 2^b) \geq \sum_{i \in S} v_i - n2^b.$$

The first inequality is true because S is optimal for the original instance, and the second one is true because $v_i \geq v'_i$. The next one is due to the optimality of S' for the approximate instance. Finally, $v'_i \geq v_i - 2^b$ and $|S| \leq n$ yield the last two inequalities. Therefore, the difference between the optimal values of S' and S is at most $n2^b$. Using the fact that V is a lower bound on the optimal value of the original instance (assuming that $w_i \leq W$ for all i) we obtain, we obtain an approximation ratio of

$$\varepsilon = \frac{n2^b}{V}.$$

Now given $\varepsilon > 0$ we can set $b = \lfloor \log_2(\varepsilon V/n) \rfloor$ to see that the above algorithm is an ε-approximation algorithm that runs in time $\mathcal{O}(n^2 V/2^b) = \mathcal{O}(n^3/\varepsilon)$, which is a polynomial n. □

Note that each optimization problem with approximation threshold 0 has a sequence of approximation algorithms such that the corresponding error ratios converge to 0. Indeed, in the case of Knapsack, those approximation algorithms are very well behaved: it is just the same algorithm supplied with different values of ε. Those algorithms are so nice, they get an extra name.

1.14 Definition A *polynomial-time ε-approximation scheme* (PTAS) for an optimization problem A is an algorithm that returns for each $\varepsilon > 0$ and each instance x of A a solution of A with relative error at most ε in time bounded by some polynomial $p_\varepsilon(|x|)$ that only depends on ε. Such an approximation scheme is called *fully polynomial* if $p_\varepsilon(|x|) = p(|x|, \frac{1}{\varepsilon})$ for some fixed polynomial p. ♢

It can be shown that, unless $\mathsf{P} = \mathsf{NP}$, not every problem with a polynomial-time approximation scheme also has a fully polynomial-time approximation scheme.

It is also interesting to see that some closely related problems like MinVertexCover and MaxIndependentSet can behave quite differently when it comes to approximation.

1.15 Theorem If there is an ε_0-approximation algorithm for MaxIndependentSet for any $\varepsilon_0 < 1$, then there is a polynomial-time approximation scheme for MaxIndependentSet. In other words, the approximation threshold for MaxIndependentSet is either 0 or 1.

The proof makes use of the following *product construction*: let $G = (V, E)$ be a graph. The *square* of G is the graph G^2 with vertex set $V \times V$ and edges

$$\{ ((u, v), (u', v')) \mid (u = v \wedge (u', v') \in E) \text{ or } (u, v) \in E \}.$$

It is easy to see that G has an independent set of size k if and only if G^2 has an independent set of size k^2. Indeed, if $I \subseteq V$ is independent, then so is

$$\{ (u, v) \mid u, v \in I \}$$

in G^2. Conversely, if $I' \subseteq V \times V$ is independent and $|I'| \geq k$, then both

$$I_1 = \{ u \mid (u, v) \in I' \},$$
$$I_2 = \{ v \mid (u, v) \in I' \}$$

are independent in G, and one of these sets must have size at least $\sqrt{k}$, because $|I_1| \cdot |I_2| \geq |I'|$.

Proof (Theorem 1.15) Assume that an $\mathcal{O}(n^k)$-time bounded ε_0-approximation algorithm for MaxIndependentSet exists. Let G be a graph with a maximal independent set of size r.

If we apply our hypothetical approximation algorithm to G^2, then we obtain an independent set of size at least $(1 - \varepsilon_0) \cdot r^2$. From this we can obtain an independent set of G of size $\sqrt{(1 - \varepsilon_0)} \cdot r$. We thus obtain an $\mathcal{O}(n^{2k})$-time bounded ε_1-approximation algorithm with $\varepsilon_1 = 1 - \sqrt{1 - \varepsilon_0} < \varepsilon_0$.

For a given $\varepsilon > 0$ we can repeat the product construction

$$\ell = \left\lfloor \log_2 \frac{\log_2(1 - \varepsilon_0)}{\log_2(1 - \varepsilon)} \right\rfloor$$

many times to obtain an $\mathcal{O}(n^{2^\ell k}) = \mathcal{O}(n^{k \cdot \log(1-\varepsilon_0)/\log(1-\varepsilon)})$-time bounded ε-approximation algorithm for MaxIndependentSet. This yields the desired approximation scheme. □

1.3 Approximation and Complexity

Given our lack of knowledge about the P = NP question, the best thing we can hope for an NPO-problem is to obtain a polynomial-time approximation scheme for it. Again, answering the questions of whether such a PTAS exists is hard. We therefore employ our usual approach of "complete problem" to show that the existence of a PTAS for certain problems implies the existence of PTAS for others. For this, however, we first need a suitable definition of a *reduction* that preserves approximability.

1.16 Definition Let A, B be optimization problems. An *L-reduction* from A to B is a pair (f, g) of logspace-computable functions such that

- if x is an instance of A, then $f(x)$ is an instance of B;
- if s is a solution for $f(x)$, then $g(s)$ is a solution for x; (i.e., (f, g) is a reduction of function problems)
- there exists $\alpha > 0$ such that if x is an instance of A, then

$$\text{OPT}(f(x)) \leq \alpha \cdot \text{OPT}(x)$$

- there exists $\beta > 0$ such that for all $s \in F(f(x))$ we have

$$|\text{OPT}(x) - \text{cost}(g(s))| \leq \beta \cdot |\text{OPT}(f(x)) - \text{cost}(s)|. \qquad \diamondsuit$$

Note that by the second property an L-reduction maps an optimal solution of $f(x)$ to an optimal solution of x. Moreover, g returns a solution of x which is only linearly further away from the optimum then the approximate solution for $f(x)$.

1.17 Example Let us consider MaxIndependentSet and MinVertexCover again. The trivial reduction between these two problems replaces the bound k by $|V| - k$. However, this reduction is not an L-reduction: the optimal minimal vertex cover may be arbitrarily larger than the optimal independent set (consider $G = K_n$ for this).

On the other hand, if we can restrict our attention to graphs G with maximal node degree k, then this reduction indeed works: a maximal independent set in G has size at least $\frac{|V|}{k+1}$, and a minimal node cover has size at most $|V|$. Thus $\alpha = k + 1$ works here. It is also easy to see that $\beta = 1$ can be chosen, as the size difference between any cover C and a minimal vertex cover, and between $V \setminus C$ and a maximal independent set is the same. $\diamondsuit$

1.18 Proposition If (f, g) is an L-reduction from A to B, and (f', g') is an L-reduction from B to C, then $(f' \circ f, g \circ g')$ is an L-reduction from A to C.

The main point of L-reductions is that they preserve approximability.

1.19 Proposition Let (f, g) be an L-reduction from A to B with constants α and β. Suppose that B admits a polynomial-time ε-approximation algorithm for some $\varepsilon > 0$. Then there exists $\frac{\alpha\beta\varepsilon}{1-\varepsilon}$-approximation algorithm for A.

The important observation is that $\frac{\alpha\beta\varepsilon}{1-\varepsilon}$ tends to zero when ε does. Thus, we can obtain the following corollary.

1.20 Corollary If there is an L-reduction from A to B and B has a PTAS, then A has a PTAS as well.

Proof (Proposition 1.19) The algorithm is simple: given an instance x of A, construct $f(x)$. Then apply the ε-approximation algorithm to $f(x)$ to obtain a solution s. Finally, compute $g(s)$. We claim that this yields an $\frac{\alpha\beta\varepsilon}{1-\varepsilon}$-approximation algorithm for A.

To see this, let us consider the ratio

$$\frac{|\text{OPT}(x) - \text{cost}(g(s))|}{\max(\text{OPT}(x), \text{cost}(g(s)))}.$$

Then we have

- $|\mathrm{OPT}(x) - \mathrm{cost}(g(s))| \leq \beta \cdot |\mathrm{OPT}(f(x)) - \mathrm{cost}(s)|$;
- $\frac{\mathrm{OPT}(f(x))}{\alpha} \leq \mathrm{OPT}(x) \leq \max(\mathrm{OPT}(x), \mathrm{cost}(g(s)))$.

Because s has been obtained by an ε-approximation algorithm, we additionally obtain

$$\mathrm{OPT}(f(x)) \geq \max(\mathrm{OPT}(f(x)), \mathrm{cost}(s)) \cdot (1 - \varepsilon).$$

We can thus conclude

$$\frac{|\mathrm{OPT}(x) - \mathrm{cost}(g(s))|}{\max(\mathrm{OPT}(x), \mathrm{cost}(g(s)))} \leq \frac{\alpha\beta}{1-\varepsilon} \cdot \underbrace{\frac{|\mathrm{OPT}(f(x)) - \mathrm{cost}(s)|}{\max(\mathrm{OPT}(f(x)), \mathrm{cost}(s))}}_{\leq \varepsilon} \leq \frac{\alpha\beta\varepsilon}{1-\varepsilon}. \qquad \square$$

We next want to introduce a complexity class that has natural complete problems under L-reductions. For this we shall first recall a famous result form descriptive complexity theory.

1.21 Definition Let $\mathcal{G}$ be a set of finite graphs (i.e., a decision problem about finite graphs). We say that $\mathcal{G}$ is *expressible in existential second-order logic* (short: $\mathcal{G}$ is $\exists$SO) if there exists an existential second-order sentence $\exists P.\varphi(P)$ such that

$$\mathcal{G} = \{\, G \mid G \text{ a finite graph and } G \models \exists P.\varphi(P) \,\}. \qquad \diamondsuit$$

Examples for $\exists$SO graph problems are VertexCover and IndependentSet. It is easy to see that all $\exists$SO graph problems are in NP. Surprisingly, the converse is also (somewhat) true.

1.22 Theorem (Fagin) NP is the class of all problems that are reducible in polynomial time to $\exists$SO graph problems.

Using this connection, we can now come back to approximation complexity.

1.23 Definition The class SNP (for *strict* NP) is the class of all problems reducible to graph problems expressible in the form of

$$\{\, G \mid \exists S \forall x_1 \dots \forall x_k . \varphi(S, G, x_1, \dots, x_k) \,\},$$

where φ is a quantifier-free FOL formula. Thus, in contrast to full NP, only universal (and no existential) first-order quantifiers are allowed.

We can obtain maximization problems by modifying the defining expression of SNP-problems slightly. Let

$$\psi' = \exists S \forall x_1 \dots \forall x_k . \varphi(S, G_1, \dots, G_m, x_1, \dots, x_k)$$

be as before, but where we now allow more than one input graph $G_1, \dots, G_m$ on the same set of vertices. Define the *corresponding maximization problem* to be

$$\max_S \left| \{\, (x_1, \dots, x_k) \in V^k \mid \varphi(S, G_1, \dots, G_m, x_1, \dots, x_k) \,\} \right|.$$

Define MaxSNP_0 to be the set of all such optimization problems. Define MaxSNP to be the class of all optimization problems that are reducible via L-reductions to problems in MaxSNP_0. $\diamondsuit$

1.24 Example The problem $\mathsf{Max2SAT}$ is in MaxSNP_0. For this to see, we represent a 2CNF-formula φ by three relations G_0, G_1, G_2 over the set of variables in φ with the following intended meaning:

- $G_0(x, y)$ if and only if $x \vee y$ is a clause in φ;
- $G_1(x, y)$ if and only if $\neg x \vee y$ is a clause in φ;
- $G_2(x, y)$ if and only if $\neg x \vee \neg y$ is a clause in φ;

Then we can write $\mathsf{Max2SAT}$ as

$$\max_S |\{\, (x, y) \mid \psi(S, G_0, G_1, G_2, x, y) \,\}|,$$

where

$$\begin{aligned}\psi(S, G_0, G_1, G_2, x, y) = {} & (G_0(x, y) \wedge (S(x) \vee S(y))) \\ & \vee (G_1(x, y) \wedge (\neg S(x) \vee S(y))) \\ & \vee (G_2(x, y) \wedge (\neg S(x) \vee \neg S(y)))\end{aligned}$$

Intuitively speaking, S is the set of true variables).

Similarly, it can be seen that $\mathsf{Max3SAT}$ is in MaxSNP_0. $\diamondsuit$

A crucial property of problems in MaxSNP_0 is that they all have ε-approximation algorithms. To see this, we first need to investigate another optimization problem.

