Computational Complexity Theory

Markus Bläser
Universität des Saarlandes

Draft—January 23, 2014 and forever

1 Simple lower bounds and gaps

Complexity theory is the science of classifying problems with respect to their usage of resources. Ideally, we would like to prove statements of the form "There is an algorithm with running time $O(t(n))$ for some problem P and every algorithm solving P has to use at least $\Omega(t(n))$ time". But as we all know, we do not live in an ideal world.

In this chapter, we prove some simple lower bounds. These bounds will be shown for natural problems. (It is quite easy to show lower bounds for "unnatural" problems that are obtained by diagonalization, like in the time and space hierarchy theorems.) Furthermore, these bounds are unconditional. While showing the NP-hardness of some problem can be viewed as a lower bound, this bound relies on the assumption that $\mathsf{P} \neq \mathsf{NP}$.

Unfortunately, the bounds in this chapter will be rather weak. But we do not know any other unconditional lower bounds for "easy" natural problems. While many people believe that SAT does not have a polynomial time algorithm, even proving a statement like $\mathtt{SAT} \notin \mathsf{DTime}(n^3)$ seems to be out of reach with current methods. We will encounter quite a few such easy looking and "almost" obvious statements that turned out to be really hard and resisted many attacks in the last fifty years.

1.1 A logarithmic space bound

Let $\mathtt{LEN} = \{a^n b^n \mid n \in \mathbb{N}\}$. LEN is the language of all words that consists of a sequence of as followed by a sequence of b of equal length. This language is one of the examples for a context-free language that is not regular. We will show that LEN can be decided with logarithmic space and that this amount of space is also necessary. The first part is easy.

Exercise 1.1 *Prove:* $\mathtt{LEN} \in \mathsf{DSpace}(\log)$.

A *small configuration* of a Turing machine M consists of the current state, the content of the work tapes, and the head positions of the work tapes. In contrast to a configuration, we neglect the position of the head on the input tape and the input itself. Since we only consider space bounds, we can assume that M has only one work tape (beside the input tape).

Exercise 1.2 *Let M be an s space bounded 1-tape Turing machine described by $(Q, \Sigma, \Gamma, \delta, q_0, Q_{\mathrm{acc}})$. Prove that the number of small configurations on*

inputs of length m is at most

$$|Q| \cdot |\Gamma|^{s(m)} \cdot (s(m) + 2).$$

If $s = o(\log)$, then the number of small configurations of M on inputs of length $m = 2n$ is $< n$ for large enough n by the previous exercise.

Assume that there is an s space bounded deterministic 1-tape Turing machine M with $s = o(\log)$ and $L(M) = \texttt{LEN}$. We consider an input $x = a^p b^n$ with $p \geq n$ and n large enough such that the number of small configurations is $< n$.

Excursus: Onesided versus twosided infinite tapes

In many books, the work tape of a Turing machine is assumed to be twosided infinite. Sometimes proofs get a little easier if we assume that the work tapes are just onesided infinite. The left end of the tape is marked by a special symbol \$ that the Turing machine is not allowed to change and whenever it reads the \$, it has to go to the right. To the right, the work tapes are infinite.

Exercise 1.3 *Show that every Turing machine M with twosided infinite work tapes can be simulated by a Turing machine M' with (the same number of) onesided infinite work tapes. If M is t time and s space bounded, then M' is $O(t)$ time and $O(s)$ space bounded. (Hint: "Fold" each tape in the middle and store the two halves on two tracks.)*

Often, the extra input tape is assumed to be twosided infinite. The Turing machine can leave the input and read plenty of blanks written on the tape (but never change them). But we can also prevent the Turing machine from leaving the input on the input tape as follows: Whenever the old Turing machine enters one of the two blanks next to the input, say the one on the lefthand side, the new Turing machine does not move its head. It has a counter on an additional work tape that is increased for every step on the input tape to the left and decreased for every step on the input tape to the right. If the counter ever reaches zero, then the new Turing machine moves its head on the first symbol on the input and goes on as normal. How much space does the counter need? No more than $O(s(n))$, the space used by the old Turing machine. With such a counter we can count up to $c^{s(n)}$, which is larger than the number of configurations of the old machine. If the old machine would stay for more steps on the blanks of the input tape, then it would be in an infinite loop and the new Turing machine can stop and reject. The time complexity at a first glance goes up by a factor of $s(n)$, since increasing the counter might take this long. There is amortized analysis but the Turing machine might be nasty and always move back and forth between two adjacent cells that causes the counter to be decreased and increased in such a way that the carry affects all positions of the counter. But there are clever redundant counters that avoid this behavior.

We assume in the following that the input tape of M is onesided infinite and the beginning of the input is marked by an extra symbol \$.

An *excursion* of M is a sequence of configurations $C_0 \vdash_M C_1 \vdash_M \ldots \vdash_M C_r$ such that the head of the input tape in C_0 and in C_r are on the \$ or on the first b of x and the head is on an a in all other configurations. An excursion is *small* if in C_0 and C_r, the heads are on the same symbol. It is *large* if the heads are on different symbols.

Lemma 1.1 *Let E be an excursion of M on $x = a^n b^n$ and E' be an excursion of M on $x' = a^{n+n!}b^n$. If the first configuration of E and E' have the same corresponding small configuration and the head of the input tape is on the same symbol, then the last configuration of E and E' have the same corresponding small configuration and the head of the input tape is on the same symbol.*

Proof. If E is small, then E' equals E. Thus, the assertion of the theorem is trivial.

Let E (and E') be large. Assume that the head starts on the \$. The other case is treated symmetrically. Since there are $< n$ small configurations, there must be two positions $1 \leq i < j \leq n$ in the first n symbols of x such that the small configurations S are the same when M first visits the cells in position i and j of the input tape. But then for all positions $i + k(j - i)$, M must be in configuration S as long as the symbol in the cell $i + k(j - i)$ is still an a. In particular, M on input x' is in S when it reaches the cell $i + \frac{n!}{j-i}(j - i) = i + n!$. But between position i and n on x and $i + n!$ and $n + n!$, M will also run through the same small configurations. Thus the small configurations at the end of E and E' are the same. ∎

Theorem 1.2 $\texttt{LEN} \notin \mathsf{DSpace}(o(\log n))$.

Proof. Assume that there is an s space bounded 1-tape Turing machine M for LEN with $s = \mathrm{o}(\log)$. Using Lemma 1.1, we can show by induction that whenever the head on the input tape of M is on the \$ or the first b (and was on an a the step before) then on input $x = a^n b^n$ and $x' = a^{n+n!}b^n$, M is in the same small configuration. If the Turing machine ends its last excursion, then it will only compute on the a's or on the b's until it halts. Since on both inputs x and x', M was in the same small configuration, it will be in the same small configurations until it halts. Thus M either accepts both x and x' or rejects both. In any case, we have a contradiction. ∎

1.2 Quadratic time bound for 1-tape Turing machines

Let $\texttt{COPY} = \{w\#w \mid w \in \{a, b\}^*\}$. We will show that COPY can be decided in quadratic time on deterministic 1-tape Turing machines but not in subquadratic time. Again, the first part is rather easy.

Exercise 1.4 *Show that* $\texttt{COPY} \in \mathsf{DTime}_1(n^2)$. *(Bonus: What about deterministic* 2*-tape Turing machines?)*

Let M be a t time bounded deterministic 1-tape Turing machine for COPY. We will assume that M always halts on the end marker \$.

Exercise 1.5 *Show that the last assumption is not a real restriction.*

Definition 1.3 *A* crossing sequence *of M on input x at position i is the sequence of the states of M when moving its head from cell i to $i+1$ or from cell $i+1$ to i. We denote this sequence by* $\mathrm{CS}(x,i)$

If q is a state in an odd position of the crossing sequence, then M is moving its head from the left to the right, if it is in an even position, it moves from the right to the left.

Lemma 1.4 *Let $x = x_1x_2$ and $y = y_1y_2$. If $\mathrm{CS}(x,|x_1|) = \mathrm{CS}(y,|y_1|)$ then $x_1x_2 \in L(M) \iff x_1y_2 \in L(M)$.*

Proof. Since the crossing sequences are the same, M will behave the same on the x_1 part regardless whether there is x_2 or y_2 standing to the right of it. Since M always halts on \$, the claim follows. ∎

Theorem 1.5 $\texttt{COPY} \notin \mathsf{DTime}_1(\mathrm{o}(n^2))$.

Proof. Let M be a deterministic 1-tape Turing machine for COPY. We consider inputs of the form $x = w\#w$ with $w = w_1w_2$ and $|w_1| = |w_2| = n$. For all $v \neq w_2$, $\mathrm{CS}(x,i) \neq \mathrm{CS}(w_1v\#w_1v,i)$ for all $2n+1 \leq i \leq 3n$ by Lemma 1.4, because otherwise, M would accept $w_1w_2\#w_1v$ for some $v \neq w_2$.

We have $\mathrm{Time}_M(x) = \sum_{i\geq 0} |\,\mathrm{CS}(x,i)|$ where $|\,\mathrm{CS}(x,i)|$ is the length of the sequence. Thus

$$\begin{aligned}\sum_{w_2\in\{a,b\}^n} \mathrm{Time}_M(w_1w_2\#w_1w_2) &\geq \sum_{w_2}\sum_{\nu=2n+1}^{3n} |\,\mathrm{CS}(w_1w_2\#w_1w_2,\nu)| \\ &= \sum_{\nu=2n+1}^{3n}\sum_{w_2} |\,\mathrm{CS}(w_1w_2\#w_1w_2,\nu)|.\end{aligned}$$

All the crossing sequences $\mathrm{CS}(w_1w_2\#w_1w_2,\nu)$ have to be pairwise distinct for all $w_2 \in \{a,b\}^n$. Let ℓ be the average length of such a crossing sequence, i.e., $\sum_{w_2} |\,\mathrm{CS}(w_1w_2\#w_1w_2,\nu)| = 2^n \cdot \ell$.

If ℓ is the average length of a crossing sequence, then at least half of the crossing sequences have length $\leq 2\ell$. There are at most $(|Q|+1)^{2\ell}$ crossing

sequences of length $\leq 2\ell$. Let $c = |Q| + 1$. Then $c^{2\ell} \geq 2^n/2$. Thus $\ell \geq c'n$ for some appropriate constant c'. This yields

$$\sum_{w_2} \mathrm{Time}_M(w_1 w_2 \# w_1 w_2) \geq \sum_{\nu=2n+1}^{3n} 2^n \cdot c'n = c' \cdot 2^n \cdot n^2.$$

For at least one w_2,

$$\mathrm{Time}_M(w_1 w_2 \# w_1 w_2) \geq c' \cdot n^2. \qquad \blacksquare$$

Exercise 1.6 *Show for the complement of* COPY, $\overline{\mathtt{COPY}} \in \mathsf{NTime}_1(n \log n)$.

1.3 A gap for deterministic space complexity

Definition 1.6 *An* extended crossing sequence *of M on input x at position i is the sequence of the small configurations of M when moving its head from cell i to $i+1$ or from cell $i+1$ to i on the input tape. We denote this sequence by* $\mathrm{ECS}(x, i)$.

Theorem 1.7 $\mathsf{DSpace}(\mathrm{o}(\log\log n)) = \mathsf{DSpace}(\mathrm{O}(1))$.

Proof. Assume that there is a Turing machine M with space bound $s(n) := \mathrm{Space}_M(n) \in \mathrm{o}(\log\log n) \setminus \mathrm{O}(1)$. We will show by contradiction that such a machine cannot exists. This proves the theorem.

By Exercise 1.2, the number of small configurations on inputs of length n is $\leq |Q| \cdot |\Gamma|^{s(n)} \cdot (s(n) + 2)$. Since s is unbounded, the number of small configurations can be bounded by $c^{s(n)}$ for large enough n, where c is some constant depending on $|Q|$ and $|\Gamma|$.

In an extended crossing sequence, no small configuration may appear twice in the same direction. Otherwise, a (large) configuration of M would appear twice in the computation of M and M would be in an infinite loop. Thus, there are at most

$$(c^{s(n)} + 1)^{2c^{s(n)}} \leq 2^{2^{ds(n)}}$$

different extended crossing sequences on inputs of length n, where d is some constant. For large enough n_0, $s(n) \leq \frac{d^{-1}}{2} \cdot \log\log n$ for all $n \geq n_0$ and therefore $2^{2^{ds(n)}} < n/2$ for all $n \geq n_0$.

Choose s_0 such that $s_0 > \max\{s(n) \mid 0 \leq n \leq n_0\}$ and such that there is an input x with $\mathrm{Space}_M(x) = s_0$. Such an s_0 exists because s is unbounded.

Now let x be a shortest input with $s_0 = \mathrm{Space}_M(x)$. Since the number of extended crossing sequences is $< n/2$ by the definition of s_0 and n_0, there are three pairwise distinct positions $i < j < k$ such that $\mathrm{ECS}(x, i) =$

$\mathrm{ECS}(x, j) = \mathrm{ECS}(x, k)$. But now we can shorten the input by either glueing the crossing sequences at positions i and j or positions j and k. On at least one of the two new inputs, M will use s_0 space, since any small configuration on x appears in at least one of the shortened strings. But this is a contradiction, since x was a shortest string. ■

Exercise 1.7 *Let* $L = \{\mathrm{bin}(0)\#\,\mathrm{bin}(1)\#\ldots\#\,\mathrm{bin}(n) \mid n \in \mathbb{N}\}$.

1. *Show that* L *is not regular.*
2. *Show that* $L \in \mathsf{DSpace}(\log\log n)$.

2 Space and time hierarchies

Hierarchies

Is more space more power? Is more time more power?
The answer is "yes" provided that the space and time bounds behave well, that is, they shall be constructible.[a]

In the case of time "more" means "somewhat more" and not just "more" (see Theorem 2.1).

[a]Non-constructible space and time bounds do not occur in reality. The first one who shows me a book on algorithms that contains a nonconstructible space or time bound gets a big bar of chocolate.

2.1 Universal Turing machines

The space and time hierarchy result will be shown via diagonalization. For this diagonalization, we need to encode and simulate Turing machines, that is, we need a Gödel numbering for Turing machines and a universal Turing machine.

We want to encode Turing machines by words over $\{0,1\}^*$. Let M be a k-tape Turing machine described by $(Q, \Sigma, \Gamma, \delta, q_1, Q_{\text{acc}})$. Let $Q = \{q_1, \ldots, q_{|Q|}\}$, $\Sigma = \{\sigma_1, \ldots, \sigma_{|\Sigma|}\}$, $\Gamma = \{\gamma_1, \ldots, \gamma_{|\Gamma|}\}$, and $Q_{\text{acc}} = \{q_{i_1}, \ldots, q_{i_f}\}$. We encode the fact that

$$\delta(q_i, \gamma_{j_1}, \ldots, \gamma_{j_k}) = (q_{i'}, \gamma_{j'_1}, \ldots, \gamma_{j'_k}, r_{h_1}, \ldots, r_{h_k}),$$

by

$$0^i 10^{j_1} 1 \ldots 10^{j_k} 10^{i'} 10^{j'_1} 1 \ldots 10^{j'_k} 10^{2+r_{h_1}} 1 \ldots 10^{2+r_{h_k}}. \tag{2.1}$$

We use a unary encoding here, because it is simpler to write down. Since the size of the encoding will usually be constant, there is no point in using a more compact but also more complicated encoding. The whole δ is encoded by concatenating all the encodings in (2.1) separated by 11s. We call this string $\text{enc}(\delta)$ Finally, we encode the whole M by

$$\text{enc}(M) = 0^k 110^{|\Gamma|} 110^{|Q|} 110^{i_1} 1 \ldots 10^{i_f} 11\, \text{enc}(\delta) 111^{2(k+1)|\Gamma|}. \tag{2.2}$$

The 1s at the end make the encoding prefix-free. We have chosen that many ones to ensure that the length of the encoding is at least $2(k+1)|\Gamma|$. This

is just a technical assumption that will spare us some case distinctions. If δ is sufficiently large, this assumption is automatically fulfilled.