1.25 Example Let $k \in \mathbb{N}$. The problem k-$\mathsf{MaxGSAT}$ (for *maximum generalized satisfiability*) is defined as follows: let $\Phi = \{\, \varphi_1, \ldots, \varphi_m \,\}$ be a set of Boolean expressions in n variables, where each φ_i is a general Boolean expression in at most k variables. We are seeking an assignment of the n variables that maximizes the number of satisfied expressions in Φ.

It can be shown (and we shall do so in the exercises) that the approximation ratio for k-$\mathsf{MaxGSAT}$ is at most $1 - 2^{-k}$. $\diamondsuit$

1.26 Theorem Let $A \in \mathsf{MaxSNP}_0$ and suppose A is of the form

$$\max_S |\{\, (x_1, \ldots, x_n) \mid \varphi(S, G_1, \ldots, G_m, x_1, \ldots, x_n) \,\}|.$$

Then A has a polynomial-time $(1 - 2^{-k_\varphi})$-approximation algorithm, where k_φ is the number of atomic expressions in φ that involve S.

Proof Consider an instance of A over V. For each n-tuple $v = (v_1, \ldots, v_n) \in V^n$ substitute $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$ in φ to obtain a formula φ_v. Then there are three kinds of atomic expressions in φ_v: those involving a G_i, those involving equality, and those involving S. The first two can readily be evaluated to either true or false. We can therefore assume wlog that φ_v is a Boolean combination of atomic expressions $S(v_{i_1}, \ldots, v_{i_r})$.

We can thus view each instance of A as a set of expressions φ_v for all $v \in V^n$ and we are asked to assign truth values to $S(v_{i_1}, \ldots, v_{i_r})$ to make as many of these φ_v's true. Obviously, this is an instance of k_φ-$\mathsf{MaxGSAT}$, and we know that this problem has a polynomial-time $(1 - 2^{-k_\varphi})$-approximation algorithm. $\square$

Thus all problems in MaxSNP_0 share the positive property of having an approximation ratio that is strictly less than one. However, it is not clear whether all problems in MaxSNP_0 also have PTASs. This is indeed a very difficult question, and calls for considering *complete problems* for MaxSNP.

1.27 Definition An optimization problem A is called MaxSNP-*complete* if $A \in \mathsf{MaxSNP}$ and for each problem $B \in \mathsf{MaxSNP}$ there exists an L-reduction from B to A. ♢

Clearly, it is sufficient to only consider MaxSNP-complete problems that are actually in MaxSNP_0.

1.28 Theorem Max3SAT is MaxSNP-complete.

Proof It suffices to show that all problems in MaxSNP_0 are reducible to Max3SAT. The previous proof shows that each such problem A defined by

$$\max_S |\{ (x_1, \ldots, x_k) \mid \varphi \}|$$

can be reduced to ℓ-MaxGSAT for some $\ell \in \mathbb{N}$. We shall show how we can extend this reduction to obtain an instance of Max3SAT by providing an L-reduction (f, g).

Recall that the construction of the previous proof produced for each instance x of A a set of Boolean expressions φ_v, where the variables in φ_v represent the fact whether certain tuples belong to S or not. We can discard each φ_v that is not satisfiable (note that this can be checked in polynomial time, since k is fixed!) All the remaining φ_v's we can represent as Boolean circuits. These circuits can be transformed into a 3SAT formula as we did when reducing CircuitSAT to 3SAT: we replace each gate by a set of clauses stating the relationship between the inputs and the output. Those clauses have at most three variables. In addition, we add a clause (t) for the output gate. The resulting set of clauses is then the instance $f(x)$ of Max3SAT. Clearly, for any truth assignment T of this instance, we can immediately read off a feasible solution $S = g(T)$ of x.

We are left to show that this reduction is indeed an L-reduction. Each φ_v is replaced by at most c_1 clauses, where c_1 depends on the size of φ (essentially, c_1 is four times the number of Boolean connectives in φ). Let $m \in \mathbb{N}$ be the number of satisfiable φ_v's. The optimal value of x is at least some constant fraction of m, say $\mathrm{OPT}(x) \geq c_2 \cdot m$ for some $c_2 > 0$ (by the previous theorem, we could choose $c_2 = 2^{-k_f}$). Clearly, $\mathrm{OPT}(f(x)) \leq c_1 \cdot m$, and thus

$$\mathrm{OPT}(f(x)) \leq \frac{c_1}{c_2} \cdot \mathrm{OPT}(x).$$

It is also easy to see that the second condition of L-reductions is satisfied with $\beta = 1$. □

1.4 Non-Approximability

The question whether MaxSNP-complete problems have polynomial-time approximation schemes is very akin to the P = NP question: the former asks whether difficult optimization

problems can be approximated efficiently to arbitrary precision, and the latter asks whether search problems can be solved efficiently by a direct method. Indeed, using a deep result from complexity theory, we shall show that these two questions are actually equivalent!

The result we shall be using is concerned with an alternative characterization of NP. We have already seen that we can understand NP in terms of short certificates. It turns out that this characterization can be relaxed to "probabilistic checkable certificates". Here a probabilistic verifier has random access to a certificate π and issues queries into this certificate. We shall demand that those queries are *non-adaptive*, i.e., the outcome of one query does not influence the choice of the next query (this restriction is mostly technical, and it turns out the result we are after does not depend on this).

1.29 Definition Let $L \subseteq \Sigma^*$ and $q, r \colon \mathbb{N} \to \mathbb{N}$. We say that L has an $(r(n), q(n))$*-PCP verifier* if the following is true: there exists a polynomial-time probabilistic TM V such that on inputs $x \in \{0,1\}^n$ and given random access to a string $\pi \in \{0,1\}^*$ of length at most $q(n)2^{r(n)}$, called the *proof*, the machine uses at most $r(n)$ random bits and makes at most $q(n)$ non-adaptive queries to locations in π. The output of V is either 1 for *accept* or 0 for *reject*.

Denote with $V^\pi(x)$ the random variable representing the output of V on input x with access to π. Then V must satisfy the following properties:

- V must be *complete*: if $x \in L$, then there exists a proof $\pi \in \{0,1\}^*$ such that $\Pr(V^\pi(x) = 1) = 1$. (In this case, π is called a *correct proof* for x).
- V must be *sound*: if $x \notin L$, then *for every* $\pi \in \{0,1\}^*$ we have $\Pr(V^\pi(x) = 1) \le \frac{1}{2}$.

We say that L is in $\mathsf{PCP}(r(n), q(n))$ if there are $c, d > 0$ such that L has a $(c \cdot r(n), d \cdot q(n))$-PCP verifier. ♢

Note that the restriction on the length of the proof to $q \cdot 2^r$ bits is inconsequential, as the verifier can look on at most this many locations during its run. Furthermore, the particular choice of the parameter $\frac{1}{2}$ in the definition of PCP-verifiers is not relevant: it can be replace by any number in $[0, 1)$ without changing the definition.

1.30 Example It is true that $\mathsf{GNI} \in \mathsf{PCP}(\mathsf{poly}(n), 1)$, where $\mathsf{PCP}(\mathsf{poly}(n), 1) = \bigcup_{c \ge 1} \mathsf{PCP}(n^c, 1)$. A PCP-verifier receives as input a pair (G_0, G_1) of two graphs on n nodes. The proof π is expected to contain for each labeled graph $\mathcal{H}$ with n nodes a bit $\pi(\mathcal{H})$ corresponding to whether $\mathcal{H} \simeq \mathcal{G}_0$ or $\mathcal{H} \simeq \mathcal{G}_1$ ($\pi(\mathcal{H})$ can be arbitrary if neither of theses cases is true). Note that π is exponentially long.

Now the verifier picks at random some $b \in \{0,1\}$ and a permutation σ on the n nodes. Then σ is applied to G_b, resulting in a graph $\mathcal{H}$. The verifier accepts if and only if $\pi(\mathcal{H}) = b$.

If $G_0 \not\simeq G_1$, then the verifier clearly always accepts with the correct proof. If $\mathcal{G}_0 \simeq \mathcal{G}_1$, then the probability that π makes the verifier accept is at most $\frac{1}{2}$. ♢

The famous and deep result about NP is now the following.

1.31 Theorem (The PCP-Theorem) $\mathsf{NP} = \mathsf{PCP}(\log n, 1)$.

It is easy to see that

$$\mathsf{PCP}(r(n), q(n)) \subseteq \mathsf{NTime}(2^{\mathcal{O}(r(n))} \cdot q(n)).$$

This is because an NTM can guess the proof in $\mathcal{O}(2^{\mathcal{O}(r(n))} \cdot q(n))$ time and verify deterministically for each of the $2^{\mathcal{O}(r(n))}$ choices of the random bits whether the verifier accepts. If the verifier accepts in all cases, the machine accepts. In particular,

$$\mathsf{PCP}(\log n, 1) \subseteq \mathsf{NTime}(2^{\mathcal{O}(\log n)}) = \mathsf{NP},$$

which establishes the easy direction of the PCP-Theorem.

There is also a "scaled-up" version of the PCP-theorem.

1.32 Theorem $\mathsf{PCP}(\mathsf{poly}(n), 1) = \mathsf{NExpTime}$.

Surprisingly, the PCP-Theorem can be used to show that Max3SAT very likely does not have a PTAS.

1.33 Theorem If there is a PTAS for Max3SAT, then $\mathsf{P} = \mathsf{NP}$.

Proof Let $L \in \mathsf{NP}$ and V be a PCP-verifier using $c \cdot \log n$ random bits and d proof queries. Suppose further that there is a PTAS for Max3SAT achieving an approximation ratio $\varepsilon > 0$ in time $p_\varepsilon(n)$, a polynomial. We shall describe a polynomial-time algorithm for deciding L.

Consider some $x \in \Sigma^*$, $|x| = n$. Let $r \in \{0, 1\}^{c \cdot \log n}$ and consider the computation of V on x using the random bits in r. Our goal is now to construct a Boolean expression representing the fact that this computation accepts.

Observe that during the computation, V seeks d bits of the proof, say $\pi_{i_1(r)}, \ldots, \pi_{i_d}(r)$. Except for these bits, all other aspects of the computation of V with random bits r are completely determined. Thus, the outcome is a Boolean function of d bits, and thus can be represented as a circuit C_r. The number of gates in C_r is a most $K = d \cdot 2^d$. Using again the idea of the reduction from CircuitSAT to SAT, we can express C_r with $4K$ or fewer clauses. Note that no matter how we set the input bits $\pi_{i_1(r)}, \ldots, \pi_{i_d(r)}$, all but one of the clauses can be satisfied (i.e., the clause corresponding to the output gate may be unsatisfied). Only certain settings of these input variables can satisfy also the last clause. Those settings correspond to an accepting computation of V.

When repeating this construction for all $2^{c \cdot \log n} = n^c$ choices for r, we obtain $4Kn^c$ clauses. The various groups of clauses may only share the inputs $\pi_{i_j(r)}$. We then have the following situation:

- if $x \in L$, then there is a truth assignment (i.e., a proof π) that satisfies all clauses;
- if $x \notin L$, any truth assignment must miss one clause in at least half of the groups: at least a fraction of $\frac{1}{2} \cdot \frac{1}{4K}$ of the overall clauses must be left unsatisfied.

We now apply our PTAS for Max3SAT to the constructed set of clauses with $\varepsilon = \frac{1}{9K}$. The running time will then be $p_{\frac{1}{12K}}(4Kn^c)$, a polynomial in n. If the PTAS returns an assignment with more than $1 - \frac{1}{8K}$ satisfied clauses, then we know that $x \in L$. Otherwise, since the returned assignment is guaranteed to be within $\frac{1}{9K}$ of the optimum, we know that no assignment can satisfy all clauses. Thus, $x \notin L$.

This shows that L can be decided in polynomial time, i.e., $L \in \mathsf{P}$. □

As Max3SAT is MaxSNP-complete, we obtain that no MaxSNP-complete problem can have a PTAS unless P = NP.

1.34 Theorem Unless P = NP, none of the following problems has a PTAS: Max3SAT, MaxNAESAT, 4-DegreeMaxIndependentSet, MinVertexCover, and MaxCut.

Proof (not really) It can be shown that all the mentioned problems are complete for MaxSNP, by reducing Max3SAT to them (see Papadimitriou's book). □

Combining this with our previous result about MaxIndependentSet, we obtain even worse news.

1.35 Corollary Unless P = NP, the approximation threshold of MaxIndependentSet and MaxClique is 1.

2 Interactive Proofs

Convincing others of the truth of a mathematical statement can usually be done in only one way: a proof has to be provided that everyone can check in "comparably" short time. However, in particular in discussion between mathematicians, the more general method of *interaction* is used: a *prover* wants to convince a *verifier* about the validity of a statement. He does so by exchanging a certain number of explanations with the verifier until she is convinced.[1]

This form of "proving" is also common in cryptographic protocols used for *authentication*: by exchanging a certain number of messages, the prover wants to convince the verifier that she posses a certain identity, usually represented by a secret. Proper cryptographic protocols ensure that this can be done in short time if and only if the prover is indeed in possession of the secret. In addition, those protocols shall also ensure that the verifier does not learn anything about the secret of the prover.

Using interactive proofs also seems to be interesting from a complexity theory point of view: the class NP can be understood as the class of all problems where a proof of polynomial length exists that can be checked in polynomial time; this roughly corresponds to the notion of "mathematical provability". An obvious followup question is: are those all the problems that are also solvable by interactive proofs, or are there more? For example, is it possible to succinctly proof *un*satisfiability of a Boolean formula? On the one hand, this is a problem that is complete for coNP, and a short proof in the sense of NP is not expected. On the other hand, and quite surprisingly, it can be shown that there *do* exist short interactive proofs for unsatisfiability of Boolean formulas, and more generally for every problem that is solvable in polynomial space. In this sense, polynomial space is equivalent to a polynomial number of interactions each realizable in polynomial time. Understanding this remarkable result is one of the goals of this part of the lecture.

2.1 Interactive Proofs Systems with Deterministic Verifiers

It turns out that a crucial part for the computational power of interactive protocols is the possibility that the verifier can make random choices and is allowed to make errors. To illustrate this, we shall show that if the verifier is *deterministic*, interactive protocols do not add any value.

[1]This part is based on Sanjeev Arora and Boaz Barak: *Computational Complexity A Modern Approach*, Cambridge University Press, 2009, Chapter 8 and Michael Sipser: *Introduction to the Theory of Computation*, 3rd ed., Cengage Learning, 2013, Section 10.4

2.1 Example We can solve 3SAT using a deterministic proof: for each clause, the prover announces values for all variables in the clause. The verifier keeps track of all values and accepts if and only if all clauses were indeed satisfied and no conflicting values for two variables were announced. ♢

Indeed, "interaction" in the previous examples is barely necessary: the prover could just send the whole assignment to the verifier in one step. As it turns out, this is not a coincidence: if the verifier is *deterministic*, interaction does not add any computational power.

2.2 Definition Let $f, g\colon \Sigma^* \to \Sigma^*$ be two functions and let $k\colon \mathbb{N} \to \mathbb{N}$. A *$k$-round interaction* of f and g on input $x \in \{0,1\}^*$ is a sequence $a_1, \dots, a_m \in \{0,1\}^*$, where $m = k(|x|)$, such that

$$\begin{aligned}
a_1 &= f(x) \\
a_2 &= g(x, a_1) \\
a_3 &= f(x, a_1, a_2) \\
&\vdots \\
a_{2i+1} &= f(x, a_1, \dots, a_{2i}) \quad (2i < m) \\
a_{2i+2} &= g(x, a_1, \dots, a_{2i+1}) \quad (2i+1 < m).
\end{aligned}$$

Let us write $\langle f, g\rangle(x) = a_1, \dots, a_m$. The *output* of f at the end of the interaction is defined as the last value $a_m = f(x, a_1, \dots, a_{m-1})$. We assume that this value is always in $\{0,1\}$ and shall we denote it by $\mathrm{out}_f\langle f, g\rangle(x)$. ♢

Based on interaction of deterministic functions, we can define what we mean by a *deterministic proof system.*

2.3 Definition Let $L \subseteq \Sigma^*$. L is said to have a *k-round deterministic interactive proof system* if there exists a deterministic Turing machine V that on input $x, a_1, \dots, a_i$ runs in time polynomial in $|x|$ and has k-round interactions with any function $P\colon \{0,1\}^* \to \{0,1\}^*$ such that

$$\begin{aligned}
x \in L &\implies \exists P.\, \mathrm{out}_V\langle V, P\rangle(x) = 1 && (\textit{completeness}) \\
x \notin L &\implies \forall P.\, \mathrm{out}_V\langle V, P\rangle(x) = 0 && (\textit{soundness})
\end{aligned}$$

Denote with dIP the class of all languages with a k-round deterministic interactive proof system, where $k(n)$ is a polynomial in n. ♢

Note that we place no limits on the power of the prover P. This is intentional: a false assertion should not be provable no matter how powerful the prover is.

2.4 Proposition $\mathsf{dIP} = \mathsf{NP}$.

Proof It is clear that $\mathsf{NP} \subseteq \mathsf{dIP}$. Let $L \in \mathsf{dIP}$ and let V be a verifier for L. We want to show $L \in \mathsf{NP}$. For this we provide a certificate for membership. Indeed, such a certificate is just an interaction $(a_1, \dots, a_m)$ of V that has caused V to accept. To check this certificate, a verifier V' only needs to check

$$\begin{aligned} V(x) &= a_1 \\ V(x, a_1, a_2) &= a_3 \\ &\vdots \\ V(x, a_1, \dots, a_m) &= 1. \end{aligned}$$

Indeed, if such a tuple $(a_1, \dots, a_m)$ exists that causes V to accept, than we can define a prover P by

$$\begin{aligned} P(x, a_1) &= a_2 \\ P(x, a_1, a_2, a_3) &= a_4 \\ &\vdots \end{aligned}$$

Since $\mathrm{out}_V \langle V, P \rangle(x) = 1$, we have $x \in L$. Conversely, if $x \in L$, then there exists a prover P and an interaction $(a_1, \dots, a_m) = \langle V, P \rangle(x)$ that makes V' accept.

Thus

$$L = \{\, x \mid \exists a_1, \dots, a_m . |a_1, \dots, a_m| \leq q(|x|) \wedge (x, a_1, \dots, a_m) \in \mathcal{L}(V') \,\} \in \mathsf{NP},$$

for some polynomial q. Indeed, since $a_1, a_3, \dots$ are outputs of V, they have to be polynomially bounded in $|x|$. Furthermore, the values $a_2, a_4, \dots$ can be chosen to be short, because V does not have enough time to read them completely if they are longer. Thus, if p is a bounding polynomial for the runtime of V, then we can choose $q(|x|) = p(|x|) \cdot k(|x|)$. □

2.2 Probabilistic Verifiers

We have seen that deterministic verifiers do not add any expressive power to interactive proofs. Indeed, to take advantage of of interaction it turns out that we need to use *randomization*. In particular, we shall allow the verifier to make "small" amounts of errors.

2.5 Example Marla wants to convince Arthur that she has two socks of different colors, red and yellow, say. Regrettably, Arthur is color-blind. Marla can nevertheless achieve her goal by means of the following protocol: Marla gives both socks to Arthur, telling him which one is red and which one is yellow. She then turns her back to Arthur. He randomly decides to swap the socks in his hand (e.g., by tossing a coin). He then asks Marla to tell him in which hand the red sock is.

Clearly, if the socks have different colors, Marla can tell Arthur with certainty. If, on the other hand, the socks would have identical colors, Marla could only guess with probability $1/2$ the correct answer (assuming there are no other means to distinguish the socks). Thus, by repeating the procedure, Arthur can convince himself with arbitrarily high probability that the socks are indeed of different color. ♢

The idea behind this protocol is actually quite general: we have already seen (and will see again) how it can be used to decide graph-nonisomorphism (GNI). It can also be used to decide quadratic non-residuosity of number in $\mathbb{Z}_n$ (see exercises).

Let us now extend the definition of interaction between two functions f, g to make f a *probabilistic* function. For this we shall add another argument to f that represents some random choice in $\{0,1\}^m$ for some $m \in \mathbb{N}$. On the other hand, the function g is left as it is, as we do not want g to see the random choice made by f. Then, $\langle f, g\rangle(x)$ and $\mathrm{out}_f\langle f, g\rangle(x)$ are not just plain values anymore, but random variables depending on the choice r.

2.6 Definition Let $k\colon \mathbb{N} \to \mathbb{N}$. The class $\mathsf{IP}[k]$ of all *interactive proofs* of length k consists of all languages $L \subseteq \Sigma^*$ satisfying the following condition: there exists a polynomial-time probabilistic Turing machine V that can have a k-round interaction with functions $P\colon \{0,1\}^* \to \{0,1\}^*$ such that

- V is *complete*: if $x \in L$, then there exists some mapping P such that
$$\Pr(\mathrm{out}_V\langle V, P\rangle(x) = 1) \geq \frac{2}{3}.$$
- V is *sound*: if $x \notin L$, then for all mappings P we have
$$\Pr(\mathrm{out}_V\langle V, P\rangle(x) = 0) \leq \frac{1}{3}.$$

Finally define
$$\mathsf{IP} = \bigcup_{c\geq 1} \mathsf{IP}[n^c]. \qquad \diamondsuit$$

As in the definition of BPP, the particular choice of $1/3$ in the above definition is not crucial.

2.7 Lemma The class IP is unchanged if we replace the completeness parameter $2/3$ by $1 - 2^{-n^s}$ and the soundness parameter $1/3$ by 2^{-n^s} for any fixed $s \in \mathbb{N}$.

Proof Use repetition and take the majority of the outcomes. Use *Chernoff's Bounds* to show that this yields the desired parameters. (See exercises for more details) $\square$

Indeed, replacing the completeness parameter $2/3$ by 1 also not change the definition of IP. This is rather difficult to prove from the definition alone, but can be done using the characterization $\mathsf{IP} = \mathsf{PSpace}$. On the other hand, replacing $1/3$ by 0 yields the class NP again, as shown in the previous section (and it is believed that $\mathsf{NP} \neq \mathsf{IP}$).

In the definition of IP, the prover g is allowed to have unbounded computational resources. It turns out that this is not necessary: restricting the prover to only polynomial space does not change the definition of IP.

2.8 Theorem $\mathsf{IP} \subseteq \mathsf{PSpace}$.

Proof Let $L \in \mathsf{IP}$ and let V be a verifier for L. Then

$$x \in L \iff \max_P \Pr(\mathrm{out}_V\langle V, P\rangle(w) = 1) \geq \frac{2}{3},$$
$$x \notin L \iff \max_P \Pr(\mathrm{out}_V\langle V, P\rangle(w) = 1) \leq \frac{1}{3}.$$

We shall now show how we can compute this maximum in polynomial space.

Let $w \in \Sigma^*$ and $n := |w|$. Assume that on inputs of length n exactly $p = p(n)$ messages are exchanged. Denote with $M_j = (m_1, \dots, m_j)$ a tuple of messages of length $j \leq p$. Let us denote with $\mathrm{out}_V\langle V, P\rangle(w, r, M_j)$ the value (if it exists) of the interaction between V and P on input w, using random string r, and starting on message history M_j. In particular, we have

$$\mathrm{out}_V\langle V, P\rangle(w, r, M_j) = 1 \tag{2.1}$$

if and only if we can extend M_j with messages $m_{j+1}, \dots, m_p$ such that

1. $V(w, r, m_1, \dots, m_i) = m_{i+1}$ if i is even;
2. $P(w, r, m_1, \dots, m_i) = m_{i+1}$ if i is odd;
3. $m_p = 1$.

Note that then the message history M_j must be consistent with the output of V. When (2.1) is true, we shall say that V *accepts* w *starting at* M_j *with random choice* r.

Now write

$$\Pr(\mathrm{out}_V\langle V, P\rangle(w, M_j) = 1) = \Pr_r(\mathrm{out}_V\langle V, P\rangle(w, r, M_j) = 1),$$

where we choose r uniformly from $\{ 0, 1 \}^{m'}$, where m' is the length of the random string for inputs of length $|w|$, and where r is consistent with the current message history M_j. If no such r exists, we set the overall value to 0.