Note that this encoding is fairly arbitrary. We could choose any other encoding that is "easy to decode" in the sense of the following theorem.

Theorem 2.1 *There is a deterministic* 1*-tape Turing machine* U *such that* U *on input* $\langle e, x\rangle$[1] *computes* $M(x)$[2] *where* e *is an encoding of a deterministic Turing machine* M *as in (2.2). If* M *uses space* s*, then* U *uses space* $\mathrm{O}(|e|\cdot s)$*. For each step of* M*,* U *performs* $\mathrm{O}(|e|^2 \cdot s)$ *steps.*

Proof. Assume that M has k tapes. The Turing machine U stores all of them on one tape. To acchieve this, it uses $2k$ tracks on this tape, i.e., if $\gamma_1, \ldots, \gamma_k$ are the symbols standing on in the ith cell of the k tapes, then this is represented by one symbol $(\gamma_1, \ldots, \gamma_k)$. But we also have to simulate k heads. Therefore, we uses even bigger symbols, namely, symbols from $(\Gamma \times \{*, \square\})^k$. The odd components store the symbols, the even components always contain $\square$ except for one cell, which contains a $*$. This $*$ marks the position of the head on the corresponding tape. U simulates one step of M as follows: We start on the leftmost cell used so far. U completely scans the worktape of M. Whenever it encounters a $*$, it writes the corresponding symbol on a separate tape. If U reaches the end of the worktapes of M, it knows the symbols that M reads. Since it knows the encoding of M, U can now simulate one step of M. U then moves to the left and makes the corresponding changes on the tracks of the tape. When it reaches the lefthand end of the tape, it can simulates the next step of M.

There is one problem to deal with: The size of the work alphabet depends on the number of tapes. Since the number of tapes of M is not known a priori, this is a problem. Even worse, we have to fix the work alphabet of U, but the size of the work alphabet of the simulated machine may vary. Therefore, we represent a symbol in

$$(\gamma_{i_1}, \theta_1, \ldots, \gamma_{i_k}, \theta_k) \in (\Gamma \times \{*, \square\})^k$$

by a string of the form

$$aaf(\gamma_{i_1})ag(\theta_1)a \ldots af(\gamma_{i_k})ag(\theta_k)$$

where

$$f(\gamma_\kappa) = b^\kappa c^{|\Gamma|-\kappa} \qquad \text{and} \qquad g(*) = b,\ \ g(\square) = c$$

[1] $\langle .,. \rangle$ denotes a pairing function. How the pairing function looks like usually does not matter, as long as we can compute it in polynomial time and can recover e and x in polynomial time, too. The easiest way is to take a new symbol, say $\#$, and set $\langle e, x\rangle = e\#x$. With this encoding, $\langle a, \langle b, c\rangle\rangle = \langle\langle a, b\rangle, c\rangle$ which can sometimes be a problem. The encoding $\langle e, x\rangle = 0b_1 0b_2 \ldots 0b_\ell 1ex$, where $b_1 \ldots b_\ell$ is the binary expansion of $|e|$, has the nice property that it does not use any new symbols but its length $|e| + |x| + 2\log|e| + 1$ is only slightly larger the sum of the lengths of e and x.

[2] By abuse of notation, $M(x)$ denotes the output of M on input x.

and a, b, c are new symbols.

First, the Turing machine U brings the input x for M in the form described above. U stores the state of M encoded by a sequence of 0s on a separate track. To simulate a transition of M, U seeks the corresponding entry in the encoding of the transition function of M by comparing each entry in e with the strings between the aas that are marked by a head. U has to compare this symbol by symbol. There are $\leq |e|$ symbols to compare. For each comparison, U might have to move its head over the whole content of the tape, which is $O(|e| \cdot s)$ cells. Thus one transition of M is simulated by $O(|e|^2 \cdot s)$ steps of U. ∎

Such a machine U is called *universal* for the class of deterministic Turing machines.

Remark 2.2 *Also U can be modified such that it also works for nondeterministic Turing machines. U searches all the possible transitions in e, marks them, and then chooses one nondeterministically.*

2.2 Deterministic space hierarchy

The basic technique for our hierarchy theorems will be *diagonalization.*

Theorem 2.3 (Deterministic space hierarchy) *Let $s_2(n) \geq \log n$ be a space constructible function and $s_1(n) = o(s_2(n))$. Then*

$$\mathsf{DSpace}(s_1) \subsetneq \mathsf{DSpace}(s_2).$$

Proof. Let U be the universal Turing machine from Theorem 2.1. We will construct a Turing machine M that is s_2 space bounded such that $L(M) \notin \mathsf{DSpace}(s_1)$. On input y, M works as follows:

Input: $y \in \{0,1\}^*$, interpreted as $y = \langle e, x \rangle$
Output: 0 if the Turing machine encoded by e accepts y, 1 otherwise

1. It first marks $s_2(|y|)$ cells on its tapes.

2. Let $y = \langle e, x \rangle$, where e only contains 0s and 1s. M checks whether e is a valid coding of a deterministic Turing machine E. This can be done in $O(\log |y|)$ space, since M only needs some counters. (M could also just skip the checking and start simulating E. If M detects that e is not a valid encoding it would just stop.)

3. M now simulates E on input y. To do this, M just behaves like U, the only difference is that the input now is y and not x, as in Theorem 2.1.

4. On an extra tape, M counts the steps of U using a ternary counter with $s_2(|y|)$ digits. (Note that we can mark $s_2(|y|)$ cells.)

5. If during this simulation, U leaves the marked space, then M rejects.
6. If E halts, then M halts. If E accepts, then M rejects and vice versa.
7. If E makes more than $3^{s_2(|y|)}$ steps, then M halts and accepts.

Let $L = L(M)$. We claim that $L \notin \mathsf{DSpace}(s_1)$. To see this, assume that N is a s_1 space bounded deterministic Turing machine with $L(N) = L$. It is sufficient to consider a one-tape Turing machine N with extra input tape. Let e be an encoding of N and let $y = e\#x$ for some sufficiently long x.

First assume that $y \in L$. We will show that in this case, N rejects y, a contradiction. If y is in L, then M accepts y. But if M accepts, then either the simulation of N terminated or N makes more than $3^{s_2(|y|)}$ steps. But in the first case, N terminated and rejected by construction and we have a contradiction. In the second case, note that N cannot make more than $c^{s_1(|y|)} \cdot (s_1(|y|) + 2) \cdot (|y| + 2)$ steps without entering an infinite loop. Thus if $3^{s_2(|y|)} > c^{s_1(|y|)} \cdot (s_1(|y|) + 2) \cdot (|y| + 2)$ then we get a contradiction, too. But $3^{s_2(|y|)} > c^{s_1(|y|)} \cdot (s_1(|y|) + 2) \cdot (|y| + 2)$ is equivalent to $\log 3 \cdot s_2(|y|) > \log c \cdot s_1(|y|) + \log(s_1(|y|)+2) + \log(|y|+2)$. This is fulfilled by assumption for all long enough y, i.e., for long enough $|x|$. Thus we obtained a contradiction again.

The possibility $y \notin L$ remains. We will show that now N accepts y, a contradiction. If M rejects y, then M ran out of space or N terminated. The second case is again easy. If the simulation of N terminated, then N accepted because $y \notin L$, a contradiction. We will next show that the first case cannot happen. Since N is s_1 space bounded, the simulation via U needs space $|e| \cdot s_1(|y|)$. But $|e| \cdot s_1(|y|) \leq s_2(|y|)$ for sufficiently large $|y|$. Thus this case cannot happen.

M is by construction s_2 space bounded. This proves the theorem. ∎

Exercise 2.1 *Describe a log space bounded Turing machine that checks whether the input is a correct encoding of a deterministic Turing machine.*

2.3 Deterministic time hierarchy

Next, we do the same for time complexity classes. The result will not be as nice as for space complexity, since the universal machine U is slower than the machine that U simulates.

Theorem 2.4 (Deterministic time hierarchy) *Let t_2 be a constructible function and $t_1^2 = \mathrm{o}(t_2)$. Then*

$$\mathsf{DTime}(t_1) \subsetneq \mathsf{DTime}(t_2).$$

Proof. Let U be the universal Turing machine from Theorem 2.1. We will construct a Turing machine M that is $O(t_2)$ time bounded such that $L(M) \notin \mathsf{DTime}(t_1)$. On input y, M works as follows:

Input: $y \in \{0,1\}^*$, interpreted as $y = \langle e, x \rangle$
Output: 0 if the Turing machine encoded by e accepts y, 1 otherwise

1. Let $y = e\#x$, where e only contains 0s and 1s. M checks whether e is a valid coding of a deterministic Turing machine E.

2. M now simulates E on input y. To this this, M just behaves like U, the only difference is that the input now is y and not x, as in Theorem 2.1.

3. M constructs $t_2(|y|)$ on an extra tape.

4. On an extra tape, M counts the steps of U using a binary counter.

5. If during this simulation, U makes more than $t_2(|y|)$ steps, then M halts and accepts.

6. If E halts, then M halts. If E accepts, then M rejects and vice versa.

Let $L = L(M)$. We claim that $L \notin \mathsf{DTime}(t_1)$. To see this, assume that N is a t_1 time bounded deterministic Turing machine with $L(N) = L$. Let e be an encoding of N and let $y = e\#x$ for some sufficiently long x.

First assume that $y \in L$. We will show that in this case, N rejects y, a contradiction. If y is in L, then M accepts y. But if M accepts, then either N makes more than $t_2(|y|)$ steps or N halts. In the second case, M accepted. But then N rejected. A contradiction. We next show that the first case cannot happen. The simulation of N needs $c \cdot |e|^2 \cdot t_1^2(|y|)$ many steps for some constant c. But $c \cdot |e|^2 \cdot t_1^2(|y|) \leq t_2(|y|)$ for sufficiently long y by assumption. Thus this case cannot happen.

Next assume that $y \notin L$. Then N terminates. But since M rejects, N accepted. A contradiction.

By construction, the Turing machine M is $O(t_2)$ time bounded. Using linear speed-up, we can get this down to t_2 time bounded, if $t_2 = \omega(n)$. If $t_2 = O(n)$, then the theorem is trivial. ∎

2.4 Remarks

The assumption $t_1^2 = o(t_2)$ in the time hierarchy theorem is needed, since the universal Turing machine U incurs an extra factor of t_1 in the running time when simulating.

Hennie and Stearns showed the following theorem.

Theorem 2.5 (Hennie & Stearns) *Every t time and s space bounded k-tape deterministic Turing machine can be simulated by an $\mathrm{O}(t \log t)$ time bounded and $\mathrm{O}(s)$ space bounded 2-tape Turing machine.*

We do not give a proof here. Using this theorem, we proceed as follows. On input $e\#x$, M only simulates if e is a valid encoding of a 2-tape Turing machine. In the proof, we will now take N to be a 2-tape Turing machine. In this way, we can replace the assumption $t_1^2 = \mathrm{o}(t_2)$ by $t_1 \log t_1 = \mathrm{o}(t_2)$.

Research Problem 2.1 *Can the assumption $t_1 \log t_1 = \mathrm{o}(t_2)$ be further weakened?*

If the number of tapes is fixed, then one can obtain a tight time hierarchy. Again we do not give a proof here.

Theorem 2.6 (Fürer) *Let $k \geq 2$, t_2 time constructible, and $t_1 = \mathrm{o}(t_2)$. Then*

$$\mathsf{DTime}_k(t_1) \subsetneq \mathsf{DTime}_k(t_2).$$

We conclude with pointing out that the assumption that s_2 and t_2 are constructible are really necessary.

Theorem 2.7 (Borodin's gap theorem) *Let g be a recursive function $\mathbb{N} \to \mathbb{N}$ with $g(n) \geq n$ for all n. Then there are functions $s, t : \mathbb{N} \to \mathbb{N}$ with $s(n) \geq n$ and $t(n) \geq n$ for all n with*

$$\mathsf{DTime}(g(t(n))) = \mathsf{DTime}(t(n)),$$
$$\mathsf{DSpace}(g(s(n))) = \mathsf{DSpace}(s(n)).$$

Set for instance $g(n) = 2^n$ (or 2^{2^n} or ...) and think for a minute how unnatural non-constructible time or space bounds are.

2.5 Translation

Assume we showed that $\mathsf{DTime}(t_1) \subseteq \mathsf{DTime}(t_2)$. In this section, we show how to get other inclusions out of this for free via a technique called *padding*.

Theorem 2.8 *Let t_1, t_2, and f be time constructible such that $t_1(n) \geq (1+\epsilon)n$, $t_2(n) \geq (1+\epsilon)n$, and $f(n) \geq n$ for all n for some $\epsilon > 0$. If*

$$\mathsf{DTime}(t_1(n)) \subseteq \mathsf{DTime}(t_2(n)),$$

then

$$\mathsf{DTime}(t_1(f(n))) \subseteq \mathsf{DTime}(t_2(f(n))).$$

Proof. Let $L_1 \in \mathsf{DTime}(t_1(f(n))$ and let M_1 be a $t_1(f(n))$ time bounded deterministic Turing machine for L_1. Let % be a symbol that is not in Σ.

Let

$$L_2 = \{x\%^r \mid M_1 \text{ accepts } x \text{ in } t_1(|x| + r) \text{ steps}\}.$$

Since t_1 is time constructible, there is an $\mathrm{O}(t_1)$ time bounded deterministic Turing machine M_2 that accepts L_2. Using acceleration, we obtain $L_2 \in \mathsf{DTime}(t_1)$. By assumption, there is a t_2 time bounded Turing machine M_3 with $L(M_3) = L_2$.

Finally, we construct a deterministic Turing machine M_4 for L_1 as follows: M_4 on input x computes $f(|x|)$ and appends $f(|x|) - |x|$ symbols % to the input. Thereafter, M_4 simulates M_3 for $t_2(f(|x|))$ steps. M_4 is $\mathrm{O}(f(n) + t_2(f(n))$ time bounded. Using linear speedup, we get $L_1 \in \mathsf{DTime}(t_2(f(n)))$. We have

$$\begin{aligned} x \in L(M_4) &\iff M_3 \text{ accepts } x\%^{f(|x|)-|x|} \text{ in } t_2(f(|x|)) \text{ steps} \\ &\iff x\%^{f(|x|)-|x|} \in L_2 \\ &\iff M_1 \text{ accepts } x \text{ in } t_1(f(|x|)) \text{ steps} \\ &\iff x \in L_1. \qquad \blacksquare \end{aligned}$$

Exercise 2.2 *Show the following: Let s_1, s_2, and f be space constructible such that $s_1(n) \geq \log n$, $s_2(n) \geq \log n$, and $f(n) \geq n$ for all n. If*

$$\mathsf{DSpace}(s_1(n)) \subseteq \mathsf{DSpace}(s_2(n)),$$

then

$$\mathsf{DSpace}(s_1(f(n))) \subseteq \mathsf{DSpace}(s_2(f(n))).$$

(Hint: Mimic the proof above. If the space bounds are sublinear, then we cannot explicitly pad with %s. We do this virtually using a counter counting the added %s.)

Remark 2.9 *While it is a nice exercise, the above result is not very meaningful, since s_1 cannot grow asymptotically faster than s_2, since we have a tight space hierarchy result. But—and this is the interesting part—the proofs work word by word for nondeterministic Turing machines, too. One can even "mix" determinism and nondeterminism as well as time and space as long as the complexity measures on the left-hand side are the same and on the right-hand side are the same.*

3 Robust complexity classes

Complexity classes

Good complexity classes should have two properties:

1. They should characterizes important problems.
2. They should be robust under reasonable changes of the computation model.

One such example is NP: There are an abundance of important NP-complete problems and it is also robust: If we defined nondeterministic WHILE programs or RAM machines, we would get the same class of problems.

Definition 3.1

$$\begin{aligned}
\mathsf{L} &= \mathsf{DSpace}(\mathrm{O}(\log n)) \\
\mathsf{NL} &= \mathsf{NSpace}(\mathrm{O}(\log n)) \\
\mathsf{P} &= \bigcup_{i\in\mathbb{N}} \mathsf{DTime}(\mathrm{O}(n^i)) \\
\mathsf{NP} &= \bigcup_{i\in\mathbb{N}} \mathsf{NTime}(\mathrm{O}(n^i)) \\
\mathsf{PSPACE} &= \bigcup_{i\in\mathbb{N}} \mathsf{DSpace}(\mathrm{O}(n^i)) \\
\mathsf{EXP} &= \bigcup_{i\in\mathbb{N}} \mathsf{DTime}(2^{\mathrm{O}(n^i)}) \\
\mathsf{NEXP} &= \bigcup_{i\in\mathbb{N}} \mathsf{NTime}(2^{\mathrm{O}(n^i)}).
\end{aligned}$$

L and NL are classes of problems that can be decided with very few space. Note that by Savitch's theorem, $\mathsf{NL} \subseteq \mathsf{DSpace}(\log^2 n)$. We will also see in later lectures, that problems in L and also NL have efficient parallel algorithms. P is *the* class of problems that are considered to be feasible or tractable. NP characterizes many important optimization problems. PSPACE are the problems that can be decided with feasible space

requirements. Note that by Savitch's theorem, there is no point in defining nondeterministic polynomial space. EXP is the smallest deterministic class known to contain NP. And NEXP, well, NEXP is just NEXP.

We have

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP},$$

which follows from Theorems 25.9 and 25.10 of the "Theoretical Computer Science" lecture. By the time and space hierarchy theorems, we know that $\mathsf{L} \subsetneq \mathsf{PSPACE}$ and $\mathsf{P} \subsetneq \mathsf{EXP}$. Thus, some of the inclusions above must be strict. We conjecture that all of them are strict. By the translation technique, $\mathsf{L} = \mathsf{P}$ would imply $\mathsf{PSPACE} = \mathsf{EXP}$, etc.

3.1 Reduction and completeness

Let R be some set of functions $\Sigma^* \to \Sigma^*$. A language L' is called R *many one reducible* to another language L if there is some function $f \in R$ (the *reduction*) such that for all $x \in \Sigma^*$,

$$x \in L' \iff f(x) \in L.$$

Thus we can decide membership in L' by deciding membership in L. Let C be some complexity class. A language L is called C-*hard* with respect to R many one reductions if for all $L' \in \mathsf{C}$, L' is R many one reducible to L. L is called C-*complete* if in addition, $L \in \mathsf{C}$.

To be useful, reductions should fulfill two properties:

- The reduction should be weaker than the presumably harder one of the two complexity classes we are comparing. Particularly, this means that if L' is R many one reducible to L and $L \in \mathsf{C}$, then L' should also be in C, that is, C is closed under R many one reductions.
- The reduction should be transitive. In this case, if L' is R many one reducible to L and L' is C-hard, then L is also C-hard.

The most popular kind of many one reductions are polynomial time many one reductions, which are used to define NP-hardness. When Steve Cook showed that Satisfiability is NP-complete, he did not use many-one polynomial time reductions (and strictly speaking did not prove that Satisfiability is NP-complete). The concept of many-one polynomial time reducibility was introduced by Richard Karp. Steve Cook used a stronger (maybe one would like to call it weaker) kind of reduction (but only formally; all his reduction where "essentially" many-one), called *Turing polynomial time reductions*. To define this kind of reductions, we need the notion of *oracle Turing machines*.

An *oracle Turing machine* M is a multitape Turing machine that in addition to its regular work tape has a distinguished oracle tape which is read/write. Furthermore, it has two distinguished states, a query state $q_?$ and an answer state $q_!$. Let $f : \Sigma^* \to \Sigma^*$ be some total function. M with oracle f, denoted by M^f, works as follows: As long as M^f is not in $q_?$ it works like a normal Turing machine. As soon as M^f enters $q_?$, the content of the oracle tape is replaced by $f(y)$, where y is the previous content of the oracle tape, and the head of the oracle tape is put on the first symbol of $f(y)$. Then M^f enters $q_!$. The content of the other tapes and the other head positions are not changed. Such an oracle query is counted as only one time step. In other words, M^f may evaluate the function f at unit cost. All the other notions, like acceptance or time complexity, are defined as before. We can also have a language L as an oracle. In this case, we take f to be the characteristic function χ_L of L. For brevity, we will write M^L instead of M^{χ_L}.

Let again $L, L' \subseteq \Sigma^*$. Now L' is *Turing reducible* to L if there is a deterministic oracle Turing machine R such that $L' = L(R^L)$. Basically this means that we can solve L' deterministically if we have oracle access to L. Many-one reductions can be viewed as a special type of Turing reductions where we can only query once at the end of the computation and have to return the same answer as the oracle. To be meaningful, the machine R should be time bounded or space bounded. If, for instance, R is polynomial time bounded, then we speak of polynomial time Turing reducibility.

Popular reductions

- polynomial time many one reductions (denoted by $L' \leq_{\mathrm{P}} L$),
- polynomial time Turing reductions ($L' \leq_{\mathrm{P}}^{\mathrm{T}} L$),
- logspace many one reductions ($L' \leq_{\log} L$).

3.2 Co-classes

For a complexity class C, co-C denotes the set of all languages $L \subseteq \Sigma^*$ such that $\bar{L} \in \mathsf{C}$. Deterministic complexity classes are usually closed under complementation. For nondeterministic time complexity classes, it is a big open problem whether a class equals its co-class, in particular, whether $\mathsf{NP} = \text{co-}\mathsf{NP}$. For nondeterministic space complexity classes, this problem is solved.

Theorem 3.2 (Immerman, Szelepcsényi) *For every space constructible*

$s(n) \geq \log n$,

$$\mathsf{NSpace}(s) = \mathsf{co\text{-}NSpace}(s).$$

We postpone the prove of this theorem to the next chapter.

Exercise 3.1 *Show that satisfiability is* co-NP*-hard under polynomial time Turing reductions. What about polynomial time many one reductions?*

Exercise 3.2 *Show that if L is* C*-hard under R many-one reductions, then $\bar{L}$ is* co-C*-hard under R many-one reductions.*

4 L and NL

4.1 Logarithmic space reductions

Logarithmic space many one reductions are the appropriate tool to investigate the relation between L and NL (and also between NL and P).

Exercise 4.1 *Let f and g be logarithmic space computable functions $\Sigma^* \to \Sigma^*$. Then $f \circ g$ is also logarithmic space computable.*

Corollary 4.1 *$\leq_{\log}$ is a transitive relation, i.e., $L \leq_{\log} L'$ and $L' \leq_{\log} L''$ implies $L \leq_{\log} L''$.*

Proof. Assume that $L \leq_{\log} L'$ and $L' \leq_{\log} L''$. That means that there are logarithmic space computable functions f and g such for all x,

$$x \in L \iff f(x) \in L'$$

and for all y,

$$y \in L' \iff g(y) \in L''.$$

Let $h = g \circ f$. h is logarithmic space computable by Exercise 4.1. We have for all x,

$$x \in L \iff f(x) \in L' \iff g(f(x)) \in L''.$$

Thus h is a many one reduction from L to L''. ∎

Lemma 4.2 *Let $L \leq_{\log} L'$.*

1. *If $L' \in$ L, then $L \in$ L,*
2. *If $L' \in$ NL, then $L \in$ NL,*
3. *If $L' \in$ P, then $L \in$ P.*

Proof. Let f be a logarithmic space many one time reduction from L to L'. Let χ_L and $\chi_{L'}$ be the characteristic functions of L and L'. We have $\chi_L = \chi_{L'} \circ f$.

1. It is clear that a language is in L if and only if its characteristic function is logarithmic space computable. By Exercise 4.1, $\chi_{L'} \circ f$ is logarithmic space computable. Thus, $L \in$ L.

2. This follows from a close inspection of the proof of Exercise 4.1. The proof also works if the Turing machine for the "outer" Turing machine is nondeterministic. (The outer Turing machine is the one that gets the input $f(x)$.)

3. Since f is logarithmic space computable, it is also polynomial time computable, since a logarithmically space bounded deterministic Turing machine can make at most polynomially many steps. Thus $\chi_{L'} \circ f$ is polynomial time computable, if $\chi_{L'}$ is polynomial time computable. ■

Corollary 4.3 *1. If L is* NL*-hard under logarithmic space many one reductions and $L \in$ L, then* L = NL*.*

2. If L is P*-hard under logarithmic space many one reductions and $L \in$ NL, then* NL = P*.*

Proof. We just show the first statement, the second one follows in a similar fashion: Since L is NL-hard, $L' \leq_{\log} L$ for all $L' \in$ NL. But since $L \in$ L, $L' \in$ L, too, by Lemma 4.2. Since L' was arbitrary, then claim follows. ■

4.2 s-t connectivity

NL is a *syntactic* class. A class C is called syntactic if we can check for a given Turing machine M whether the resources that M uses are bounded as described by C. (This is only an informal concept, not a definition!) While we cannot check whether a Turing machine M is $O(\log n)$ space bounded, we can make the Turing machine $O(\log n)$ space bounded by first marking the appropriate number of cells and then simulate M. Whenever M leaves the marked space, we reject.

Exercise 4.2 *Prove that it is not decidable to check whether a Turing machine is* $\log n$ *space bounded.*

Syntactic classes usually have generic complete problems. For NL, one such problem is

$$\texttt{Gen-NL} = \{e\#x\#1^s \mid e \text{ is an encoding of a nondeterministic Turing machine that accepts } x \text{ in } (\log s)/|e| \text{ space.}\}$$

Exercise 4.3 *Prove that* Gen-NL *is* NL*-complete.*

But there are also natural complete problems for NL. One of the most important ones is CONN, the question whether there is a path from a given node s to another given node t in a directed graph. CONN plays the role for NL that Satisfiability plays for NP.

Definition 4.4 CONN *is the following problem: Given (an encoding of) a directed graph G and two nodes s and t, decide whether there is a path from s to t in G.*

Theorem 4.5 CONN *is* NL*-complete.*

Proof. We have to show two things: CONN $\in$ NL and CONN is NL-hard.

For the first statement, we construct an $O(\log n)$ space bounded Turing machine M for CONN. We may assume that the nodes of the graph are numbered from $1, \ldots, n$. Writing down one node needs space $\log n$. M first writes s on the work tape and initializes a counter with zero. During the whole simulation, the work tape of M will contain one node and the counter. If v the the node currently stored, then M nondeterministically chooses an edge (v, u) of G and replaces v by u and increases the counter by one. If u happens to be t, then M stops and accepts. If the counter reaches the value n, then M rejects.

It is easy to see that if there is a path from s to t, then there is an accepting computation path of C, because if there is a path, then there is one with at most $n - 1$ edges. If there is no path from s to t, then C will never accept. (We actually do not need the counter, it is just used to cut off infinite (rejecting) paths.) C only uses $O(\log n)$ space.

For the second statement, let $L \in$ NL. Let M be some $\log n$ space bounded nondeterministic Turing machine for L. We may assume w.l.o.g. that for all inputs of length n, there is one unique accepting configuration.[1] M accepts an input x, if we can reach this accepting configuration from $SC(x)$ in the configuration graph. We only have to consider $c^{\log n} = \text{poly}(n)$ many configurations.

It remains to show how to compute the reduction in logarithmic space, that is, how to generate the configuration graph in logarithmic space. To do this, we enumerate all configurations that use $s(|x|)$ space where x is the given input. For each such configuration C we construct all possible successor configurations C' and write the edge (C, C') on the output tape. To do so, we only have to store two configurations at a time. Finally, we have to append the starting configurations $\mathsf{SC}(x)$ as s and the unique accepting configuration as t to the output. ∎

[1] We can assume that the worktape of M is onesided infinite. Whenever M would like to stop, then it erases all the symbols it has written and then moves its head to the \$ that marks the beginning of the tape, moves the head of the input tape one the first symbol, and finally halts.

4.3 Proof of the Immerman–Szelepcsényi theorem

Our goal is to show that the complement of CONN is in NL. Since CONN is NL-complete, $\overline{\texttt{CONN}}$ is co-NL-complete. Thus NL = co-NL. The Immerman-Szelepcsényi theorem follows by translation.

Let $G = (V, E)$ be a directed graph and $s, t \in V$. We want to check whether there is *no* path from s to t. For each d, let

$$N_d = \{x \in V \mid \text{there is a path of length} \leq d \text{ from } s \text{ to } x.\}$$

be the neighbourhood of s of radius d. Let

$$\texttt{DIST} = \{\langle G, s, x, d\rangle \mid \text{there is a path of length} \leq d \text{ from } s \text{ to } x\}$$

DIST is the language of all tuples $\langle G, s, x, d\rangle$ such that x has distance at most d from the source node s. Next we define a partial complement of DIST. A tuple $\langle G, s, x, d, |N_d|\rangle$ is in NEGDIST if there is *no* path from s to x of length $\leq d$. If there is a path from s to x of length $\leq d$, then $\langle G, s, x, d, |N_d|\rangle \notin$ NEGDIST. For all $\langle G, s, x, d, S\rangle$ with $S \neq |N_d|$, we do not care whether it is in NEGDIST or not.[2]

Lemma 4.6 DIST $\in$ NL.

Proof. The proof is the same as showing that CONN $\in$ NL. The only difference is that we count to d and not to n. ∎

Lemma 4.7 NEGDIST $\in$ NL.

Proof. The following Turing machine accepts NEGDIST:

Input: $\langle G, s, x, d, S\rangle$

1. Guess S pairwise distinct nodes $v \neq x$, one after another.
2. For each v: Check whether v is at a distance of at most d from s by guessing a path of length $\leq d$ from s to v.
3. Whenever one of these tests fails, reject.
4. If all of these tests are passed, then accept

If $S = |N_d|$ and there is no path of length d from s to x, then M has an accepting path, namely the path where M guesses all S nodes v in N_d correctly and guesses the right paths that prove $v \in N_d$. If $S = |N_d|$ and

[2] Strictly speaking, NEGDIST is not one language but a family of languages. When we say that NEGDIST $\in$ NL, we mean that for one choice of the do-not-care triples, the language is in NL.

there is a path of length $\leq d$ from s to x, then M can never accept, since there are not $|N_d|$ many nodes different from x with distance $\leq d$ from s.