For $0 \leq j \leq p$ and M_j a message history, we now introduce values N_{M_j} as follows. For $j = p$ we set $N_{M_p} = 1$ if V accepts w starting at M_p for some choice of r. This amounts to checking that M_p is consistent for V and some random choice r and the interaction accepts. Otherwise, set $N_{M_p} = 0$.

For $j < p$ and some message history M_j define N_{M_j} recursively by

$$N_{M_j} := \begin{cases} \max_{m_{j+1}} N_{(M_j, m_{j+1})} & \text{for } j \text{ odd} \\ \text{wt-avg}_{m_{j+1}} N_{(M_j, m_{j+1})} & \text{for } j \text{ even,} \end{cases}$$

where (M_j, m_{j+1}) shall denote the message history M_j extended by the new message m_{j+1}, and

$$\text{wt-avg}_{m_{j+1}} N_{(M_j, m_{j+1})} = \sum_{m_{j+1}} \Pr_r(V(w, r, M_j) = m_{j+1}) \cdot N_{(M_j, m_{j+1})}.$$

We shall now show that

$$N_{M_j} = \max_P \Pr(\mathrm{out}_V\langle V, P\rangle(w, M_j) = 1).$$

We do this by induction over j, starting with $j = p$.

Base Case ($j = p$). We know that m_p is either 0 or 1. If $m_p = 1$ and M_p is consistent for V for some random choice r, then $N_{M_p} = 1 = \max_P \Pr(\text{out}_V\langle V, P\rangle(w, M_p) = 1)$. Otherwise, $N_{M_p} = 0 = \max_P \Pr(\text{out}_V\langle V, P\rangle(w, M_p) = 1)$.

Step Case: We assume the claim is true for $j + 1 \leq p$ and any message history M_{j+1} of length $j + 1$. The goal is to show that the claims is true for j and any message history M_j of length j. We distinguish two cases.

If j is even, then m_{j+1} is a message from V to P. Then

$$\begin{aligned}
N_{M_j} &= \text{wt-avg}_{m_{j+1}} N_{(M_j, m_{j+1})} \\
&= \sum_{m_{j+1}} \Pr_r(V(w, r, M_j) = m_{j+1}) \cdot N_{(M_j, m_{j+1})} \\
&= \sum_{m_{j+1}} \Pr_r(V(w, r, M_j) = m_{j+1}) \cdot \max_P \Pr(\text{out}_V\langle V, P\rangle(w, M_j) = 1) \\
&= \max_P \Pr(\text{out}_V\langle V, P\rangle(w, M_j) = 1),
\end{aligned}$$

where the first two equalities are by definition, and the third follows from the induction hypothesis. The last equation is easy to see and will be discussed in the exercises.

If j is odd, then m_{j+1} is a message from P to V. In this case we have

$$\begin{aligned}
N_{M_j} &= \max_{m_{j+1}} N_{(M_j, m_{j+1})} \\
&= \max_{m_{j+1}} \max_P \Pr(\text{out}_V\langle V, P\rangle(w, (M_j, m_{j+1})) = 1) \\
&= \max_P \Pr(\text{out}_V\langle V, P\rangle(w, M_j) = 1),
\end{aligned}$$

where the last equation will again be discussed in the exercises.

Thus, to know whether $\max_P \Pr(\text{out}_V\langle V, P\rangle(w) = 1) \geq 2/3$ it is enough to compute N_{M_0}, where M_0 is the empty message history. But this is easy, since the only problem is to compute the wt-avg expression. To compute this, we go through all choices r of length m' and eliminate those that are inconsistent with M_j. If no r remains, the wt-avg-expression is 0. Otherwise, determine the fraction of strings r that cause V to output m_{j+1} by simulating V in polynomial time (and thus polynomial space). Then compute $N_{(M_j, m_{j+1})}$. Since the depth of the recursion is p (a polynomial), only polynomial space is needed. □

2.3 Public Coin Protocols

It seems as if in the protocols we have seen so far, keeping the random choices of the verifier secret was crucial for the protocols to work. A natural question emerging from this conjecture is to ask which problems can be solved if the random choices are made public to the prover. This is formalized using so-called *public-coin* protocols.

2.9 Definition Let $k \colon \mathbb{N} \to \mathbb{N}$. The class $\mathsf{AM}[k]$ is the class of all languages that have a polynomial-time probabilistic verifier V that is only allowed to send messages of random

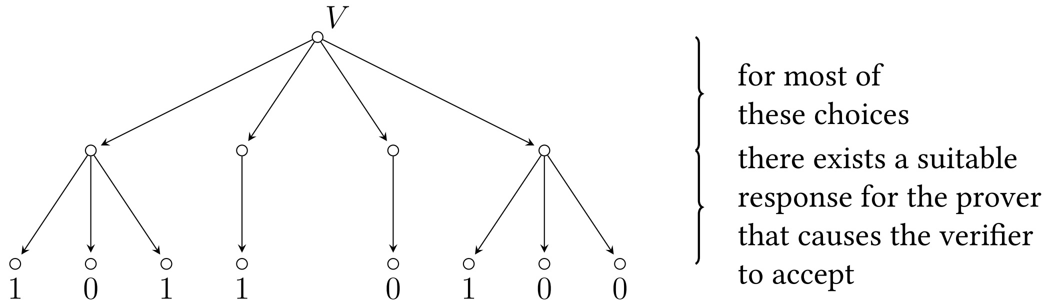


Figure 2.1: Decision tree of an $\mathsf{AM}[2]$-interaction

bits and these are all the random bits V can use. An interactive proof with such a verifier is called a *public-coin* proof or an *Arthur-Merlin* proof. In contrast, interactive proofs that are not necessarily public-coin protocols are called *private-coin* protocols. $\diamondsuit$

Note that in this type of protocol, the prover does not get to see all of the random choices of the verifier immediately. Indeed, for a k-round interaction, the verifier chooses $\ell = \lceil k/2 \rceil$ random strings $r_1, \dots, r_\ell$ and the ith message is then simply r_i. Then after the last message, the verifier applies a deterministic procedure to the transcript of the interaction to decide about acceptance.

Clearly, public-coin protocols are a special case of interactive proofs, i.e. $\mathsf{AM}[k] \subseteq \mathsf{IP}[k]$. It can also be shown that $\mathsf{AM}[k] = \mathsf{AM}[2]$ for all constant k. Because of this, we define $\mathsf{AM} = \mathsf{AM}[2]$.

Similarly, one can define the class MA of public coin interactive protocols where the prover sends the first message. In this type of interaction, the verifier receives this message, makes some random choices, and then applies a deterministic procedure to decide acceptance.

Note that the class AM is quite similar to Π_2^p, where the first $\forall$-quantifiers are replaced by probabilistic choices. See Figure 2.1.

Question: Are public-coin protocols strictly weaker than private-coin protocols?

Expectation: The protocol of GNI crucially depends on private randomness, so "maybe yes".

Answer: No.

2.10 Theorem (Goldwasser, Sipser, 1987) For $k\colon \mathbb{N} \to \mathbb{N}$ computable in polynomial time it is true that

$$\mathsf{IP}[k] \subseteq \mathsf{AM}[k+2].$$

The proof is rather involved, and we shall not discuss it here in detail. Instead, we shall illustrate its main idea by sketching $\mathsf{GNI} \in \mathsf{AM}[2]$.

2.11 Theorem $\mathsf{GNI} \in \mathsf{AM}[2]$.

The main strategy of the proof is as follows: let G_1, G_2 be two graphs on n vertices. Assume first that both G_1 and G_2 have *exactly* $n!$ equivalence classes: every permutation $\pi \in S_n$ yields different graphs $\pi(G_1)$ and $\pi(G_2)$. Define

$$S = \{\, \mathcal{H} \mid \mathcal{H} \simeq G_1 \lor \mathcal{H} \simeq G_2 \,\}.$$

Then

$$\left.\begin{array}{l}\text{if } G_1 \simeq G_2, \text{ then } |S| = n! \\ \text{if } G_1 \not\simeq G_2, \text{ then } |S| = 2n!\end{array}\right\} \tag{2.2}$$

Thus, to decide whether $G_1 \not\simeq G_2$, the prover needs to convince the verifier that $|S| = 2n!$. We shall shortly see how this can be done.

In the general case, we cannot assume that G_1 and G_2 have exactly $n!$ equivalence classes. To also handle this case, we recall that an *automorphism* of a graph G is a permutation $\pi \in S_n$ such that $G = \pi(G)$. Denote with $\mathrm{Aut}(G)$ the subgroup of S_n of all automorphism of G. If we then change the definition of S to

$$S = \{\, (\mathcal{H}, \pi) \mid (\mathcal{H} \simeq G_1 \vee \mathcal{H} \simeq G_2) \wedge \pi \in \mathrm{Aut}(\mathcal{H}) \,\},$$

then (2.2) still holds true (Exercise!).

We now want to describe a protocol in which the prover can convince the verifier that a certain set S has cardinality $2n!$ instead of $n!$. For this we assume that S is known to both the prover and the verifier, in the sense that membership in S can be certified easily: if $x \in S$, the prover can provide a certificate to the verifier to this effect.

In the following we shall describe what is called the *set lower-bound protocol* that allows the prover to certify the *approximate* size of S to the verifier.

Let $K = |S|$. The prover can compute this value and announce it to the verifier. The problem is then to *convince* the verifier about the correctness of this claim. For this, the set lower-bound protocol makes the verifier accept with high probability if $K = |S|$, and makes the verifier reject with high probability if $|S| \leq K/2$.

Before describing the protocol in detail, let us first introduce a technical notion.

2.12 Definition Let $\mathcal{H}_{n,k}$ be a set of functions from $\{0,1\}^n$ to $\{0,1\}^k$. We say that $\mathcal{H}_{n,k}$ is a set of *pairwise independent hash-functions* if for all $x, x' \in \{0,1\}^n$, $x \neq x'$, and $y, y' \in \{0,1\}^k$ we have

$$\Pr(h(x) = y \wedge h(x') = y') = 2^{-2k},$$

where $h \in \mathcal{H}_{n,k}$ is chosen uniformly at random. ◇

Clearly, $\mathcal{H}_{n,k}$ is pairwise independent iff when fixing two distinct string $x, x' \in \{0,1\}^n$ the random variable

$$h \mapsto (h(x), h(x'))$$

is uniformly distributed on $\{0,1\}^k \times \{0,1\}^k$.

It can be shown that sets of *efficiently computable* pairwise-independent hash-functions exist for all $k, n \in \mathbb{N} \setminus \{0\}$ (see exercises).

We are now ready to describe the set lower-bound protocol.

2.13 Protocol (Set Lower-Bound Protocol) Let $S \subseteq \{0,1\}^n$ such that membership in S can be efficiently verified with high probability. Let $K \in \mathbb{N}$ and assume that both the prover and the verifier know K. The prover wants to convince the verifier that $|S| \geq K$, and the verifier should reject with high probability if $|S| \leq K/2$.

Let $k \in \mathbb{N}$ such that $2^{k-2} < K \leq 2^{k-1}$ and let $\mathcal{H}_{n,k}$ be a set of pairwise independent hash-functions.

1. The verifier chooses uniformly at random some $h \in \mathcal{H}_{n,k}$ and some $y \in \{0,1\}^k$ and sends both to the prover.

2. The prover finds $x \in S$ such that $h(x) = y$. If such an x exists, it is sent to the verifier together with a certificate that $x \in S$. Otherwise, an arbitrary $x \in S$ is sent.

3. The verifier checks that $h(x) = y$ and that the certificate for $x \in S$ is valid. If this is the case, the verifier accepts, and rejects otherwise. ♢

Let $p = K/2^k$. If $|S| \leq K/2$, then $|h(S)| \leq p/2 \cdot 2^k$ and thus the verifier will accept with probability at most $p/2$. Conversely, if $|S| \geq K$, we can show that the acceptance probability is considerably larger than $p/2$. Using repetition, we can amplify these probabilities as desired.[2]

2.14 Lemma Let $S \subseteq \{0,1\}^n$ such that $|S| \leq 2^k/2$. Then

$$\Pr(\exists x \in S. h(x) = y) \geq \frac{3}{4}\frac{|S|}{2^k}$$

for randomly chosen $y \in \{0,1\}^k$ and $h \in \mathcal{H}_{n,k}$.