M is surely $\log n$ space bounded, since it only has to store a constant number of nodes and counters. ∎

If we knew $|N_d|$, then we would be able to decide $\overline{\texttt{CONN}}$ with nondeterministic logarithmic space.

Definition 4.8 *A nondeterministic Turing machine M computes a function $f : \{0,1\}^* \to \{0,1\}^*$ if for every input $x \in \{0,1\}^*$,*

1. *M halts with $f(x)$ on the work tape on every accepting computation path and*
2. *there is at least one accepting computation path on x.*

Note that if $L = L(M)$ for some nondeterministic Turing machine M, then it is not clear whether χ_L is computable (in the sense of definition above) by a nondeterministic Turing machine with the same resources—in contrast to deterministic Turing machines.

Lemma 4.9 *There is a $\log n$ space bounded nondeterministic Turing machine that computes the mapping $\langle G, s, d\rangle \to |N_d|$.*

Proof. We construct a Turing machine M that starts by computing $|N_0|$ and then, given $|N_i|$, it computes $|N_{i+1}|$. Once it has computed $|N_{i+1}|$, it can forget about $|N_i|$.

Input: $\langle G, s, d\rangle$
Output: $|N_d|$

1. Set $|N_0| = 1$.
2. For $i = 0$ to $d - 1$ do
3. $\quad c := 0$
4. $\quad$ For each node $v \in V$ nondeterministically guess whether $v \in N_{i+1}$
5. $\quad\quad$ If $v \in N_{i+1}$ was guessed, test whether $\langle G, s, v, i + 1\rangle \in \texttt{DIST}$.
6. $\quad\quad\quad$ If the test fails, reject, else set $c = c + 1$.
7. $\quad\quad$ If $v \notin N_{i+1}$ was guessed, do the following:
8. $\quad\quad\quad$ For all $u \in V$, test whether $u \neq v$ and (u, v) is not an edge.
9. $\quad\quad\quad\quad$ If not, test whether $\langle G, s, u, i, |N_i|\rangle \in \texttt{NEGDIST}$
10. $\quad\quad\quad\quad\quad$ If not, reject.

11. $|N_{i+1}| := c$

12. return $|N_d|$

We prove by induction on j that $|N_j|$ is computed correctly.
Induction base: $|N_0|$ is certainly computed correctly.
Induction step: Assume that $|N_j|$ is computed correctly. This means that M on every computation path on which the for loop of line 2 was executed for the value $j-1$ computed the true value $|N_j|$ in line 11. Consider the path on which for each v, M correctly guesses whether $v \in N_{j+1}$. If $v \in N_{j+1}$, then there is a computation path on which M passes the test $\langle G, s, v, j+1\rangle \in \texttt{DIST}$ in line 5 and increases c in line 6. (Note that this test is again performed nondeterministically.) If $v \notin N_{j+1}$, then for all u such that either $u = v$ or (u, v) is an edge of G, we have that $u \notin N_j$. Hence on some computation path, M will pass all the tests in line 9, since by the induction hypothesis, M computed $|N_j|$ correctly.

On a path on which M made a wrong guess about $v \in N_{j+1}$, M cannot pass the corresponding test and M will reject.

Thus on all paths, on which M does not reject, c has the same value in the end, this value is $|N_{j+1}|$, and there is a least one such path. This proves the claim about the correctness.

M is logarithmically space bounded, since it only has to store a constant number of nodes and the values $|N_j|$ and c. Testing membership in DIST and NEGDIST can also be done in logarithmic space. ∎

Theorem 4.10 $\overline{\texttt{CONN}} \in \mathsf{NL}$.

Proof. $\langle G, s, t\rangle \in \overline{\texttt{CONN}}$ is equivalent to $\langle G, s, t, n, |N_n|\rangle \in \texttt{NEGDIST}$, where n is the number of nodes of G. ∎

Exercise 4.4 *Finish the proof of the Immerman–Szelepcsényi theorem. (Hint: Translation)*

4.4 Undirected s-t connectivity

Now that we have found a class that characterizes directed s-t connectivity, it is natural to ask whether we can find a class that describes undirected connectivity. UCONN is the following problem: Given an *undirected* graph G and two nodes s and t, is there a path connecting s and t? To get a complexity class that describes UCONN, we can define

$$\mathsf{SL} = \{L \mid L \leq_{\log} \texttt{UCONN}\}.$$

This makes UCONN automatically to an SL-complete problem. Now we have to look for some other interpretation of SL. A Turing machine M is called symmetric if for all configurations C and C',

$$C \vdash_M C' \quad \Rightarrow \quad C' \vdash_M C,$$

that is, the configuration graph is symmetric. It is fairly easy to see that

$$\mathsf{SL} = \{L \mid L = L(M) \text{ for some symmetric logarithmic space bounded Turing machine } M\}$$

One can argue whether symmetric Turing machines are a natural concept or not. However, right now, they are obsolete, since Omer Reingold proved the remarkable result below, a proof of which we might see later.

Theorem 4.11 (Reingold) $\mathsf{L} = \mathsf{SL}$.

Corollary 4.12 $\mathtt{UCONN} \in \mathsf{L}$.

To appreciate the result above, note that in space $O(\log n)$, we can barely store a constant number of nodes. Now take your favourite algorithm for undirected connectivity and try to implement it with just logarithmic space.

Excursus: SL-complete problems

While SL looks like a little esoteric complexity class (well, at least as long as you do not know that it equals L), a lot of natural problems where shown to be SL-complete, but before Reingold's result, nobody could prove that they were in L. Here are some examples:

Planarity testing: Is a given graph planar?

Bipartiteness testing: Is a given graph bipartite?

k-disjoint paths testing: Has a given graph k node-disjoint paths from s to t? (This generalizes UCONN.)

There is even a compendium of SL-complete problems (which are now L-complete problems):

Carme Àlvarez and Raymond Greenlaw. A compendium of problems complete for symmetric logarithmic space. *Computational Complexity*, 9:73–95, 2000.

Excursus: Vladimir Trifonov

Vladimir Trifonov is (was?) a poor PhD student at University of Texas who showed that $\mathtt{UCONN} \in \mathsf{DSpace}(\log n \log\log n)$[3] at the same time when Omer Reingold showed $\mathtt{UCONN} \in \mathsf{L}$. This way, a remarkable result became a footnote (or an excursus).

---

[3] Savitch's Theorem gives $\mathtt{UCONN} \in \mathsf{DSpace}(\log^2 n)$ and the best result at that time was $\mathtt{UCONN} \in \mathsf{DSpace}(\log^{4/3} n)$ which was achieved by derandomizing a random walk on the graph. We will come to this later ...

5 Boolean circuits

In this chapter we study another model of computation, Boolean circuits. This model is useful in at least three ways:

- Boolean circuits are a natural model for parallel computation.
- Boolean circuits serve as a nonuniform model of computation. (We will explain what this means later on.)
- Evaluating a Boolean circuit is a natural P-complete problem.

5.1 Boolean functions and circuits

We interpret the value 0 as Boolean false and 1 as Boolean true. A function $\{0,1\}^n \to \{0,1\}^m$ is called a Boolean function. n is its arity, also called the input size, and m is its output size.

A *Boolean circuit* C with n inputs and m outputs is an acyclic digraph with $\geq n$ nodes of indegree zero and m nodes of outdegree zero. Each node has either indegree zero, one or two. If its indegree is zero, then it is labeled with $x_1, \ldots, x_n$ or 0 or 1. Such a node is called an input node. If a node has indegree one, then it is labeled with $\neg$. Such a node computes the Boolean Negation. If a node has indegree two, it is labeled with $\vee$ or $\wedge$ and the node computes the Boolean Or or Boolean And, respectively. The nodes with outdegree zero are ordered. The *depth* of a node v of C is the length of a longest path from a node of indegree zero to v. (The length of a path is the number of edges in it.) The depth of v is denoted by $\mathrm{depth}(v)$. The depth of C is defined as $\mathrm{depth}(C) = \max\{\mathrm{depth}(v) \mid v \text{ is a node of } C\}$. The size of C is the number of nodes in it and is denoted by $\mathrm{size}(C)$.

Such a Boolean circuit C computes a Boolean function $\{0,1\}^n \to \{0,1\}^m$ as follows. Let $\xi \in \{0,1\}^n$ be a given input. With each node, we associate a value $\mathrm{val}(v,\xi) \in \{0,1\}$ computed at it. If v is an input node, then $\mathrm{val}(v,\xi) = \xi_i$, if v is labeled with x_i. If v is labeled with 0 or 1, then $\mathrm{val}(v,\xi)$ is 0 or 1, respectively. This defines the values for all nodes of depth 0. Assume that the value of all nodes of depth d are known. Then we compute $\mathrm{val}(v,\xi)$ of a node v of depth $d+1$ as follows: If v is labeled with $\neg$ and u is the predecessor of v, then $\mathrm{val}(v,\xi) = \neg\,\mathrm{val}(u,\xi)$. If v is labeled with $\vee$ or $\wedge$ and u_1, u_2 are the predecessors of v, then $\mathrm{val}(v,\xi) = \mathrm{val}(u_1,\xi) \vee \mathrm{val}(u_2,\xi)$ or $\mathrm{val}(v,\xi) = \mathrm{val}(u_1,\xi) \wedge \mathrm{val}(u_2,\xi)$. For each node v, this defines a function $\{0,1\}^n \to \{0,1\}$ computed at v by $\xi \mapsto \mathrm{val}(v,\xi)$. Let $g_1, \ldots, g_m$ be the

functions computed at the output nodes (in this order). Then C computes a function $\{0,1\}^n \to \{0,1\}^m$ defined by $\xi \mapsto g_1(\xi)g_2(\xi)\dots g_m(\xi)$. We denote this function by $C(\xi)$.

The labels are taken from $\{\neg, \vee, \wedge\}$. This set is also called *standard basis*. This standard is known to be complete, that is, for any Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, there is Boolean circuit (over the standard basis) that computes it. For instance, the CNF of a function directly defines a circuit for it. (Note that we can simulate one Boolean And or Or of arity n by $n-1$ Boolean And or Or of arity 2.)

Finally, a circuit is called a *Boolean formula* if all nodes have outdegree ≤ 1.

Exercise 5.1 *Show that for any Boolean circuit of depth d, there is an equivalent Boolean formula of depth $\mathrm{O}(d)$ and size $2^{\mathrm{O}(d)}$.*

Exercise 5.2 *Prove that for any Boolean circuit C of size s, there is an equivalent one C' of size $\leq 2s + n$ such that all negations have depth 1 in C'. (Hint: De Morgan's law. It is easier to prove the statement first for formulas.)*

Boolean circuits can be viewed as a model of parallel computation, since a node can compute its value as soon as it knows the value of its predecessor. Thus, the depth of a circuits can be seen as the time taken by the circuit to compute the result. Its size measures the "hardware" needed to built the circuit.

Exercise 5.3 *Every Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a Boolean circuit of size $2^{\mathrm{O}(n)}$.*[1]

5.2 Uniform families of circuits

There is a fundamental difference between circuits and Turing machines. Turing machines compute functions with variable input length, e.g., $\Sigma^* \to \Sigma^*$. Boolean circuits only compute a function of fixed size $\{0,1\}^n \to \{0,1\}^m$. Since the input alphabet Σ is fixed, we can encode the symbols of Σ by a fixed length binary code. In this way, we overcome the problem that Turing machines and Boolean circuits compute on different symbols. To overcome the problem that circuits compute functions of fixed length, we will introduce families of circuits.

In the following, we will only look at Boolean circuits with one output node, i.e., circuits that decide languages. Most of the concepts and results

[1] This can be sharpened to $(1+\epsilon)\cdot 2^n/n$ for any $\epsilon > 0$. The latter bound is tight: For any $\epsilon > 0$ and any large enough n, there is a Boolean function $f : \{0,1\}^n \to \{0,1\}$ such that every circuit computing f has size $(1-\epsilon)2^n/n$. This is called the *Shannon–Lupanow bound*.

presented in the remainder of this chapter also work for circuits with more output nodes, that is, circuits that compute functions.

Definition 5.1 *1. A sequence $C = C_1, C_2, C_3, \ldots$ of Boolean circuits such that C_i has i inputs is called a family of Boolean circuits.*

2. C is s size bounded and d depth bounded if $\mathrm{size}(C_i) \leq s(i)$ and $\mathrm{depth}(C_i) \leq d(i)$ for all i.

3. C computes the function $\{0,1\}^* \to \{0,1\}$ given by $x \mapsto C_{|x|}(x)$. Since we can interpret this as a characteristic function, we also say that C decides a language.

Families of Boolean circuits can decide nonrecursive languages, in fact any $L \subseteq \{0,1\}^*$ is decided by a family of Boolean circuits. To exclude such phenomena, we put some restriction on the families.

Definition 5.2 *1. A family of circuits is called s space and t time constructible, if there is an s space bounded and t time bounded deterministic Turing machine that given input 1^n writes down an encoding of C_n that is topologically sorted.*

2. A s size and d depth bounded family of circuits C is called logarithmic space uniform *if it is $\mathrm{O}(\log s(n))$ space constructible. It is called* polynomial time uniform*, if it is $\mathrm{poly}(s)$ time constructible.*

3. We define

$$\mathsf{log\text{-}unif\text{-}DepthSize}(d,s) = \{L \mid \textit{there is a } d \textit{ depth and } s \textit{ size bounded logarithmic space uniform family of circuits that decides } L.\}$$

$$\mathsf{P\text{-}unif\text{-}DepthSize}(d,s) = \{L \mid \textit{there is a } d \textit{ depth and } s \textit{ size bounded polynomial time uniform family of circuits that decides } L.\}$$

Note that logarithmic space uniformity implies polynomial time uniformity. Typically, logarithmic space uniformity seems to be the more appropriate concept.

5.3 Simulating families of circuits by Turing machines

Theorem 5.3 *If $L \subseteq \{0,1\}^*$ is decided by a d depth bounded and s space constructible family of circuits C, then*

$$L \in \mathsf{DSpace}(d+s).$$

Proof. Let ξ be the given input, $|\xi| = n$. To evaluate the Boolean circuit C_n at a $\xi \in \{0,1\}^n$, we have to compute $\mathrm{val}(v, \xi)$ where v is the output node of C. To compute $\mathrm{val}(u, \xi)$ for some u we just have to find the predecessors u_1 and u_2 of u (or just one predecessor in the case of a $\neg$ gate or no predecessor in the case of an input gate). Then we compute recursively $\mathrm{val}(u_1, \xi)$ and $\mathrm{val}(u_2, \xi)$. From these to values, we easily obtain $\mathrm{val}(v, \xi)$.

To do this, we would need a stack of size $d(n)$, the depth of C_n. Each entry of the stack basically consists of two nodes. How much space do we need to write down the nodes? Since each node in a circuit has at most 2 predecessors, the number of nodes of C_n is bounded by $2^{d(n)}$. Thus we need $d(n)$ bits to write down a name of a node. Thus our stack needs $\mathrm{O}(d^2(n))$ many bits altogether. While this is not bad at all, it is more than promised in the theorem.