In particular, if $|S| = K$, then

$$\Pr(\exists x \in S. h(x) = y) \geq \frac{3}{4}p.$$

Proof Let $y \in \{0,1\}^*$. We shall show that

$$\Pr(\exists x \in S. h(x) = y) \geq \frac{3}{4}\frac{|S|}{2^k},$$

where h is chosen randomly.

Define for $x \in S$ the event E_x by

$$E_x = \{ h \in \mathcal{H}_{n,k} \mid h(x) = y \}.$$

Then

$$\Pr(\exists x \in S. h(x) = y) = \Pr\Big(\bigcup_{x \in S} E_x\Big) \tag{2.3}$$

By the Inclusion-Exclusion Principle we obtain that this is at least

$$\sum_{x \in S} \Pr(E_x) - \frac{1}{2} \sum_{\substack{x,x' \in S \\ x \neq x'}} \Pr(E_x \cap E_{x'}).$$

However, from pairwise independence we get $\Pr(E_x) = 2^{-k}$ (exercise!) and $\Pr(E_x \cap E_{x'}) = 2^{-2k}$. Thus, (2.3) is at least

$$\frac{|S|}{2^k} - \frac{1}{2} \cdot \frac{|S|^2}{2^{2k}} = \frac{|S|}{2^k} \cdot \left(1 - \frac{|S|}{2^{k+1}}\right) \geq \frac{3}{4} \cdot \frac{|S|}{2^k}$$

as required. □

[2] The following lemma is Claim 8.13.1 in Sanjeev Arora and Boaz Barak: *Computational Complexity: A Modern Approach – Internet Draft*, 2007, URL: http://theory.cs.princeton.edu/complexity/book.pdf.

Proof (Theorem 2.11, Sketch) The public-coin protocol for GNI now proceeds by repeating the set lower-bound protocol for a certain number of steps. The verifier accepts if at least $5/8 \cdot K/2^k$ iterations succeed (note that $5/8 = (3/4 + 1/2)/2$). Using Chernoff's bound, it can be shown that a fixed number of iterations is sufficient to reach completeness $2/3$ and soundness $1/3$. This shows $\mathsf{GNI} \in \mathsf{AM}[\ell]$ for some constant ℓ.

Finally, we can conduct all those repetitions *in parallel* and obtain $\mathsf{GNI} \in \mathsf{AM}[2]$. □

Note that the resulting public-coin protocol for GNI, in contrast to the private-coin protocol for GNI, does not have *perfect completeness* (i.e., completeness 1). However, it can be shown that a public-coin protocol for set lower-bound with perfect completeness can be constructed. Indeed, *every* private-coin protocol can be transformed into a public-coin protocol with perfect completeness and a similar number of rounds – even if it wasn't perfectly complete in the beginning!

2.4 IP = PSpace

It was long open whether there exists a meaningful characterization of IP in terms of known complexity classes. What we have already seen is $\mathsf{NP} \subseteq \mathsf{IP} \subseteq \mathsf{PSpace}$, and since we already know $\mathsf{GNI} \in \mathsf{IP}$ it seems that the first inclusion is strict. There were also persuasive arguments for the second inclusion to be strict, namely

- without randomization, $\mathsf{IP} = \mathsf{NP}$, and
- randomization does not seem to add more computational power; indeed, it is conjectured that $\mathsf{BPP} = \mathsf{P}$.

Thus the following result came as a surprise when it was shown in 1990.

2.15 Theorem (Shamir, Lund, Fortnow, Karloff, Nisan, 1990) $\mathsf{IP} = \mathsf{PSpace}$.

For the proof, we shall show that TQBF has a public-coin protocol with perfect completeness. Before we do so, however, we shall show that another hard problem also has polynomial-round interactive proof systems.

2.16 Theorem Let

$$\#\mathsf{SAT_D} := \{ \langle \varphi, k \rangle \mid \varphi \text{ a 3CNF formula with exactly } k \text{ satisfying assignments} \}.$$

Then $\#\mathsf{SAT_D} \in \mathsf{IP}$.

Clearly $\overline{\mathsf{SAT}}$ can be reduced to $\#\mathsf{SAT_D}$ in polynomial time. An easy corollary of this result is that $\mathsf{coNP} \subseteq \mathsf{IP}$. Furthermore, it is not hard to see that $\#\mathsf{SAT_D}$ is complete for the class #P of all counting problems of non-deterministic polynomial-time Turing machines (see next chapter). Thus, all problems in #P are also contained in IP.

The interactive protocol we shall devise for $\#\mathsf{SAT_D}$ will be an n-round protocol, where n is the number of variables in φ. The main idea is to use a technique called *arithmetization.*

Let φ be a 3CNF-formula and consider the field $\mathbb{F}_2$ of two elements. Let m be the number of clauses in φ and denote with $x_1, \dots, x_n$ the variables in φ. We can transform logical connectives into polynomials over $X_1, \dots, X_n$ by means of a mapping τ as follows:

$$\tau(x_i) = X_i,$$
$$\tau(\neg\psi) = (1 - \tau(\psi)),$$
$$\tau(\psi_1 \wedge \psi_2) = \tau(\psi_1) \cdot \tau(\psi_2),$$

where ψ, ψ_1, ψ_2 are Boolean formulas. In particular (and up to equivalence),

$$\tau(\psi_1 \vee \psi_2 \vee \psi_3) = 1 - (1 - \tau(\psi_1))(1 - \tau(\psi_2))(1 - \tau(\psi_3)).$$

Denote with $P_C(X_1, \dots, X_n)$ the polynomial $\tau(C)$ of the clause C of φ. Then P_C depends on at most three variables. Thus, the polynomial

$$P_\varphi(X_1, \dots, X_n) := \prod_{C \text{ clause of } \varphi} P_C(X_1, \dots, X_n)$$

is a polynomial of degree at most $3m$. Note that we represent P_φ as this product without expanding the individual factors. In this way, P_φ has a representation of size $\mathcal{O}(m)$.

Each satisfying assignment of φ is an assignment of the variables $X_1, \dots, X_n$ to values $b_1, \dots, b_n$ such that $P_\varphi(b_1, \dots, b_n) = 1$. Conversely, if an assignment for the variables $x_1, \dots, x_n$ does not satisfy φ, then $P_\varphi(b_1, \dots, b_n) = 0$. Indeed, the number $\#\varphi$ of satisfying assignment of φ is

$$\#\varphi = \sum_{b_1 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_\varphi(b_1, \dots, b_n).$$

In order to show $\#\mathsf{SAT_D} \in \mathsf{IP}$, it thus suffices to show that there is an interactive proof system deciding whether a given polynomial has exactly k non-roots over $\mathbb{F}_2$.

Let us consider the problem from a more abstract point of view. Let $g(X_1, \dots, X_n)$ be a polynomial of degree d that can be evaluated in polynomial time in the number of variables. Let $k \in \{0, \dots, 2^n\}$. We want to find an interactive proof for the fact that

$$k = \sum_{b_1 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(b_1, \dots, b_n). \tag{2.4}$$

Choose $p \in \{2^n + 1, \dots, 2^{2n}\}$ prime. Then (2.4) is true if and only if it is true in $\mathbb{F}_p$. Thus, from now on, we shall perform all our computations in $\mathbb{F}_p$.

To describe the protocol, we introduce another univariate polynomial $h(X)$ by

$$h(X) := \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(X, b_2, \dots, b_n).$$

Note that (2.4) is true if and only if $h(1) + h(0) = k$. The following protocol makes use of this idea to provide the desired interactive proof system.

2.17 Protocol (Sum-Check Protocol) Let $g(X_1, \dots, X_n)$ be a polynomial of degree d that can be evaluated in polynomial time, and let $k \in \mathbb{N}$. Assume both the prover and the verifier have agreed on some prime number $p \in \{ 2^n + 1, \dots, 2^{2n} \}$.

1. The verifier does the following:
   - If $n = 1$, check whether $g(0) + g(1) = k$. If so, accept, otherwise reject.
   - If $n > 1$, ask the prover to send $h(X)$.
2. The prover sends some polynomial $s(X)$ of degree at most d.
3. The verifier checks if $s(0) + s(1) = k$, and rejects if this is not true. Otherwise, the verifier picks $a \in \mathbb{F}_p$ uniformly at random. It then recursively asks the verifier to proof

$$s(a) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(a, b_2, \dots, b_n). \qquad \diamondsuit$$

It is clear that if (2.4) is true, the prover can make the verifier accept. In the case (2.4) does not hold, we have the following result.

2.18 Proposition Let V be the verifier as described in the Sum-Check Protocol. If (2.4) is not true, then V rejects with probability at least $\left(1 - \frac{d}{p}\right)^n$.

Proof We prove the claim by induction over n. The base case $n = 1$ is clear: V checks the claim directly and rejects with probability 1.

For the case $n > 1$, if the prover returns the correct polynomial h in the first step, then $h(0)+h(1) \neq k$ and the verifier rejects again with probability 1. Thus assume that the prover returns some polynomial $s(X) \neq h(X)$. Since $s(X) - h(X)$ has degree at most d, there are at most d elements a in $\mathbb{F}_p$ such that $s(a) = h(a)$. Thus, the probability to choose $a \in \mathbb{F}_p$ such that $s(a) \neq h(a)$ is at least $1 - \frac{d}{p}$. In this case, the prover is left with showing the false claim $s(a) = h(a)$. By induction hypothesis, this prove is rejected with probability at least $\left(1 - \frac{d}{p}\right)^{n-1}$. Thus, the probability that the verifier rejects is at least

$$\left(1 - \frac{d}{p}\right) \cdot \left(1 - \frac{d}{p}\right)^{n-1} = \left(1 - \frac{d}{p}\right)^n. \qquad \square$$

From the Sum-Check Protocol we can take some inspiration to show the main claim of this section, namely that TQBF has interactive proofs.

2.19 Theorem $\mathsf{TQBF} \in \mathsf{IP}$.

Proof We showcase the proof using the formula

$$\Psi = \forall x_1 \exists x_2 \forall x_3 \dots \exists x_n . \varphi(x_1, \dots, x_n)$$

(the general proof works in the same way, but with substantially more notational overhead). Let us extend out arithmetization approach to Ψ by defining

$$\begin{aligned}\tau(\forall X_i.p(X_1,\dots,X_n)) &= p(X_1,\dots,X_{i-1},0,X_{i+1},\dots,X_n)\\ &\quad\cdot p(X_1,\dots,X_{i-1},1,X_{i+1},\dots,X_n),\\ \tau(\exists X_i.p(X_1,\dots,X_n)) &= p(X_1,\dots,X_{i-1},0,X_{i+1},\dots,X_n)\\ &\quad+ p(X_1,\dots,X_{i-1},1,X_{i+1},\dots,X_n).\end{aligned}$$

Then the problem of deciding whether Ψ is satisfiable amounts to showing that

$$\tau(\Psi) = \prod_{b_1\in\{0,1\}}\sum_{b_2\in\{0,1\}}\prod_{b_3\in\{0,1\}}\cdots\sum_{b_n\in\{0,1\}} P_\varphi(b_1,\dots,b_n) \neq 0. \tag{2.5}$$

A first idea would be to use the Sum-Check Protocol and for each $\forall$ check $s(0)\cdot s(1) = k$ instead. However, the problem with this approach is that the polynomial h as defined before may have exponentially large degree. Then, the prover could not send the polynomial to the verifier in polynomial time.

To solve this issue, we observe that we only evaluate the polynomials for values in $x \in \{0,1\}$, and then the overall degree does not matter, because $x^k = x$. In this way, we can turn each polynomial $p(X_1,\dots,X_n)$ into a polynomial $q(X_1,\dots,X_n)$ such that each variable has degree at most one, such that

$$p(x_1,\dots,x_n) = q(x_1,\dots,x_n)$$

for all $x_1,\dots,x_n \in \{0,1\}$.

To achieve this transformation, we introduce the following *linearization operators* LX_i:

$$\begin{aligned}LX_i.p(X_1,\dots,X_n) &= X_i\cdot p(X_1,\dots,X_{i-1},1,X_{i+1},\dots,X_n)\\ &\quad+(1-X_i)\cdot p(X_1,\dots,X_{i-1},0,X_{i+1},\dots,X_n).\end{aligned}$$

Then $LX_i.p$ is linear in X_i and agrees with p for all choices of X_i in $\{0,1\}$. In particular,

$$LX_1.LX_2.\dots LX_n.p(X_1,\dots,X_n)$$

is a linear polynomial that agrees with p on all points in $\{0,1\}^n$. Thus (2.5) is equivalent to

$$\forall X_1.LX_1.\exists X_2.LX_1.LX_2.\dots\exists X_n.LX_1\dots LX_n.P_\varphi(X_1,\dots,X_n)\neq 0 \tag{2.6}$$

The size of this expression is polynomial in n.

We now describe a recursive interactive protocol to check (2.6). Suppose for this that we are given a polynomial

$$U(X_1,\dots,X_\ell) = Tg(X_1,\dots,X_k),$$

where $T \in \{\exists X_i, \forall X_i, LX_i \mid 1\le i\le n\}$. The prover wants to convince the verifier that

$$U(a_1,\dots,a_\ell) = C'$$

for some given number C' and $a_1,\dots,a_\ell \in \{0,1\}$. For this they interact as follows:

- if $T = \exists X_i$, then the verifier asks the prover to provide a polynomial $s(X_i)$ that is supposed to be $g(a_1, \dots, X_i, \dots, a_k)$. The verifier checks if $s(0) + s(1) = C'$ and rejects if not. Otherwise, the verifier picks uniformly at random some $a \in \mathbb{F}_p$ and asks the prover to show $s(a) = g(a_1, \dots, a_{i-1}, a, a_{i+1}, \dots, a_k)$.
- if $T = \forall X_i$, then the verifier proceeds in the same way, but checks $s(0) \cdot s(1) = C'$ instead.
- if $T = LX_i$, the verifier asks the prover to send some polynomial $s(X_i)$ that is supposed to be $g(a_1, \dots, X_i, \dots, a_k)$. The verifier checks that U is linear in X_i and that $a_i s(1) + (1 - a_i)s(0) = C'$, and rejects if this fails. Otherwise, it picks some $a \in \mathbb{F}_p$ and asks the prover to show $s(a) = g(a_1, \dots, a_{i-1}, a, a_{i+1}, \dots, a_k)$.

The correctness of this protocol is shown as in the case of the Sum-Check Protocol, but this time the induction is over the number of operators. More precisely, assuming that this protocol can convince the verifier of $g(a_1, \dots, a_k) = k_1$ with completeness 1 and soundness ε, it is easy to see that the verifier can be convinced of $U(a_1, \dots, a_\ell) = k_2$ with completeness 1 and soundness at most $\varepsilon + \frac{d}{p}$. ☐

An interesting curiosity that immediately follows from $\mathsf{IP} = \mathsf{PSpace}$ is the following result.

2.20 Corollary If $\mathsf{PSpace} \subseteq \mathsf{P/poly}$, then $\mathsf{PSpace} = \mathsf{MA}$.

Proof If $\mathsf{PSpace} \subseteq \mathsf{P/poly}$, we can replace the prover by a circuit family $(C_n \mid n \in \mathbb{N})$ of polynomial size. This can be done because the computational power of the prover is effectively bounded above by PSpace. In an MA protocol for TQBF, for inputs of length n the first (and only) thing the prover does is to provide the verifier with the circuit C_n. From then on, the verifier can interact with the circuit as described in the interactive protocol for TQBF. Therefore, no more interaction between the prover and the verifier is necessary.

Note that if the circuit provided by the prover is not C_n, then the correctness of the protocol for TQBF still ensures that the result of the verifier interacting with this other circuit leads to a correct result. ☐

Another curiosity of $\mathsf{IP} = \mathsf{PSpace}$ is that it is a natural counterexample to the *random oracle hypothesis*[3]: call an oracle $\mathcal{O}$ *random* if for every new input x it tosses a fair coin to decide whether $x \in \mathcal{O}$ or not. Then the random oracle hypothesis asserts that if $\mathcal{C}_1$ and $\mathcal{C}_2$ are "acceptable" complexity classes, then $\mathcal{C}_1^{\mathcal{O}} = \mathcal{C}_2^{\mathcal{O}}$ with probability 1 for random oracles $\mathcal{O}$ if and only if $\mathcal{C}_1 = \mathcal{C}_2$.

The validity of the random oracle hypothesis would immediately imply $\mathsf{P} \neq \mathsf{NP}$, because it has been shown that $\mathsf{P}^{\mathcal{O}} \subsetneq \mathsf{NP}^{\mathcal{O}}$ with probability 1 for random oracles $\mathcal{O}$. However, it turns out that the random oracle hypothesis is not true, and an "acceptable" counterexample are the classes IP and PSpace: $\mathsf{IP} = \mathsf{PSpace}$, but for random oracles $\mathcal{O}$ we have $\mathsf{IP}^{\mathcal{O}} \subsetneq \mathsf{PSpace}^{\mathcal{O}}$ with probability one.[4]

[3] Charles Bennett and John Gill: "Relative to a Random Oracle A, P != NP != co-NP with Probability 1", in: *SIAM Journal of Computation* 10 (1 1981), pp. 96–113.

[4] Richard Chang et al.: "The Random Oracle Hypothesis is False", in: *Journal of Computer and System Sciences* 49 (1 1994), pp. 24–39.

2.5 Outlook: Multi-Prover Systems

It is possible to extend the notion of interactive proofs to protocols with more than one prover. The class of all languages decidable by *multi-prover* interactive proofs is call MIP. The main point about having more than one prover is that these provers are *not* allowed to communicate during the interaction with the verifier. They are allowed, though, to communicate before the interaction starts, for example to agree on a common answering strategy.

The analogy here is a police interrogation of two (or more) criminals in separate rooms: although the criminals may agree on a common strategy in before, a thorough interrogation will eventually reveal inconsistencies in their answers.

Indeed, having multiple provers allows the verifier to force *non-adaptivity* of her queries: to make sure the answer to some query q to one prover does not depend on the answers of the previous queries to the same prover, the verifier can just send q to the *other* prover and check whether its answer coincides with the one of the first prover.

It turns out that having just another prover is already enough: it can be shown that having polynomially many provers is as good as having just two. Furthermore, elaborating on the technique of forcing non-adaptivity, it is possible to give a characterization of MIP similar to the one of IP.

2.21 Theorem $\mathsf{MIP} = \mathsf{NExpTime}$.

3 Counting Complexity

In some cases, instead of seeking to answer whether some solution for a given problem exists, it is more important to know what the *number* of solutions is. The corresponding notion of a *counting problem* as well as its complexity theoretic investigation are the topics of the last part of this lecture.[1]

3.1 Counting problems and the class #P

3.1 Example Let G be a graph with nodes $\{1, \dots, n\}$. We want to consider the *graph reliability problem*: assuming that each edge of G can fail with probability $1/2$, what is the probability that there still exists a path from 1 to n?

Clearly, under this failure assumption each subgraph on all n nodes is equally likely to occur. Thus, the probability that a path from 1 to n exists is given by

$$\frac{1}{2^{|E|}} \cdot \big(\text{number of subgraphs of } G \text{ containing a path from } 1 \text{ to } n\big). \qquad \diamondsuit$$

In general, we can consider counting versions of classical decision problems:

#SAT*:* compute for some given propositional formula φ the number of satisfying assignments.

#Cycle*:* given a directed graph G, count the number of *simple* cycles in G.

Clearly, #SAT and #Cycle are at least as difficult as their corresponding decision versions. In the case of SAT this immediately implies that #SAT is hard. On the other hand, finding a simple cycle in a directed graph can be done easily (in linear time) and thus #Cycle could also be an easy problem. Surprisingly, this is not the case.

3.2 Theorem If #Cycle $\in$ FP, then P $=$ NP.

This shows that counting version for some easy decision problems can be hard. On the other hand, not all counting versions of decision problems are of this form. For example, counting *spanning trees* of arbitrary graphs can be done in polynomial time, c.f. Kirchhoff's Theorem.

[1] This part is based on Christos H. Papadimitriou: *Computational Complexity*, Addison-Wesley, 1995, Chapter 18, Sanjeev Arora and Boaz Barak: *Computational Complexity A Modern Approach*, Cambridge University Press, 2009, Chapter 17, and Sanjeev Arora and Boaz Barak: *Computational Complexity: A Modern Approach – Internet Draft*, 2007, URL: http://theory.cs.princeton.edu/complexity/book.pdf, Chapter 9.

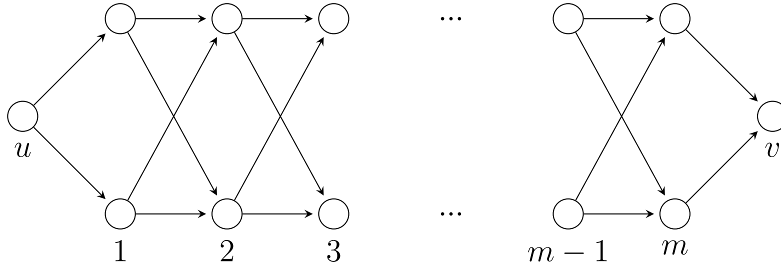


Figure 3.1: Edge from u to v in G'

Proof Suppose #Cycle $\in$ FP, i.e., counting the number of simple cycles in a directed graph can be done in deterministic polynomial time. We shall show that in this case DirectedHamiltonianCycle can be solved in polynomial time as well, implying P $=$ NP.

Let G be a directed graph with n nodes. We shall construct another graph G' of polynomial size such that G has a Hamiltonian cycle if and only if G' has at least n^{n^2} simple cycles.

Let (u, v) be some edge in G. The corresponding edge in G' is shown in Figure 3.1, where $m = \lceil n \log_2 n \rceil$. Then each simple cycle in G of length ℓ corresponds to $(2^m)^\ell$ simple cycles in G'.

Thus, if G has a Hamiltonian cycle, then G' has at least

$$(2^m)^n = (2^{n \log n})^n = (2^{\log n^n})^n = (n^n)^n = n^{n^2}$$

many cycles. Conversely, if G does not have a Hamiltonian cycle, then the longest cycle in G has length at most $n - 1$. Hence there are at most

$$(2^m)^{n-1} \cdot n^{n-1} = n^{n(n-1)} \cdot n^{n-1} = n^{(n+1)(n-1)} = n^{n^2-1}$$

many cycles in G'. This concludes the proof. $\square$

The goal of our investigation now is to distinguish between easy and hard counting problems. For this we introduce a new complexity class.

3.3 Definition Let $I \subseteq \Sigma^*$ and let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation. The *counting problem* associated with (I, R) is: given $x \in I$, what is the number of elements y such that $(x, y) \in R$? We write $\#_R(x)$ for this number. The set of all counting problems associated with polynomially balanced polynomially decidable binary relations is called #P (pronounced "number P" or "sharp P"). $\diamondsuit$

If clear from the context, we shall omit the explicit reference to the set I of instances of a counting problem (I, R). Examples for problems in #P are #SAT, #Cycle, graph reliability, counting spanning trees, and counting the number of Hamiltonian paths.

Indeed, another way of looking at #P is as the set of counting problems associated to decision problems in NP. From this we immediately obtain that a problem R is in #P if and only if there exists a non-deterministic polynomial-time Turing machine M such that

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

The main question now is whether all problems in #P can be solved in polynomial time, i.e., whether #P $\subseteq$ FP. Here we interpret counting problems (I, R) as computation problems of the form

$$R' = \{ (x, \#_R(x)) \mid x \in I \}.$$

Clearly, R' is a polynomially balanced binary relation.

As always, the question #P $\subseteq$ FP is open. Indeed, since computing the number of certificates is at least as hard as checking the existence of one certificate, #P $\subseteq$ FP readily implies P $=$ NP. However, it is unknown whether the converse is also true. What is clear is that P $=$ PSpace implies #P $\subseteq$ FP, because counting the number of polynomially long certificates can be done in polynomial space.

We thus do not completely understand which impact the relationship between P and NP has on the relationship between #P and FP. But for another class very akin to NP more can be said.