A second problem is the following: How do we get the predecessors of u? Just constructing the whole circuit would take too much space.

The second problem is easily overcome and we saw the solution to it before: Whenever we want to find out the predecessors of a node u, we simulate the Turing machine M constructing C_n, let it write down the edges one by one, always using the space again. Whenever we see an edge of the form (u', u), we have found a predecessor u' of u.

For the first problem, we again use the trick of recomputing instead of storing data. In the stack, we do not explicitly store the predecessors of a node. Instead we just write down which of the at most two predecessors we are currently evaluating (that means, the first or the second in the representation written by M). In this way, we only have to store a constant amount of information in each stack entry. The total size of the stack is $\mathrm{O}(d(n))$. To find the name of a particular node, we have to compute the names of all the nodes that were pushed on the stack before using M. But we can reuse the space each time.

Altogether, we need the space that is used for simulating M, which is $\mathrm{O}(s)$, and the space needed to store the stack, which is $\mathrm{O}(d)$. This proves the theorem. ∎

Remark 5.4 *If we assume that the family of circuits in the theorem is s size bounded and t time constructible, then the proof above shows that $L \in \mathsf{DTime}(t + \mathrm{poly}(s))$. The proof gets even simpler since we can construct the circuit explicitly and store encodings of the nodes in the stack. The best simulation known regarding time is given in the next exercise.*

Exercise 5.4 *If $L \subseteq \{0,1\}^*$ is decided by a s size bounded and t time constructible family of circuits C, then*

$$L \in \mathsf{DTime}(t + s \log^2 s).$$

5.4 Simulating Turing machines by families of circuits

In the "Theoretical Computer Science" lecture, we gave a size efficient simulation of Turing machines by circuits (Lemma 27.5), which we restate here for arbitrary time functions (but the proof stays the same!).

Theorem 5.5 *Let t be a time constructible function, and let $L \subseteq \{0,1\}^*$ be in $\mathsf{DTime}(t)$. Then there is a $\mathrm{O}(t^2)$ time constructible family of circuits that is $\mathrm{O}(t^2)$ size bounded and decides L.*

Remark 5.6 *If t is computable in $\mathrm{O}(\log t)$ space (this is true for all reasonable functions), then the family is also $\mathrm{O}(\log t)$ space constructible.*

Theorem 5.5 is a size efficient construction. The following result gives a depth efficient construction.

Theorem 5.7 *Let $s \geq \log$ be space constructible and let $L \subseteq \{0,1\}^*$ be in $\mathsf{NSpace}(s)$. Then there is a s space constructible family of circuits that is $\mathrm{O}(s^2)$ depth bounded and decides L.*

Before we give the proof, we need some definitions and facts. For two Boolean matrices $A, B \in \{0,1\}^{n\times n}$, $A \vee B$ denotes the matrix that is obtained by taking the Or of the entries of A and B componentwisely. $A \odot B$ denotes the Boolean matrix product of A and B. The entry in position (i,j) of $A \odot B$ is given by $\bigvee_{k=1}^{n} a_{i,k} \wedge b_{k,j}$. It is defined as the usual matrix product, we just replace addition by Boolean Or and multiplication by Boolean And. The mth Boolean power of a Boolean matrix A is defined as

$$A^{\odot m} = \underbrace{A \odot \cdots \odot A}_{m \text{ times}},$$

with the convention that $A^{\odot 0} = I$, the identity matrix.

Exercise 5.5 *Show the following:*

1. *There is an $\mathrm{O}(\log n)$ depth and $\mathrm{O}(n^3)$ size bounded logarithmic space uniform family of circuits C such that C_n computes the Boolean product of two given Boolean $n \times n$ matrices.*
2. *There is an $\mathrm{O}(\log^2 n)$ depth and $\mathrm{O}(n^3 \log n)$ size bounded uniform family of circuits D such that D_n computes the nth Boolean power of a given Boolean $n \times n$ matrix.*

Note that we here deviate a little from our usual notation. We only allow inputs of sizes $2n^2$ and n^2, respectively, for $n \in \mathbb{N}$. We measure the depths and size as a function in n (though it does not make any difference here).

For a graph G with n nodes, the incidence matrix of G is the Boolean matrix $E = (e_{i,j}) \in \{0,1\}^{n \times n}$ defined by

$$e_{i,j} = \begin{cases} 1 & \text{if there is an edge } (i,j) \text{ in } G, \\ 0 & \text{otherwise.} \end{cases}$$

Exercise 5.6 *Let G and E be as above.*

1. *Show that there is a path from i to j in G of length ℓ iff the entry of $E^{\odot \ell}$ in position (i,j) equals 1.*

2. *Show that there is a path from i to j in G iff the entry in position (i,j) of $(I \vee E)^{\odot n}$ equals 1.*

Proof of Theorem 5.7. Let M be an s space bounded nondeterministic Turing machine with $L(M) = L$. We may assume that M has a unique accepting configuration D. On inputs of length n, M has $c^{s(n)}$ many configurations. The idea is to construct the incidence matrix E of the configuration graph and then compute the $c^{s(n)}$th Boolean power of $I \vee E$. M accepts an input x iff the entry in the position corresponding to the pair $(\mathrm{SC}(x), D)$ in the $c^{s(n)}$th Boolean power of $I \vee E$ is 1.

Once we have constructed the matrix, we can use the circuit of Exercise 5.5. The size of the matrices is $c^{s(n)} \times c^{s(n)}$. A circuit for computing the $c^{s(n)}$the power of it is $\mathrm{O}(\log c^{s(n)}) = \mathrm{O}(s(n))$ space constructible and $\mathrm{O}(\log^2(c^{s(n)})) = \mathrm{O}(s^2(n))$ depth bounded.

Thus the only problem that remains is to construct a circuit that given x outputs a $c^{s(n)}) \times c^{s(n)}$ Boolean matrix that is the incidence matrix of the configuration graph of M with input x. This can be done as follows: We enumerate all pairs C, C' of possible configurations. In C, the head on the input tape is standing on some particular symbol, say, x_i. If $C \vdash_M C'$ independent of the value of x_i, then the output gate corresponding to (C, C') is 1. If $C \vdash_M C'$ only if $x_i = 0$, then the output gate corresponding to (C, C') computes $\neg x_i$. If $C \vdash_M C'$ only if $x_i = 1$, then the output gate corresponding to (C, C') computes x_i. Otherwise, it computes 0. Thus the circuit computing the matrix is very simple. It can be constructed in space $\mathrm{O}(s)$ since we only have to store two configurations at a time. ∎

Remark 5.8 *Theorem 5.3 and 5.7 yield an alternative proof of Savitch's theorem.*

5.5 Nick's Class

Circuits are a model of parallel computation. To be really faster than sequential computation, we want to have a exponential speedup for parallel

computations. That means if one wants to study circuits as a model of parallelism, the depth of the circuits should be polylogarithmic. On the other hand, we do not want too much "hardware". Thus the size of the circuits should be polynomial.

Definition 5.9

$$\mathsf{NC}_k = \bigcup_{i\in\mathbb{N}} \mathsf{log\text{-}unif\text{-}DepthSize}(\log^k(n), \mathrm{O}(n^i)) \qquad k = 1, 2, 3, \ldots$$
$$\mathsf{NC} = \bigcup_{k\in\mathbb{N}} \mathsf{NC}_k$$

NC stands for "Nick's Class". Nicholas Pippenger was the first one to study such classes. Steve Cook then chose this name.

Obviously,

$$\mathsf{NC}_1 \subseteq \mathsf{NC}_2 \subseteq \cdots \subseteq \mathsf{NC}.$$

By Theorem 5.3 and 5.7 and Remark 5.4

$$\mathsf{NC}_1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}_2 \subseteq \cdots \subseteq \mathsf{NC} \subseteq \mathsf{P}.$$

Problems in NL *have efficient parallel algorithms*

The inclusion $\mathsf{NC}_1 \subseteq \mathsf{L}$ suggests that logarithmic space uniformity is too strong for NC_1. There are solutions to this problem but we will not deal with it here.

Excursus: The division breakthrough

Often you find NC_k defined with respect to polynomial time uniformity instead of logarithmic space uniformity. One reason might be that for a long time, we knew that integer division was in polynomial time uniform NC_1 but it was not known whether it was in logarithmic space uniform NC_1. This was finally shown by Andrew Chiu in his Master's thesis. After that, Chiu attended law school.

Paul Beame, Steve Cook, James Hoover. Log depth circuits for division and related problems. *SIAM J. Comput.*, 15:994–1003, 1986.
Andrew Chiu, George I. Davida, Bruce E. Litow. Division in logspace-uniform NC_1. *Informatique Théoretique et Applications* 35:259-275, 2001.

---

6 NL, NC, and P

Lemma 6.1 *Let $L \subseteq \{0,1\}^*$ be* P*-complete under logarithmic space many-one reductions. If $L \in$* NC*, then* NC = P*.*

Proof. Let $L' \in \mathsf{P}$ be arbitrary. Since L is P-hard, $L' \leq_{\log} L$. Since $L \in \mathsf{NC}$, this means that there is a uniform family of circuits C that is $\log^k(n)$ depth and $\text{poly}(n)$ size bounded for some constant k and decides L. We want to use this family to get a uniform family of circuits for L' by using the fact that $L' \leq_{\log} L$.

Since $\mathsf{L} \subseteq \mathsf{NC}$, this is clear in principle, but there are some technical issues we have to deal with. First, we have to compute a function f and not decide a language. Second, a logspace computable function can map strings of the same length to images of different length. We deal with these problems as follows:

- Since f is in particular polynomial time computable, we know that $|f(x)| \leq p(|x|)$ for all x for some polynomial p. Instead of mapping x to $f(x)$, we map x to $0^{|f(x)|}1^{p(|x|)-|f(x)|}f(x)0^{p(|x|)-|f(x)|}$, that is, we pad $f(x)$ with 0's to length $p(|x|)$. In front we place another $p(x)$ bits indicating how long the actual string $f(x)$ is and how many 0's were added. This new function, call it f', is surely logarithmic space computable.

- We modify the family of circuits C such that it can deal with the strings of the form $f'(x)$. We only need a circuit for strings of lengths $2p(n)$. Such a circuit consists of copies of all circuits $C_1, C_2, \ldots, C_{p(n)}$. (This is still polynomial size!) C_i gets the first i bits of the second half of $f'(x)$. The first have of the bits of $f'(x)$ is used to decide which of the $p(|x|)$ circuits computes the desired result. Call this new family C'.

- Since f' is logarithmic space computable, the language

$$B = \{\langle x, i\rangle \mid \text{the } i\text{th bit of } f'(x) \text{ is } 1\}$$

  is in L. Since $\mathsf{L} \subseteq \mathsf{NC}$, there is a logarithmic space uniform family of circuits that decides B. Using these family, we can get circuits that we can use to feed the corresponding bits of $f'(x)$ into C'.

It is easy but a little lengthy to verify that the new family is still logarithmic space constructible. ∎

Thus a P complete problem is neither likely to have an algorithm that uses few space (even a nondeterministic one) nor to have an efficient parallel algorithm.

6.1 Circuit evaluation

Definition 6.2 *The* circuit value problem `CVAL` *is the following problem: Given (an encoding of) a circuit C with n inputs and one output and an input $x \in \{0,1\}^n$, decide whether $C(x) = 1$.*

Since C can only output two different values, the problem `CVAL` is basically equivalent to evaluating the circuit.

Theorem 6.3 `CVAL` *is* P*-complete.*

Proof. The proof of Theorem 5.3 together with Remark 5.4 basically shows that the problem is in P.

It remains to show the hardness. By Theorem 5.5, every language in $L \in \mathsf{P}$ is decided by a uniform family of circuits C whose size is polynomially bounded. Since the family is uniform, the function $1^n \mapsto C_n$ is logarithmic space computable. But then also $x \mapsto \langle C_{|x|}, x\rangle$ is logarithmic space computable since we only have to append the x to the description of the circuit. But this mapping is a logarithmic space reduction from L to `CVAL`. ■

Excursus: P-complete problems

If a problem is P-complete under logarithmic space many-one reductions, then this means that it does not have an efficient parallel algorithm by Lemma 6.1, unless you believe that $\mathsf{NC} = \mathsf{P}$. Here are some more P-complete problems:

Breadth-depth search: Given a graph G with ordered nodes and two nodes u and v, is u visited before v in a breadth-depth search induced by the vertex ordering?

Maximum flow: Given a directed graph G, a capacity function c on the edges, a source s and a target t and a bound f, is there a feasible flow from s to t of value $\geq f$.

Word problem for context-free grammars: Given a word w and a context-free grammar G, is $w \in L(G)$?

The following book contains and abundance of further problems:

Raymond Greenlaw, James Hoover, Walter L. Ruzzo. *Limits to Parallel Computation:* P*-Completeness Theory*, Oxford University Press, 1995.

There are also some interesting problems of which we do not know whether they are P-hard, for instance:

Integer GCD: Given two n-bit integers, compute their greatest common divisor. (This is a search problem, not a decision problem.)

Perfect Matching: Given a graph G, does it have a perfect matching? (We will see this problem again, when we deal with randomization.)

6.2 Steve's Class

So far, we compared P with L and NL. But also $L \in \mathsf{DSpace}(\log^k(n))$ for some constant k means that L has an algorithm with very few space. The only problem with this class is that we do not know whether it is contained in P. (For instance, with $\log^2 n$ space, you can count up to $n^{\log n}$. Though that is still a moderately growing function, it is not polynomial.) The right classes to study are the simultaneous time and space bounded classes

$$\mathsf{SC}_k = \bigcup_{i \in \mathbb{N}} \mathsf{DTimeSpace}(\mathrm{O}(n^i), \log^k n)$$
$$\mathsf{SC} = \bigcup_{i \in \mathbb{N}} \mathsf{SC}_i$$

These classes are known as *Steve's classes*. Nicholas Pippenger named them after Steve Cook because Steve was a nice guy who named the NC_k classes after Nicholas Pippenger before. Manus manum lavat.

7 The polynomial method

In this chapter, we prove a lower bound for the circuit size of constant depth unbounded fanin circuits for PARITY. The lower bounds even hold in the nonuniform setting.

7.1 Arithmetization

As a first step, we represent Boolean functions $\{0,1\}^n \to \{0,1\}$ as polynomials $p \in \mathbb{R}[X_1, \ldots, X_n]$. A Boolean function is *represented* by a polynomial p if

$$p(x) = f(x) \qquad \text{for all } x \in \{0,1\}^n$$

Above, we embed the Boolean values $\{0,1\}$ into $\mathbb{R}$ by mapping 0 (false) to 0 and 1 (true) to 1.

Since $x^2 = x$ for all $x \in \{0,1\}$, whenever a monomial in the polynomial p contains a factor X_i^j, we can replace it by X_i and the polynomial still represents the same Boolean function. Therefore, we can always assume that the representing polynomial is multilinear. In this case, the representing polynomial is unique.

Exercise 7.1 *Show the following: Let $f : \{0,1\}^n \to \{0,1\}$. If p is a multilinear polynomial with $p(x) = f(x)$ for all $x \in \{0,1\}^n$, then p is unique.*

Example 7.1

1. *The Boolean AND of n inputs is represented by the degree n polynomial $X_1 X_2 \cdots X_n$.*
2. *The Boolean NOT is represented by $1 - X$.*
3. *The Boolean OR is represented by $1 - (1 - X_1)(1 - X_2) \cdots (1 - X_n)$ (de Morgan's law).*

7.2 Approximation

One potential way to prove lower bounds is the following:

1. Introduce some complexity measure or potential function.
2. Show that functions computed by devices of type X have low complexity.
3. Show that function Y has high complexity.