Recall that the class PP consists of all languages for which a polynomial-time probabilistic Turing machine can guess more often right than wrong. Put more formally, $L \in$ PP if and only if there exists some polynomial-time deterministic Turing machine M and some polynomial p such that

$$x \in L \iff \left|\{ y \in \{ 0, 1 \}^{p(|x|)} \mid (x, y) \in \mathcal{L}(M) \}\right| \geq \frac{1}{2} \cdot 2^{p(|x|)}. \tag{3.1}$$

Intuitively, PP corresponds to computing the most significant bit of the result of some #P-problem. We can put this idea into use by showing the following result (recall NP $\subseteq$ PP $\subseteq$ PSpace).

3.4 Theorem P $=$ PP if and only if #P $\subseteq$ FP.

Proof If #P $\subseteq$ FP, we can compute the right-hand value in (3.1) in polynomial time. This immediately implies P $=$ PP.

Suppose conversely that P $=$ PP. Let $(I, R) \in$ #P and let p be a polynomial such that $(x, y) \in R$ implies $|y| < p(|x|)$. Then $0 \leq \#_R(x) < 2^{p(|x|)}$ for all $x \in I$. We shall construct a polynomial-time probabilistic Turing machine $\mathcal{K}_{N,R}$ for each $N \in \{ 0, \ldots, 2^{p(|x|)} \}$ such that

$$\#_{\mathcal{K}_{N,R}}(x) = N + \#_R(x),$$

where

$$\#_{\mathcal{K}_{N,R}}(x) \coloneqq \left|\{ y \in \{ 0, 1 \}^{p(|x|)+1} \mid (x, y) \in \mathcal{L}(\mathcal{K}_{N,R}) \}\right|$$

Since P $=$ PP, we can compute in polynomial time whether

$$N + \#_R(x) = \#_{\mathcal{K}_{N,R}}(x) \geq \frac{1}{2} \cdot 2^{p(|x|)+1} = 2^{p(|x|)},$$

and using binary search we can thus compute in polynomial time the value N' such that $N' + \#_R(x) = 2^{p(|x|)}$. In particular, the value $\#_R(x) = 2^{p(|x|)} - N'$ can be computed in polynomial time.

Now, obtaining the machine $\mathcal{K}_{N,R}$ is not very difficult:

$\mathcal{K}_{N,R}(x, y_0 y_1 \dots y_{p(|x|)}) \to$
if $y_0 = 0$ **then**
  if $[y_1 y_2 \dots y_{p(|x|)}]_2 < N$ **then**
    accept
  else
    reject
else
  return $(x, y_1 y_2 \dots y_{p(|x|)}) \in R$

Clearly, $\mathcal{K}_{N,R}$ runs in polynomial time and $\#_{\mathcal{K}_{N,R}}(x) = N + \#_R(x)$ as required. □

The class #P cannot be put directly into relation with other classical complexity classes, because it does not consist of decision problems. However, there is a famous result that relates the polynomial hierarchy to decision problems solvable in polynomial time with oracle access to #P.

3.5 Theorem (Toda, 1989) $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$.

The proof is quite long and involved, and contains interesting applications of ideas not discussed in this lecture. For example, in the proof a complexity class $\oplus$P ("parity-P ") is introduced that consists of all counting problems for which it can be determined in polynomial time whether the number of solutions is *even*. In other words, for the counting problems in $\oplus$P the *last bit* of the solution can be computed in polynomial time. This class naturally complements the class PP as the class of counting problems for which the *first* bit of the solution can be computed in polynomial time.

3.2 #P-completeness and Valiant's Theorem

We have again the usual situation: while some problems in #P seem to be computationally hard (like #SAT), we are not able to show they cannot be solved in polynomial time. We therefore apply our usual idea of *completeness* to #P to be able to at least identify the "most complicated" problems in #P.

For this approach to make sense, we first need to specify the reduction to use. Since counting problems are very akin to function problems, we just adapt the corresponding reduction notion.

3.6 Definition Let A and B be two counting problems. A *reduction* from A to B is a pair (f, g) of logspace-computable functions such that

$$\#_A(x) = g(\#_B(f(x)))$$

for all $x \in I$. A problem $(I, R) \in$ #P is called *complete* for #P if for all $S \in$ #P there exists a reduction from S to R. ♢

Reductions between function problems can easily be obtained from special kinds of polytime reductions between decisions problems.

3.7 Definition Let $L_1, L_2 \in \mathsf{NP}$ be represented as

$$L_1 = \{\, x \in \Sigma^* \mid \exists y. |y| \leq p_1(|x|) \wedge (x, y) \in \mathcal{L}(\mathcal{K}_1) \,\}$$
$$L_2 = \{\, x \in \Sigma^* \mid \exists y. |y| \leq p_2(|x|) \wedge (x, y) \in \mathcal{L}(\mathcal{K}_2) \,\}$$

for polynomials p_1, p_2 and deterministic Turing machines $\mathcal{K}_1, \mathcal{K}_2$. A polytime reduction f from L_1 to L_2 is called *parsimonious* if for all $x \in \Sigma^*$ the number of y such that $(x, y) \in \mathcal{L}(\mathcal{K}_1)$ is equal to the number of y' such that $(f(x), y') \in \mathcal{L}(\mathcal{K}_2)$. $\diamondsuit$

Parsimonious reductions turn up very frequently in proofs of NP-completeness. Indeed, when showing NP-completeness for various classical problems, usually (actually: almost always) parsimonious reductions are employed. In particular, the reduction from the proof of the Cook-Levin Theorem from the halting problem of non-deterministic polytime Turing machines to satisfiability of propositional formulas is parsimonious. Using this fact, one can easily obtain the following result.

3.8 Theorem #SAT is #P-complete.

Note that instead of requiring for parsimonious reductions the number of certificates to be the same, it is also sufficient to require the number of certificates for each instance x of L_1 is easily computable from the number of certificates of the instance $f(x)$ of L_2.

Because of the plenitude of parsimonious reductions, many counting problems corresponding to NP-complete decision problems are naturally #P-complete. However, there are also #P-complete problems that arise as counting variants of problems that are actually in P. A prominent example for this is finding *perfect matchings* in bipartite graphs. The resulting counting problem is to compute the permanent of binary matrices, i.e., matrices that only contain entries 0 or 1.

Let $G = (V, E)$ be an undirected graph. Recall that G is called *bipartite* if there exist disjoint $U_1, U_2 \neq \emptyset$ such that $U_1 \cup U_2 = V$ and $E \subseteq \{\, \{\, u_1, u_2 \,\} \mid u_1 \in U_1, u_2 \in U_2 \,\}$. If G is bipartite, we shall also denote it by $G = (U_1, U_2, E)$.

A matching in the graph G is a subset $M \subseteq E$ such that each two distinct edges in M are disjoint. The matching M is called *perfect* if every node in V occurs in some (and thus exactly one) edge in M; in other words, $\bigcup M = V$. Computing some perfect matching of G, if it exists, can be done in polynomial time. For counting perfect matchings, however, no fast algorithm is known.

Perfect matchings in G can be characterized algebraically. Let $|U_1| = |U_2| = n$ and let $U_1 = \{\, u_1, \dots, u_n \,\}$, $U_2 = \{\, v_1, \dots, v_n \,\}$. Define the $n \times n$-matrix $A^G \in \{\, 0, 1 \,\}^{n \times n}$ by

$$A^G(i, j) := \begin{cases} 1 & \text{if } \{\, u_i, v_j \,\} \in E, \\ 0 & \text{otherwise.} \end{cases}$$

A perfect matching M in G then correspond to a *bijective* mapping σ from U_1 to U_2 by virtue of $\sigma(u_i) = v_j$ if $\{\, u_i, v_j \,\} \in E$. Such a mapping σ can be considered as a permutation $\sigma \in S_n$ by defining $\sigma(i) = j$. In this case,

$$\prod_{i=1}^{n} A^G(i, \sigma(i)) = 1.$$

Conversely, if some $\tau \in S_n$ does not define a perfect matching in the way just described, then

$$\prod_{i=1}^{n} A^G(i, \tau(i)) = 0.$$

Thus, the number of perfect matchings in G is precisely

$$\sum_{\sigma \in S_n} \prod_{i=1}^{n} A^G(i, \sigma(i)) = \text{perm } A^G.$$

Note the similarity of perm A^G to the determinant of A^G:

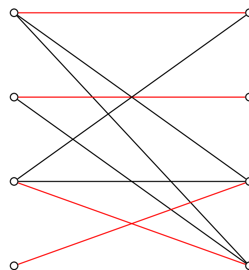
$$\det A^G := \sum_{\sigma \in S_n} \text{sgn}(\sigma) \cdot \prod_{i=1}^{n} A^G(i, \sigma(i)).$$

In other words, the determinant of A^G differs in the occurrence of the extra factor sgn(σ) in the sum. This change, appearing negligible, has severe consequences: the determinant can be computed in polynomial time, while computing the permanent is a problem that is #P-complete.

Permanents of binary matrices allow for an alternative graph-theoretic interpretation in terms of *cycle covers*. For this let us consider the matrix A^G as the adjacency matrix of a graph G' on n vertices $\{ w_1, \dots, w_n \}$. Then $\{ w_i, w_j \} \in E$ if and only if $A^G(i,j) = 1$. A *cycle cover* in G' is a set of node-disjoint cycles that cover all nodes. It is easy to see that a perfect matching in G corresponds to a cycle cover in G' and vice versa. Consequently, perm(A^G) is the number of cycle covers in G'.

Let us illustrate this correspondence by means of a small example. We leave the formal description as an exercise.

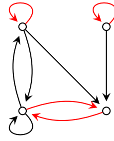
3.9 Example Let us consider the following bipartite graph G and the perfect matching marked by the red edges



The matrix A^G of this graph is (up to simultaneous reordering of the rows and columns)

$$A^G = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

and the graph G' as described before is



and the cycle cover corresponding to the perfect matching in G is again marked by red edges. ◇

It is not hard to see that this correspondence can be generalized to bipartite multigraphs with weighted edges, where the weights are taken from $\mathbb{Z}$. Let G be such a graph. Define the matrix A^G to have at position (i, j) the sum of the weights of the edges from v_i to w_j, and let G' be the edge-weighted graph with A^G as adjacency matrix. Define the *weight* of a cycle cover in G' to be the product of the weights of the edges contained in this cycle cover. Then the permanent $\mathrm{perm}(A^G)$ is just the sum of the weights of the cycle covers in G'.

(Note: the graph does not need to be a proper multigraph: two edges between nodes v and w can be replaced by one edge with weight the sum of the weights of the original edges.)

We are now ready to prove the main result of this section.

3.10 Theorem (Valiant, 1979) Computing the permanent of a binary matrix is #P-complete.

The proof consists of two steps.

3.11 Proposition For each 3CNF-formula φ there exists a matrix $A \in \{-1, 0, 1, 2, 3\}^{n \times n}$ such that

$$\mathrm{perm}(A) = 4^m \cdot \#\varphi,$$

where m is the number of literals in φ. Furthermore, the matrix A can be computed in logarithmic space.

3.12 Proposition For each matrix $A \in \mathbb{Z}^{n \times n}$ with entries in $\{-1, 0, 1, 2, 3\}$ there exists a binary matrix B and a number $N \in \mathbb{N}$ such that

$$\mathrm{perm}(A) = \mathrm{perm}(B) \bmod N,$$

where to modulo operator uses representatives in $\{-n!, \ldots, n!\}$. Both the matrix B and the number N can be computed in logarithmic space.

Proof (Theorem 3.10) We reduce #SAT to computing the permanent of 0-1-matrices. Let φ be a 3CNF-formula with m literals. By Proposition 3.11 there exists a matrix A such that $\mathrm{perm}(A) = 4^m \cdot \#\varphi$. Furthermore, all entries in A are in $\{-1, 0, 1, 2, 3\}$. By Proposition 3.12 there exists a matrix B and some $N \in \mathbb{N}$ such that $\mathrm{perm}(A) = \mathrm{perm}(B) \bmod N$, and B has only entries 0 or 1.