A complexity measure that comes to mind is the degree of the representing polynomial. However, since Boolean AND and Boolean OR have representing polynomials of degree n, there is no hope that constant depth unbounded fanin circuits have low degree. However, we can show that Boolean AND and Boolean OR can be *approximated* by low degree polynomials in the following sense: A Boolean function is *randomly approximated with error probability* ϵ by a family of polynomials P if

$$\Pr_{p \in P}[p(x) = f(x)] \geq 1 - \epsilon \qquad \text{for all } x \in \{0,1\}^*.$$

Proof overview: This suggest the following route:

1. Unbounded fanin Boolean AND and Boolean OR can be approximated by low degree polynomials, i.e., degree $\mathrm{O}(\log n)$.
2. Boolean functions that are approximated by unbounded fanin circuits of size s and depth d have degree $\mathrm{O}(\log^{d+1} s)$.
3. `PARITY` cannot be approximated by small degree polynomials.

We start with a technical lemma.

Lemma 7.2 *Let S_0 be a set of size n. Let $\ell = \log n + 2$. Starting with S_0, iteratively construct a tower $S_0 \supseteq S_1 \supseteq S_2 \supseteq \cdots \supseteq S_\ell$ by putting each element of S_{i-1} into S_i with probability $1/2$. Then for all non-empty $T \subseteq S_0$,*

$$\Pr[\textit{there is an } i \textit{ such that } |S_i \cap T| = 1] \geq 1/2$$

Proof. We consider three cases:
Bad case: $|T \cap S_\ell| > 1$. We have

$$\begin{aligned}\Pr[|T \cap S_\ell| > 1] &\leq \Pr[|T \cap S_\ell| \geq 1] \\ &\leq n \cdot 2^{-\ell} \\ &\leq 1/4,\end{aligned}$$

since $|T \cap S_\ell| \geq 1$ means that at least one element survived all ℓ coin flips.
Very good case: $|T| = |T \cap S_0| = 1$.
Good case: $|T \cap S_0| > 1$ and there is an i such that $|T \cap S_i| \leq 1$. We can assume that $|T \cap S_{i-1}| = s > 1$. Then the probability that $T \cap S_i$ has exactly one element is the probability that all but one elements do not survive the coin flip, which is $s \cdot 2^{-s}$ divided by the probability that after the coin flips $|T \cap S_i| \leq 1$. The latter probability is $(s+1) \cdot 2^{-s}$. Thus the overall probability is $s/(s+1) \geq 2/3$.

With probability 3/4, we are in the very good or good case. If we are in the very good or good case, then with probability $\geq 2/3$, there is an i such that $|T \cap S_i| = 1$. Thus we have success with probability at least $3/4 \cdot 2/3 = 1/2$. ∎

Lemma 7.3 *Let $1 > \epsilon > 0$. There is a family of polynomials P of degree $O(\log(1/\epsilon) \cdot \log n)$ such that*

$$\Pr_{p \in P}[\bigvee_{i=1}^{n} x_i = p(x)] \geq 1 - \epsilon \qquad \text{for all } x \in \{0,1\}^n.$$

Proof. We construct random sets $S_0, \ldots, S_\ell$ like in Lemma 7.2. Let

$$p_1(x) = \left(1 - \sum_{j \in S_0} x_j\right) \cdots \left(1 - \sum_{j \in S_\ell} x_j\right)$$

If all x_j's are zero, then all the sums in p_1 are zero and $p_1(x) = 1$. Next comes the case where at least one $x_j = 1$. Let $T = \{j \mid x_j \neq 0\}$. By Lemma 7.2, there is an i such that $|S_i \cap T| = 1$ with probability $\geq 1/2$, i.e, the ith factor of p_1 is zero with probability $\geq 1/2$. Now instead of one p_1, we take k independent instances $p_1, \ldots, p_k$ and set $\hat{p} = p_1 \ldots p_k$. If all x_j's are zero then $\hat{p}(x) = 1$. If at least one $x_j = 1$, then at least one $p_k(x) = 0$ with probability $\geq 1 - (1/2)^k \geq 1 - \epsilon$ for $k \geq \log(1/\epsilon)$ and henceforth, $\hat{p}(x) = 0$. Thus $1 - \hat{p}(x) = \bigvee_{i=1}^{n} x_i$ happens with probability $\geq 1 - \epsilon$ for all $x \in \{0,1\}^n$. ∎

Exercise 7.2 *Show that $\bigwedge_{i=1}^{n} x_i$ can be approximated in the same way.*

Lemma 7.4 *Let $f : \{0,1\}^n \to \{0,1\}$ be computed by an s size bounded and d depth bounded circuit of unbounded fanin. Then f can be randomly approximated with error probability ϵ by a family of polynomials of degree $O(\log^d(s/\epsilon) \cdot \log^d(s))$.*

Proof. We replace every OR or AND gate by a random polynomial from a family with error probability ϵ/s. Each polynomial has degree $O(\log(s/\epsilon) \cdot \log(s))$, since every gate can have at most s inputs. The degree of the polynomial at the output gate is $O(\log^d(s/\epsilon) \cdot \log^d(s))$, since the composition of polynomials multiplies the degree bounds.

If all polynomials compute their respective gate correctly, then the polynomial computes the function f correctly. The probability that a single polynomial fails is at most ϵ/s. By a union bound, the overall error probability is thus at most $s\epsilon/s = \epsilon$. ∎

Corollary 7.5 *Let $f : \{0,1\}^n \to \{0,1\}$ be computed by an s size bounded and d depth bounded unbounded fanin circuit. Then there is a polynomial p of degree $\mathrm{O}(\log^d(s/\epsilon) \cdot \log^d(s))$ such that*

$$\Pr_{x \in \{0,1\}^n}[f(x) = p(x)] \geq 1 - \epsilon.$$

Proof. For every x, $\Pr_{p \in P}[f(x) = p(x)] \geq 1 - \epsilon$, where P is the corresponding approximating family. Thus, $\Pr_{x,p}[f(x) = p(x)] \geq 1 - \epsilon$. Thus, there must be at least one p that achieves this probability. ∎

Exercise 7.3 *The statement of Corollary 7.5 is sufficient for our proof. Why does the following argument not work: The zero polynomial and the one polynomial approximate the AND and OR polynomial with error probability $1 - 2^{-n}$. Now take a circuit as in Lemma 7.4 and do the same construction with these polynomials.*

7.3 Parity

So far, we identified the truth value 0 with the natural number 0 and the truth value 1 with the natural number 1. Why this looks natural, in the following it is advantageous to work with the representation -1 for the truth value 1 and 1 for the truth value 0. This representation is also called the *Fourier representation*.

The linear function $1 - 2x$ maps 0 to 1 and 1 to -1. Its inverse function is $\frac{1}{2}(1 - x)$. Thus we can switch between these two representations without changing the degrees of the polynomials in the previous sections.

Using the Fourier representation, the parity function can be written as $\prod_{i=1}^{n} x_i$.

Lemma 7.6 *There is no polynomial p of degree $\leq \sqrt{n}/2$ such that*

$$\Pr_{x \in \{-1,1\}^n}[p(x) = x_1 \cdots x_n] \geq 0.9.$$

Proof. Let p be a polynomial of degree $\leq \sqrt{n}/2$. As seen above, we can assume that p is multilinear. Let $A = \{x \in \{-1,1\}^n \mid p(x) = x_1 \cdots x_n\}$.

Let V be the $\mathbb{R}$-vector space of all functions $A \to \mathbb{R}$. Its dimension is $|A|$.

The set M of all multilinear polynomials of degree $\leq (n + \sqrt{n})/2$ forms a vector space, too. A basis of this vector space are all multilinear monomials

of degree $\leq (n+\sqrt{n})/2$. Thus

$$\begin{aligned}
\dim M &= \sum_{i=0}^{(n+\sqrt{n})/2} \binom{n}{i} \\
&= \sum_{i=0}^{n/2} \binom{n}{i} + \sum_{i=n/2+1}^{n/2+\sqrt{n}/2} \binom{n}{i} \\
&\leq \frac{1}{2} \cdot 2^n + \frac{\sqrt{n}}{2} \binom{n}{n/2} \\
&\leq \frac{1}{2} \cdot 2^n + \frac{\sqrt{n}}{2} \frac{2^n}{\sqrt{\pi n/2}} \qquad \text{(Stirling's formula)} \\
&< 0.9 \cdot 2^n.
\end{aligned}$$

Finally we show that every function in V can be represented by an element of M on A. Then

$$|A| = \dim V \leq \dim M < 0.9 \cdot 2^n.$$

and we are done.

There is a natural isomorphism between the vector space of all functions $\{-1,1\}^n \to \{-1,1\}$ and the vector space of *all* multilinear polynomials, cf. Exercise 7.1. Let $\prod_{i\in I} x_i$ be some multilinear monomial of degree $> n/2$. Then

$$\prod_{i\in I} a_i = \prod_{i\in I} a_i \left(\prod_{i\notin I} a_i\right)^2 = p(a) \prod_{i\notin I} a_i$$

for all $a \in A$. Thus for all functions $A \to \mathbb{R}$, the monomials of degree $(n+\sqrt{n})/2$ are sufficient to represent them. This concludes the proof. ∎

Corollary 7.7 *1. Any constant depth d circuit that computes parity on n inputs has size at least $2^{\Omega(\sqrt[2d]{n})}$.*

2. Any polynomial size circuit that computes parity on n inputs has depth at least $\Omega(\log n / \log\log n)$.

Excursus: AC

AC_i is the class of all languages that are recognized by logarithmic space uniform unbounded fanin circuits with depth $O(\log^i n)$ and polynomial size. Let $\mathsf{AC} = \bigcup_{i\in\mathbb{N}} \mathsf{AC}_i$.

If a circuit has polynomial size, then every gate can have at most a polynomial number of inputs. Thus we can simulate every unbounded fanin gate by a binary tree of logarithmic depth. Hence we get

$$\mathsf{AC}_0 \subseteq \mathsf{NC}_1 \subseteq \mathsf{AC}_1 \subseteq \mathsf{NC}_2 \subseteq \ldots$$

and $\mathsf{AC} = \mathsf{NC}$. Corollary 7.7 in particular shows that $\mathsf{AC}_0 \neq \mathsf{NC}_1$. We do not know whether any of the other inclusions is strict.

Exercise 7.4 *Show that* $\texttt{PARITY} \in \mathsf{NC}_1$.

8 P and NP

8.1 NP and quantifiers

Beside the usual definition of NP, there is an equivalent one based one verifiers. We showed this characterization in the "Theoretical Computer Science" lecture.

Definition 8.1 *A deterministic polynomial time Turing machine M is called a* polynomial time verifier *for $L \subseteq \Sigma^*$, if there is a polynomial p such that the following holds:*

1. *For all $x \in L$ there is a $c \in \{0,1\}^*$ with $|c| \leq p(|x|)$ such that M accepts $\langle x, c\rangle$.*
2. *For all $x \notin L$ and all $c \in \{0,1\}^*$, M on input $\langle x, c\rangle$ reads at most $p(|x|)$ bits of c and always rejects $\langle x, c\rangle$.*

We denote the language L that M verifies by $V(M)$.

The string c serves as a *certificate* (or *witness* or *proof*) that x is in L. A language L is verifiable in polynomial time if each x in L has a polynomially long proof. For each x not in L no such proof exists.

Note that the language $V(M)$ that a verifier verifies is not the language that it accepts as a "normal" Turing machine. $L(M)$ can be viewed as a binary relation, the pairs of all (x, c) such that M accepts $\langle x, c\rangle$, i.e., $M(\langle x, c\rangle) = 1$.

The following theorem is proven in the "Theoretical Computer Science" lecture (as Theorem 26.5):

Theorem 8.2 *$L \in$ NP iff there is a polynomial time verifier for L.*

Let M be a polynomial time verifier for L. As described above, we can view $L(M)$ as a binary relation. We denote this relation by R. Instead for writing $(x, c) \in R$ we will also write $R(x, c) = 1$. Thus $R(x, c) = 1$ iff $M(\langle x, c\rangle) = 1$. Thus, a language L is in NP if and only if there is a polynomial p and a polynomial time computable relation R such that the following holds:

$$x \in L \iff \exists y \in \{0,1\}^{p(|x|)} : R(x, y) = 1. \tag{8.1}$$

The string y models the nondeterministic choices of the Turing machine.

Recall that co-NP is the class of all L such that $\bar{L} \in$ NP. Thus L is in co-NP if there is a polynomial time bounded nondeterministic Turing machine M such that for all $x \in L$, each path in the computation tree of M is accepting and for all $x \notin L$, there is at least one rejecting path in the computation tree. This gives us a characterization in terms of certificates for co-NP: co-NP is characterized via

$$x \in L \iff \forall y \in \{0,1\}^{p(|x|)} : R(x,y) = 1.$$

In other words, languages in co-NP have polynomially long proofs for non-membership.

In the following, $\exists^P y$ and $\forall^P y$ means $\exists y \in \{0,1\}^{p(|x|)}$ and $\forall y \in \{0,1\}^{p(|x|)}$, respectively, for some polynomial p.

8.2 NP-complete problems

The class NP is very important, since it characterizes the complexity of an abundance of relevant problems. The most prominent of them is probably the satisfiability problem. It comes in several variations:

Definition 8.3 *1.* CSAT *is the following problem: Given (an encoding of) a Boolean circuit C, decide whether there is a Boolean vector ξ with $C(\xi) = 1$.*

2. SAT *is the following problem: Given (an encoding of) a Boolean formula in CNF, decide whether there is a satisfying assignment.*

3. ℓSAT *is the following problem: Given (an encoding of) a Boolean formula in ℓ-CNF, decide whether there is a satisfying assignment.*

We usually use polynomial time many one reductions to compare problems in NP. However, we do not know of any problem that is NP-complete under polynomial time many one reductions but not complete under logarithmic space polynomial time reductions. Obviously,

$$\ell\texttt{SAT} \leq_{\mathrm{P}} \texttt{SAT} \leq_{\mathrm{P}} \texttt{CSAT}.$$

ℓSAT is obviously a special case of SAT and so is SAT a special case of CSAT. For the latter note that any formula can be interpreted as a circuit. We showed that 3SAT is NP-complete, in turn, SAT and CSAT are NP-complete, too.

Definition 8.4 TAUT *is the following problem: Given a formula in DNF, decide whether it is a tautology, i.e., whether all assignments satisfy it.*

TAUT is co-NP-complete. Let UNSAT be the encodings of all formulas in CNF that are not satisfiable. Note that UNSAT is not the complement of SAT. The complement of SAT is UNSAT together with all strings that are not an encoding of a formula in CNF. But since such strings can be recognized in polynomial time, we get that $\overline{\texttt{SAT}} \leq_{\mathrm{P}} \texttt{UNSAT}$. But a formula F is unsatisfiable iff $\neg F$ is a tautology. If F is in CNF, then we can compute the DNF of $\neg F$ in polynomial time using De Morgan's law. Thus $\texttt{UNSAT} \leq_{\mathrm{P}} \texttt{TAUT}$. Since SAT is NP-complete, $\overline{\texttt{SAT}}$ is co-NP-complete and so is TAUT.