We now define the reduction (f, g) from #SAT to computing the permanent of 0-1-matrices as follows:

$$f(\varphi) = B, \quad g(s) = (s \bmod N)/4^m.$$

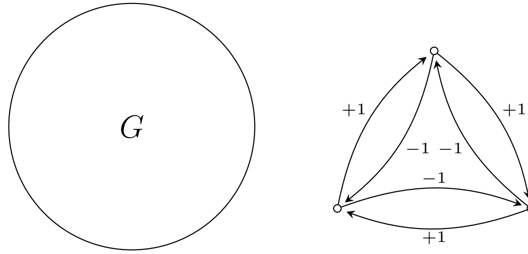
Then f and g are computable in logarithmic space. Furthermore,

$$\begin{aligned} g(\operatorname{perm}(f(\varphi))) &= g(\operatorname{perm}(B)) \\ &= (\operatorname{perm}(B) \bmod N)/4^m \\ &= \operatorname{perm}(A)/4^m \\ &= \#\varphi \cdot 4^m/4^m \\ &= \#\varphi \end{aligned}$$

and the proof is complete. □

Before we are going to prove Proposition 3.11, let us first consider an example.

3.13 Example Let us consider the graph H consisting of these two subgraphs



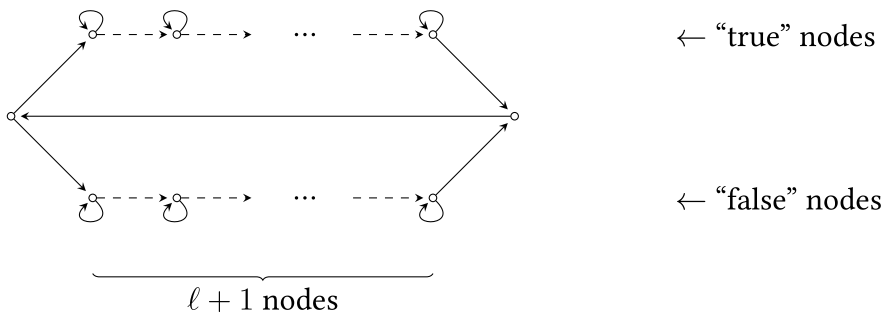
Then each cycle cover of H consists of a cycle cover of G and one cycle cover in the smaller graph. In particular, for each cycle cover of G of weight w, there exists exactly two cycle covers of G' of weights w and $-w$. Thus, the sum of all cycle covers of G' is zero, and thus $\operatorname{perm}(A^H) = 0$ for A^H being the adjacency matrix of H. ♢

For what follows, we shall adopt the following convention: all graphs are complete graphs with weighted edges. Edges not shown have weights 0, and edges without labels have weight 1.

Proof (Proposition 3.11) Let φ be a 3CNF-formula with n variables and ℓ clauses. We shall construct the graph G such that there are two types of cycle covers: those that correspond to satisfying assignments of φ and those that don't. For the latter, we shall use the trick of the previous example to ensure that these do not add to the permanent of (the adjacency matrix of) G.

We first introduce the following two types of gadgets.

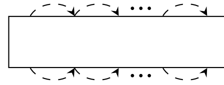
Variable gadget: for each variable occurring in φ we introduce the following subgraph:



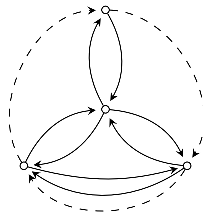
This subgraph has exactly two cycle covers: one for setting the variable to true (using the upper cycle) and one for setting it to false (using the lower cycle).

Variable gadgets have two types of edges: *internal* edges (depicted as solid lines) and *external* edges (depicted as dashed lines). The internal edges will not be connected to any other gadget. The external edges will be connected to clause gadgets (to be introduced shortly) corresponding to whether the variable appears positively or negatively in that clause.

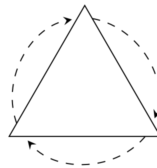
We shall depict variable gadgets schematically as follows:



Clause gadget: for each clause in φ introduce the following subgraph:



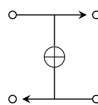
Again, the external edges (shown as dashed edges) are the only ones that are connected to other edges. We shall depict the clause gadget schematically by



The crucial property of the clause gadget is now the following: all cycle covers of this subgraph have to omit at least one external edge. Furthermore, for each proper subset A of the set of external edges, there exists exactly one cycle cover in this subgraph that traverses all the edges in A and omits all other external edges.

The idea of the proof is now as follows. Suppose we have a means to express that in a cycle cover of a graph $\mathcal{H}$, *exactly one* of two edges (u, u') and (v, v') is used. More precisely, we want to ensure that all *other* cycle covers (either using both these edges or none) do not add to the final count of the overall weight of all cycle covers. This will be done by ensuring that these covers contribute an overall weight of zero.

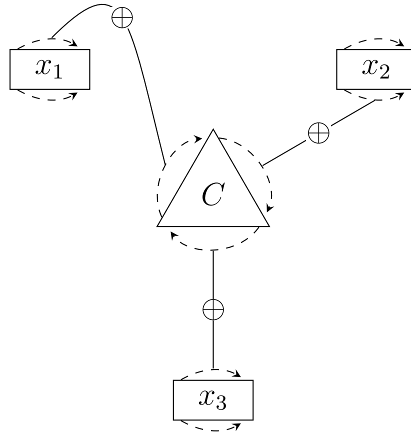
Let us express the fact that of the edges (u, u') and (v, v') exactly one is used schematically as follows:



Let us call this gadget *XOR-gadget.*

Using this new gadget, we transform the formula φ into the following graph $\mathcal{G}_\varphi$: for each clause C in φ we identify each variable x occurring in C with an external edge of the corresponding clause gadget of C. We then connect the external edge of the clause gadget of C with an unused external edge of the variable gadget of x using the XOR gadget. We choose a "true" external edge of the variable gadget of x if x appears positively in C, and a "false" external edge otherwise.

For example, the clause $C = (x_1 \vee \neg x_2 \vee x_3)$ is transformed into the following graph $\mathcal{G}_C$:

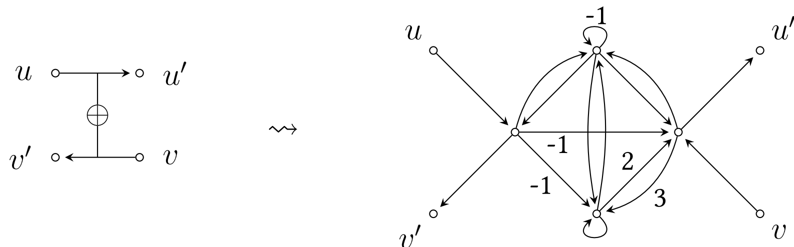


To count the cycle covers of $\mathcal{G}_C$ (or, equivalently, to compute the permanent of its adjacency matrix) it is sufficient to count only those cycle covers that respect the XOR gadgets. All such cycle covers traverse the clause gadget for C and thus omit at least one external edge. Setting the variable corresponding to this omitted edge to the value given by the connection to the variable gadget yields a satisfying assignment of C. Conversely, each satisfying assignment of C corresponds to exactly one cycle cover in $\mathcal{G}_C$ that respects the XOR gadgets.

It can be seen easily that if more than one clause is given the overall construction of the variable gadgets will ensure that each cycle cover assigns exactly one value to each variable. Furthermore, some literal in each clause is assigned true. Therefore, each cycle cover of $\mathcal{G}_\varphi$ corresponds one-to-one to a satisfying assignment of φ.

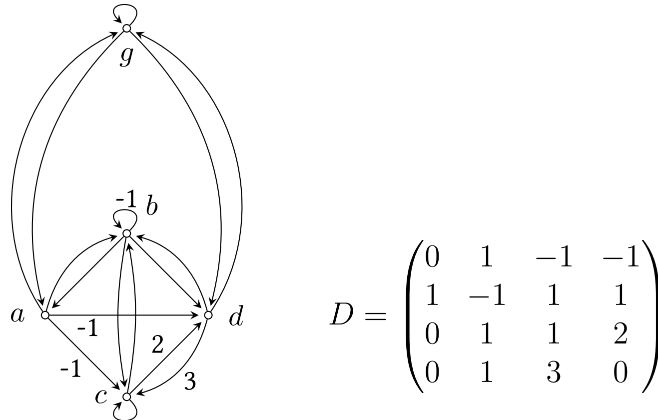
We shall now show how to implement the XOR gadget. In the resulting construction, all cycle covers that do not obey this gadget will sum up to a weight of zero. All other cycle covers, however, will be multiplied by a constant factor of 4. Therefore, the resulting graph will have a permanent of $4^m \cdot \#\varphi$, since all m XOR gadgets need to be traversed.

XOR gadget: suppose we are given two edges (u, u') and (v, v') in a graph $\mathcal{H}$ and we want to ensure that all cycle covers that contribute to the final permanent contain exactly one of them. For this we replace the two edges by the following gadget.



This gadget has the following remarkable property: if this gadget is embedded into the original graph $\mathcal{H}$ instead of the two edges (u, u'), (v, v') to obtain a new graph $\mathcal{H}'$, then each cycle cover in $\mathcal{H}$ of weight w that uses exactly one of the original two edges results in a cycle cover in $\mathcal{H}'$ of weight $4w$. Moreover, every other cycle cover in $\mathcal{H}$ results in a cycle cover in $\mathcal{H}'$ of weight zero.

We can illustrate this as follows: suppose we represent the "rest of $\mathcal{H}$" by a single node g. Then the graph $\mathcal{H}$ looks as follows:



$$D = \begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 3 & 0 \end{pmatrix}$$

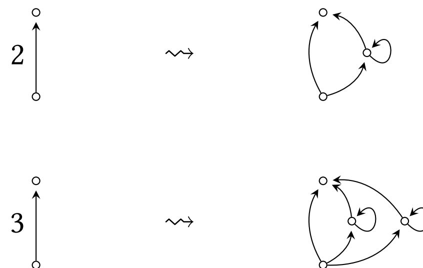
The matrix D denotes the adjacency matrix of $\mathcal{H}$ with the node g deleted.

Then

- the permanent of $\mathcal{H}$ is 8;
- a weight of 4 comes from traversing edges (g, d) and (a, g): if we delete the first row (corresponding to node a) and the last column (node d) of D, then the permanent of the resulting matrix is 4;
- a weight of 4 comes from traversing edges (d, g) and (g, a): deleting the first column and last row of D results in a permanent of 4;
- all other cycle covers contribute weight 0:
  - using the loop at g results in a cycle cover of weight zero, because $\mathrm{perm}(D) = 0$;
  - using the cycles (a, g), (g, a) or (g, d), (d, g) results in a permanent of zero.

Therefore, the proposed gadget is as required and the proof is finished. □

Proof (Proposition 3.12) Let us represent the matrix A as an edge-labeled directed graph. Each edge labeled with 2 or 3 can be replace by the following subgraphs:

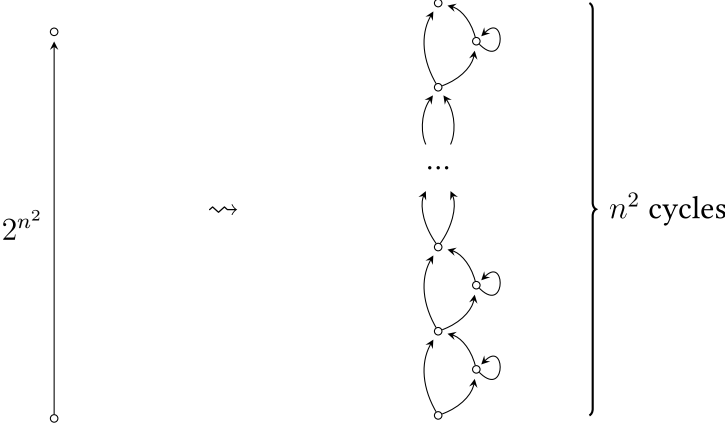


It is easy to see that the permanent of A is invariant under these replacements. Because of this we can assume without loss of generality that A has only entries from $\{-1, 0, 1\}$. In this case, $\text{perm}(A) \in \{-n!, \ldots, n!\}$. Therefore,

$$\text{perm}(A) = \text{perm}(A) \bmod N$$

for all $N > n!$.

Let $N = 2^{n^2} + 1$; then $N > n!$. Calculating modulo N, we can replace the weights -1 by 2^{n^2}. Those weights can in turn be replaced by unit edges as follows.



Let B be the adjacency matrix of the resulting graph. This matrix can be computed in polynomial time in the size of the original matrix A. Moreover,

$$\text{perm}(A) = \text{perm}(B) \bmod N,$$

and the proof is done. □