8.3 Self reducibility

Proving existence versus searching

Is showing the existence of a proof easier than finding the proof itself?

Maybe in real life but not for NP-complete problems ...

Assume we have a polynomial time deterministic algorithm for SAT. Then given a formula F, we can find out whether it is satisfiable or not in polynomial time. But what we really want is a satisfying assignment.

Let us first have a look at SAT. SAT has a nice property, we can reduce questions about a formula F in CNF to questions about smaller formulas. Let x be a variable of F. Let F_0 and F_1 be the two formulas that are obtained by setting x to 0 or 1, respectively, and then removing clauses that are satisfied by this and removing literals that became 0. (This proceed can produce an empty clause. Such a formula is not satisfiable by definition. Or the procedure could produce the empty formula in CNF. This one is satisfiable.) Then F is satisfiable if and only if F_0 or F_1 is satisfiable. Note that the length of F_0 and F_1 is smaller than the length of F.

Definition 8.5 *A language A is called* downward self-reducible*, if there is a polynomial time oracle deterministic Turing machine M that on input x, only queries oracle strings with length $< |x|$ such that $A = L(M^A)$.*

Without the restriction that M can only query strings of smaller size, this would not be a useful concept since M could query the oracle about the input itself.

The considerations above show the following result.

Theorem 8.6 SAT *is downward self-reducible. The same is true for* CSAT*.*

Exercise 8.1 *Show that if A is downward self-reducible, then $A \in$ PSPACE.*

Exercise 8.2 *Let M be a deterministic Turing machine that only queries oracle strings that are shorter than the input string. Show that if $A = L(M^A)$ and $B = L(M^B)$ then $A = B$. (Hint: show that for all n, $A^{\leq n} = B^{\leq n}$ using induction.)*

For each problem $A \in \mathsf{NP}$, there is a relation R that is polynomial time computable such that (8.1) holds. But vice versa, each such relation R defines a language in NP via

$$L(R) = \{x \mid \exists^P y : R(x, y) = 1\}.$$

We call such an R an NP-*relation* or *polynomially bounded relation.* Given such a NP relation R, search(R) is the set of all functions

$$x \mapsto \begin{cases} y & \text{with } R(x, y) = 1 \text{ if such a } y \text{ exists} \\ \text{undef} & \text{otherwise.} \end{cases}$$

Example 8.7 *Let R be the relation corresponding to* SAT*, i.e., $R(x, y) = 1$ if x encodes a formula F and y is a satisfying assignment of F.*

1. $L(R) =$ SAT.
2. search(R) *is the set of all functions that map a formula F to a satisfying assignment if one exists.*

We call search(R) *self-reducible* if there is an $f \in$ search(R) such that $f = M^{L(R)}$ for some polynomial time deterministic oracle Turing machine M. That means, we can reduce the problem of finding a certificate to the decision problem.

Recall that $M^{L(R)}$ denotes that function that is computed by M with oracle $L(R)$. For our example this means that we could find a satisfying assignment in polynomial time given that SAT $\in$ P.

Theorem 8.8 *Let R be an* NP*-relation. If $L(R)$ is* NP*-complete, then* search(R) *is self-reducible.*

Proof. A deterministic Turing machine M for computing $f \in$ search(R) works as follows:

Input: x and oracle access to $L(R)$
Output: "undef" or a certificate y such that $R(x, y) = 1$.

1. Ask whether $x \in L(R)$. If no, then output "undef".
2. Let N be some polynomial time verifier that computes R. From N, we get a circuit C such that each satisfying assignment y is a proof that $x \in L(R)$. This construction basically is the same one as the construction that shows that CSAT is NP-complete and can be performed in polynomial time.

3. Since $L(R)$ is NP-complete, there is a polynomial time many one reduction f from CSAT to $L(R)$.

4. Since CSAT is downward selfreducible, we can find a y such that $C(y) = 1$ in polynomial time provided we have an oracle to CSAT. Instead of asking whether $x \in$ CSAT, we ask whether $f(x) \in L(R)$. Since f is a many one reduction, these questions are equivalent.

5. Such a y fulfills $R(x, y) = 1$ by construction. Return y.

The above procedure can be performed by a polynomial time deterministic Turing machine with oracle access to $L(R)$. It computes a function in search(R) by construction. Thus search(R) is self-reducible. ∎

In other words, the theorem above says that for NP-complete problems, computing a witness for membership is only polynomially harder than deciding membership.

Search problems in the context of NP were introduced by Leonid Levin.

8.4 NP and co-NP

One approach to show that P ≠ NP would be to show that NP is not closed under complementation, i.e., NP ≠ co-NP. To show that NP is closed under complementation, it is sufficient to show that an NP-complete problem is in co-NP.

Theorem 8.9 *If* co-NP *contains an* NP*-complete problem, then* NP = co-NP.

Proof. Let $L \in$ co-NP be NP-complete.

Let $A \in$ NP arbitrary. Since L is NP-complete, there is a polynomial time many one reductions f from A to L. But since $L \in$ co-NP, A is also in co-NP, since we can write $A = \{x \mid \forall^P y : R(f(x), y) = 1\}$ for some polynomial time computable relation R with $L = L(R)$. Thus NP ⊆ co-NP.

Let $B \in$ co-NP. If L is NP-complete, then $\bar{L}$ is co-NP-complete by Exercise 3.2. A similar argument as above now shows that $B \in$ NP. ∎

A natural co-NP-complete problem is UNSAT, another one is TAUT. But we do not know whether they are in NP or not. Most researchers conjecture that they are not.

What is the relation between P and NP ∩ co-NP? PRIMES, the problem whether a given number (in binary) is a prime number, was *the* example of an interesting language in NP∩co-NP that is not known to be in P. Recently, Agrawal, Kayal, and Saxena showed that PRIMES ∈ P. (Maybe this is again a good point in time to do some advertisement for the complexity theory seminar next semester.)

Here is another problem that is in NP ∩ co-NP that is not known to be in P.

Definition 8.10 FACTOR *is the following problem: Given two numbers x and c in binary, decide whether x has a factor b with $2 \leq b \leq c$.*

Exercise 8.3 *Prove the following:*

1. FACTOR $\in$ NP.
2. FACTOR $\in$ co-NP. *(You can use that* PRIMES $\in$ P*)*

9 The polynomial time hierarchy

Alternating quantifiers

- Give a theoretical computer scientists some operators!
- Teach him recursion!
- Lean back and watch ...

9.1 Alternating quantifiers

A language L is in NP if and only if there is a polynomial time computable relation R such that the following holds:

$$x \in L \iff \exists^P y : R(x, y) = 1.$$

The string y models the nondeterministic choices of the Turing machine. In the same way, co-NP is characterized via

$$x \in L \iff \forall^P y : R(x, y) = 1.$$

Given such a definition as above, theorists do not hesitate to generalize them and see what happens: More precisely, a language is in the class Σ_k^{P} if there is a polynomial-time computable $(k+1)$-ary relation R such that

$$x \in L \iff \exists^P y_1 \forall^P y_2 \ldots Q^P y_k : R(x, y_1, \ldots, y_{k-1}, y_k) = 1.$$

Above, Q^P stands for $\exists^P$ if k is odd and for $\forall^P$ otherwise. In the same way, the classes Π_k^{P} are defined: A language is in the class Π_k^{P} if there is a polynomial-time computable relation R such that

$$x \in L \iff \forall^P y_1 \exists^P y_2 \ldots Q^P y_k : R(x, y_1, \ldots, y_{k-1}, y_k) = 1.$$

By definition, $\mathsf{NP} = \Sigma_1^{\mathsf{P}}$ and $\mathsf{co\text{-}NP} = \Pi_1^{\mathsf{P}}$.

The following inclusions are easy to show.

Exercise 9.1 *Show that for all k,*

$$\Sigma_k^{\mathsf{P}} \subseteq \Sigma_{k+1}^{\mathsf{P}},$$
$$\Pi_k^{\mathsf{P}} \subseteq \Pi_{k+1}^{\mathsf{P}},$$
$$\Sigma_k^{\mathsf{P}} \subseteq \Pi_{k+1}^{\mathsf{P}},$$
$$\Pi_k^{\mathsf{P}} \subseteq \Sigma_{k+1}^{\mathsf{P}}.$$

The union of these classes

$$\mathsf{PH} = \bigcup_{k \geq 1} \Sigma_k^{\mathsf{P}} = \bigcup_{k \geq 1} \Pi_k^{\mathsf{P}}$$

is called the *polynomial time hierarchy*. We have

$$\mathsf{PH} \subseteq \mathsf{PSPACE},$$

since we can check all possibilities for $y_1, \ldots, y_k$ in polynomial-space.

We can generalize this concept to arbitrary complexity classes. For a polynomial p, let $\exists y, |y| \leq p(n)$ be denoted by $\exists^p x$. For a language L, and a polynomial p,

$$\exists^p L = \{x \mid \exists^p y : \langle x, y \rangle \in L\}.$$

In the same way,

$$\forall^p L = \{x \mid \forall^p y : \langle x, y \rangle \in L\}.$$

For some complexity class C,

$$\exists \mathsf{C} = \bigcup_{p \text{ a polynomial}} \{\exists^p L \mid L \in \mathsf{C}\},$$

$$\forall \mathsf{C} = \bigcup_{p \text{ a polynomial}} \{\forall^p L \mid L \in \mathsf{C}\}.$$

Theorem 9.1 *We have*

$$\Sigma_k^{\mathsf{P}} = \exists \forall \exists \ldots Q \ \mathsf{P},$$
$$\Pi_k^{\mathsf{P}} = \forall \exists \forall \ldots Q \ \mathsf{P}$$

where each sequence consists of k alternating quantifiers.

Proof. The proof is by induction on k.
Induction base: Clearly, $\exists \mathsf{P} = \mathsf{NP} = \Sigma_1^{\mathsf{P}}$ and $\forall \mathsf{P} = \mathsf{co\text{-}NP} = \Pi_1^{\mathsf{P}}$.
Induction step: Let $k > 1$. We only show the induction step for Σ_k^{P}, the case Π_k^{P} is proved in a similar fashion. By the induction hypothesis,

$$\exists \forall \exists \ldots Q \ \mathsf{P} = \exists \Pi_{k-1}^{\mathsf{P}}.$$

Let $L \in \exists \forall \exists \ldots Q \ \mathsf{P}$. This means that there is a language $A \in \Pi_{k-1}^{\mathsf{P}}$ such that $x \in L$ iff there is some y of polynomial length such that $\langle x, y \rangle \in A$. But there is a relation R such that $\langle x, y \rangle$ is in A iff

$$\forall^P y_1 \exists^P y_2 \ldots Q^p y_{k-1} : R(\langle x, y \rangle, y_1, \ldots, y_{k-1}) = 1.$$

Let R' be the relation defined by $R'(x, y, y_1, \ldots, y_{k-1}) = R(\langle x, y \rangle, y_1, \ldots, y_{k-1})$. Clearly R' is polynomial time computable iff R is. Thus $x \in L$ iff

$$\exists^P y \forall^P y_1 \exists^P y_2 \ldots Q^P y_{k-1} : R'(x, y, y_1, \ldots, y_{k-1}) = 1.$$

But this means that $L \in \Sigma_k^{\mathsf{P}}$. Thus $\exists\forall\exists \ldots Q\ \mathsf{P} \subseteq \Sigma_k^{\mathsf{P}}$. Since the argument above can be reversed, the opposite inclusion is true, too. ∎

It is an open question whether the polynomial time hierarchy is infinite, that means, $\Sigma_i^{\mathsf{P}} \subsetneq \Sigma_{i+1}^{\mathsf{P}}$ for all i. The next theorem shows that in order to show that the polynomial time hierarchy is not infinite, it suffices to find an i such that $\Sigma_i^{\mathsf{P}} = \Pi_i^{\mathsf{P}}$.

Theorem 9.2 *If* $\Sigma_i^{\mathsf{P}} = \Pi_i^{\mathsf{P}}$, *then* $\Sigma_i^{\mathsf{P}} = \Pi_i^{\mathsf{P}} = \mathsf{PH}$.

We need the following lemma for the proof.

Lemma 9.3 *For all classes* C,

$$\forall\forall\mathsf{C} = \forall\mathsf{C},$$
$$\exists\exists\mathsf{C} = \exists\mathsf{C},$$

provided that $\langle .,. \rangle$ *and the corresponding inverse projections are linear time computable and* C *is closed under linear time transformations of the input.*

Proof. We only prove the first statement, the proof of the second one is similar. Let L be in $\forall\forall\mathsf{C}$. This means that there is a language $A \in \forall\mathsf{C}$ such that

$$x \in L \iff \forall^P y : \langle x, y \rangle \in A$$

Since $A \in \forall\mathsf{C}$, there is some $B \in \mathsf{C}$ such that

$$a \in A \iff \forall^P b : \langle x, y \rangle \in B.$$

Thus

$$x \in L \iff \forall^P y \forall^P b : \langle \langle x, y \rangle, b \rangle \in B.$$

Now we want to replace the two quantifiers by one big one quantifying over $\langle y, b \rangle$. There is only one technical problem: Words in B are of the form $\langle \langle x, y \rangle, b \rangle$ but we need words of the form $\langle x, \langle y, b \rangle \rangle$. Define B' by

$$B' = \{ \langle x, \langle y, b \rangle \rangle \mid \langle \langle x, y \rangle, b \rangle \in B \}. \tag{9.1}$$

B' is again in C, since $\langle \langle x, y \rangle, b \rangle$ is computable in linear time from $\langle x, \langle y, b \rangle \rangle$ and C is closed under linear time transformations of the input.

Now we are almost done but there is one little problem left: We cannot quantify over all $\langle y, b \rangle$, we can only quantify over all $z \in \{0,1\}^{p(n)}$. Some strings z might not correspond to pairs $\langle y, b \rangle$, since either y or b is longer than the polynomial of the corresponding $\forall^P$-quantifier in (9.1). B'' now contains all the words that are in B' and in addition all words $\langle x, z \rangle$ such that z is not a valid pair. Since we can also find out in linear time, whether z is a valid pair, $B'' \in \mathsf{C}$, too.

By construction,

$$x \in L \iff \forall^P z : \langle x, z \rangle \in B''.$$

Thus $L \in \forall C$. ■

Technicalities

The condition of linear time computability and being closed under linear time transformations is fairly arbitrary. Usually, polynomial time computability and being closed under polynomial time reductions is enough. But since $\exists$ and $\forall$ are pretty general operators, we have tried to put as few as possible constraints on C.

(Note that there are linear time computable pairing functions.)

Proof of Theorem 9.2. We show that if $\Sigma_i^\mathsf{P} = \Pi_i^\mathsf{P}$, then $\Sigma_i^\mathsf{P} = \Sigma_{i+1}^\mathsf{P} = \Pi_{i+1}^\mathsf{P}$. Then the theorem follows by induction.

We have $\Sigma_{i+1}^\mathsf{P} = \exists\Pi_i^\mathsf{P}$. Since $\Sigma_i^\mathsf{P} = \Pi_i^\mathsf{P}$, $\Sigma_{i+1}^\mathsf{P} = \exists\Sigma_i^\mathsf{P}$. By Theorem 9.1 and Lemma 9.3, $\Sigma_{i+1}^\mathsf{P} = \Sigma_i^\mathsf{P}$. In the same way we get $\Pi_{i+1}^\mathsf{P} = \Pi_i^\mathsf{P}$. This proves our claim above. ■

Most researchers believe that the polynomial time hierarchy is infinite. So whenever some assumption makes the polynomial time hierarchy collapse, then this assumption is most likely not true (where the probability is taken over the opinions of all researchers in complexity theory). Here "PH collapses" means that $\mathsf{PH} = \Sigma_i$ for some i.

9.2 Complete problems

Let F be a Boolean formula over some variables X. A quantified Boolean formula is a formula of the form

$$Q_1 x_{i_1} \dots Q_n x_{i_n} F(x_1, \dots, x_n).$$

where each Q_i is either $\exists$ or $\forall$. (Note that the x_i are Boolean variables here that can attain values from $\{0, 1\}$ only.) A quantifier alternation is an index j such that $Q_j \neq Q_{j+1}$, i.e., an existential quantifier is followed by a universal one or vice versa. We will always assume that there are no free variables, i.e., the formula is closed.

Definition 9.4 *1.* $\mathtt{QBF}$ *is the following problem: Given a closed quantified Boolean formula, is it true?*

2. $\mathtt{QBF}\Sigma_k$ *is the following problem: Given a closed quantified Boolean formula starting with an existential quantifier and with* $\leq k-1$ *quantifier alternations, is it true?*

3. $\texttt{QBF}\Pi_k$ *is the following problem: Given a closed quantified Boolean formula starting with a universal quantifier and with* $\leq k-1$ *quantifier alternations, is it true?*

We will show in the next chapter that QBF is PSPACE-complete. $\texttt{QBF}\Sigma_k$ and $\texttt{QBF}\Pi_k$ are complete problems for Σ_k^{P} and Π_k^{P}, respectively.

Exercise 9.2 *Show that* $\texttt{QBF}\Sigma_k$ *is* Σ_k^{P}*-complete.*

On the other hand, PH most likely does not have complete problems.

Exercise 9.3 *If* PH *has complete problems, then* PH *collapses.*

9.3 A definition in terms of oracles

For a language A, $\mathsf{DTime}^A(t)$ denotes the set of all languages that are decided by a t time bounded deterministic Turing machine M with oracle A. In the same way, we define $\mathsf{NTime}^A(t)$, $\mathsf{DSpace}^A(s)$, and $\mathsf{NSpace}^A(s)$. If C is a set of languages, then $\mathsf{DTime}^{\mathsf{C}}(t) = \bigcup_{A\in\mathsf{C}} \mathsf{DTime}^A(t)$. In the same way, we define $\mathsf{NTime}^{\mathsf{C}}(t)$, $\mathsf{DSpace}^{\mathsf{C}}(s)$, and $\mathsf{NSpace}^{\mathsf{C}}(s)$. Finally, if T is some set of functions $t : \mathbb{N} \to \mathbb{N}$, then $\mathsf{DTime}^{\mathsf{C}}(T) = \bigcup_{t\in T} \mathsf{DTime}^{\mathsf{C}}(t)$. We do the same for $\mathsf{NTime}^{\mathsf{C}}(T)$, $\mathsf{DSpace}^{\mathsf{C}}(S)$, and $\mathsf{NSpace}^{\mathsf{C}}(S)$.

Let $S_1 = \mathsf{NP}$ and $S_i = \mathsf{NP}^{S_{i-1}}$. In other words, $S_2 = \mathsf{NP}^{\mathsf{NP}}$, $S_3 = \mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}}}$, and so on. S_i is another definition of the polynomial time hierarchy. More precisely, we have the following theorem, where $P_i = \mathsf{co\text{-}NP}^{S_{i-1}}$.

Theorem 9.5 *For all* i, $\Sigma_i^{\mathsf{P}} = S_i$ *and* $\Pi_i^{\mathsf{P}} = P_i$.

Proof. The proof is by induction on i.
Induction base: For $i = 1$, we have $\Sigma_1^{\mathsf{P}} = \mathsf{NP} = S_1$ and $\Pi_1^{\mathsf{P}} = \mathsf{co\text{-}NP} = P_1$.
Induction step: Assume that the claim is valid for i. We first show that $\Sigma_{i+1}^{\mathsf{P}} \subseteq S_{i+1}$. $L \in \Sigma_{i+1}^{\mathsf{P}}$ if there is a polynomial time computable relation R such that

$$x \in L \iff \exists^P y_1 \forall^P y_2 \ldots Q^P y_{i+1} : R(x, y_1, \ldots, y_{i+1}) = 1.$$

Let $L' = \{\langle x, y\rangle \mid \forall^P y_2 \ldots Q^P y_{i+1} : R(x, y, y_2 \ldots, y_{i+1})\}$. By construction, $L' \in \Pi_i^{\mathsf{P}}$. By the induction hypothesis, $L' \in P_i$. The following Turing machine tests whether $x \in L$:

Input: x

1. Guess a y.
2. Query the oracle to check whether $\langle x, y\rangle \in L'$.

3. Accept if the answer is yes, otherwise reject.

This shows that $L \in \mathsf{NP}^{P_i}$. But $S_i = \mathsf{co}\text{-}P_i$. Thus $\mathsf{NP}^{S_i} = \mathsf{NP}^{P_i}$ and we have $L \in \mathsf{NP}^{S_i} = S_{i+1}$.

To show $S_{i+1} \subseteq \Sigma^{\mathsf{P}}_{i+1}$, let $L \in S_{i+1}$. Let M be a polynomial time nondeterministic Turing machine M that decides L with an oracle $W \in S_i$. Let M be t time bounded. W.l.o.g. we may assume that in each step M has at most two nondeterministic choices. Consider M on input x: Let $y \in \{0,1\}^{t(|x|)}$ be a string that describes the nondeterministic choices of M along some computation path. On such a path, M might query W at most $t(|x|)$ times. Let $a \in \{0,1\}^{t(|x|)}$ describe the answers to the queries.[1]

Given y and a, the following function f is polynomial time computable: $f(x,y,a,i)$ is the ith string that M on input x asks the oracle on the path given by y provided that the answers to previous oracle queries are as given by a. $f(x,y,a,i)$ is undefined (represented by some special symbol) if the oracle is asked fewer than i times. To compute $f(x,y,a,i)$, we just simulate M and make the nondeterministic choices as given by y and instead of really asking W, we pretend that the answer is as given by a. With the same simulation, we can also compute the following relation R given by $R(x,y,a) = 1$ iff M accepts x on the path given by y with oracle answers a. Now we have

$$x \in L \iff \exists^P \langle y,a\rangle [R(x,y,a) = 1 \wedge \bigwedge_{j:a_j=1} f(x,y,a,j) \in W \wedge \bigwedge_{j:a_j=0} f(x,y,a,j) \notin W]$$

where an expression involving an undefined $f(a,y,j)$ is always true. Note that the first part of the expression tests whether M has an accepting path and the second part verifies whether a contains the correct answers.

Each oracle answer check is either in $S_i = \Sigma^{\mathsf{P}}_i$ (positive answers) or in $P_i = \Pi^{\mathsf{P}}_i$ (negative answers). Thus we can replace each "$f(x,y,a,j) \in W$" and "$f(x,y,a,j) \notin W$" by a quantified expression with i quantifiers by the induction hypothesis. We can write all the quantifiers in front and combine quantifiers in such a way that we get a quantified expression for L with $i+1$ quantifiers starting with an existential quantifier. Thus $L \in \Sigma^{\mathsf{P}}_{i+1}$.

$\Pi^{\mathsf{P}}_{i+1} = P_{i+1}$ follow from the fact that both classes are the co-classes of $\Sigma^{\mathsf{P}}_{i+1}$ and S_{i+1}, respectively. ∎

[1] If M asks the oracle τ times, then the first τ bits of a are the answer.

10 P, NP, and PSPACE

The main result of this chapter to show that QBF is PSPACE-complete.

Exercise 10.1 *If L is* PSPACE*-hard under polynomial time many one reductions and $L \in$ NP, then* NP = PSPACE*. If $L \in$ P, then* P = PSPACE*.*

Theorem 10.1 QBF *is* PSPACE*-complete*

Proof. We first show that QBF is in PSPACE. We devise a recursive procedure: If F is a quantified Boolean formula without any quantifiers, we can evaluate it in polynomial space using the same procedure that we used to evaluate Boolean circuits. The problem is even easier, since there are no variables in the formula but only constants. Let $F = Qx_iF'$. We replace in F' every free occurrence of x_i by 0 and by 1, respectively. Let F'_0 and F'_1 be the resulting formulas. We now recursively evaluate F'_0 and F'_1. If Q is an $\exists$-quantifier, then F is true iff F'_0 is true or F'_1 is true. If Q is a $\forall$-quantifier, then F is true if F'_0 and F'_1 are true.

To implement this procedure, we need a stack whose size is linear in the number of quantifiers. Thus the total space requirement is surely polynomially bounded.

Next we show that QBF is PSPACE-hard. Let L be some language in PSPACE and let M be some polynomially space bounded deterministic Turing machine with $L(M) = L$. W.l.o.g. we may assume that M has only one work tape and no separate input tape. Furthermore, we may assume that M has a unique accepting configuration. We will encode configurations by bit strings. We encode the state by some fixed length binary representation and also the position of the head. (For the head position, the length is fixed for inputs of the same length.) Each symbol is represented by a binary string of fixed size, too. The configuration is encoded by concatenating all the bit strings above. Since the sizes of the concatenated strings are fixed, these encoding is an injective mapping. Let $p(n)$ be the length of the encoding on inputs of length n.

Let x be an input of length n. We will construct a formula F_x that is true iff $x \in L$. Let s_x denote the encoding of the start and t the encoding of the accepting configuration. Note that M can make at most $2^{p(n)}$ many steps for some polynomial p.

We will inductively construct a formula $F_j(X, Y)$. Here X and Y are disjoint sets of $p(n)$ distinct variables each. Let ξ and η be two encodings of configurations. $F_j(\xi, \eta)$ denotes the formula where we assign each variable

in X a bit from ξ and each variable in Y a bit from η. (Assume that the variables in X and Y are ordered.) We will construct F_j in such a way that $F(\xi, \eta)$ is true iff η can be reached from ξ by M with $\leq 2^j$ steps.

The inductions start is easy: $F_0(X, Y) = (X = Y) \vee S(X, Y)$. Here $X = Y$ denotes the formula that compares X and Y bit by bit and is true iff all bits are the same. $S(X, Y)$ is true if Y can be reached from X in one step (see the exercise below). The size of F_0 is polynomial in n.

For the induction step, the first thing that comes to mind is to mimic the proof of Savitch's theorem. We try

$$F_j(X, Y) = \exists Z : F_{j-1}(X, Z) \wedge F_{j-1}(Z, X).$$

("$\exists Z$" means that every Boolean variable in Z is quantified with an existential quantifier.) While this formula precisely describes what we want, its size is too big. It is easy to see that the size grows exponentially. Therefore, we exploit the following trick and use the formula F_j "twice":

$$F_j(X, Y) = \exists C \forall A \forall B : F_{j-1}(A, B) \vee (\neg(A = X \wedge B = C) \wedge \neg(A = C \wedge B = Y)).$$

Here $F_{j-1}(A, B)$ is used two times, namely if $A = X$ and $B = C$ or $A = C$ and $B = Y$. Therefore it checks whether X is reachable from Y within 2^j steps. However, its size now is only polynomial, since when going from F_{j-1} to F_j, we only get an additional additive increase of the formula size that is polynomial.

The final formula now is $Q_x = F_{p(n)}(s_x, t)$. By construction, Q_x is true iff $x \in L$. ∎

Exercise 10.2 *Construct a formula S such that $S(\xi, \eta)$ is true iff $\xi \vdash_M \eta$. Show that S has polynomial size and can be computed in polynomial time.*

Excursus: Games

Many games are PSPACE-hard, more precisely, given a board configuration, deciding whether this is a winning position for one player is PSPACE-hard (but not necessarily in PSPACE, since many games can go on for more than a polynomial number of steps).

To talk about complexity, we have to generalize the games to arbitrarily large boards. For Checkers or the ancient game Go, this is no problem. For Chess, one has to be a little creative: On a board of size $7n + 1$, one has e.g. 1 king, n queens, $2n$ bishops, $2n$ knights, $2n$ rooks, and $7n + 1$ pawns in each color.

While it looks at a first glance astonishing that many games are PSPACE-hard, it is rather natural. Being in a winning position means that for all moves of my opponent I have an answer such that for all moves of my opponent I have an answer ... which is a sequence of alternating quantifiers like in QBF.

Geography: Given a directed graph G with a start node s, two players construct a path by adding an edge to the front of the path until one player cannot add an edge anymore since this would reach a node already visited. Decide whether the first player has a winning strategy on G.

Checkers: Given a board position in a Checkers game, is this a winning position for white?

Go: Given a board position in a Go game, is this a winning position for white?

Chess: Given a board position in a (generalized) Chess game, is this a winning position for white?

All of the games are PSPACE-hard, Geography and some variants of Go are also in PSPACE.

In a symmetric game (i.e, both players can make the same moves and start in the same configuration) where a player can legally "pass" and in the case of a tie, the first player is declared to be the winner, the first player will always win. If the second player had a winning strategy, the first player could steal it by passing. Thus the board positions used for the hardness results above have to be rather exotic.

11 The Karp–Lipton Theorem

The class P is precisely the class of all languages that are decided by polynomial size uniform families of circuits. What happens if we drop the uniformity constraint? That is, we look at families of polynomial size circuits but do not care how to construct them. For reasons that will become clear later, call the resulting class P/poly. Is it then possible that NP $\subseteq$ P/poly? While not as strong as P = NP, this inclusion would have a huge impact: One (governments are a clear candidate) just takes a lot of resources and tries to find the polynomial size circuit for SAT with, say, 10000 variables. This would break any current crypto-system, for instance.

11.1 P/poly

We defined the class P/poly in terms of circuits. One can also define this class in terms of Turing machines that take advice.[1] Such a Turing machine has an additional read-only advice tape. On this tape, it gets an additional advice string, that only may depend on the length of the input.

Definition 11.1 *Let t and a be two functions $\mathbb{N} \to \mathbb{N}$. A language L is in the class $\mathsf{DTime}(t)/a$ if there is a deterministic Turing machine M with running time t and with an additional advice tape and a sequence of strings $\alpha(n) \in \{0,1\}^{a(n)}$ such that the following holds: For all $x \in L$, M accepts x with $\alpha(|x|)$ written on the advice tape. For all $x \notin L$, M rejects x with $\alpha(|x|)$ written on the advice tape. (The advice string is not counted as part of the input!)*

This definition extends to nondeterministic classes and space classes in the obvious way. We can also extend the definition to sets of functions T and A. We define $\mathsf{DTime}(T)/A = \bigcup_{t \in T, a \in A} \mathsf{DTime}(t)/a$. If we choose T and A both to be the class of all polynomials, then we get exactly P/poly.

Lemma 11.2 *For all languages $L \in \{0,1\}^*$, there is a polynomial time Turing machine M with polynomial advice accepting L iff $L \in$ P/poly.*

Proof. Let L be decided by a polynomial time Turing machine M with polynomial advice. Let x be an input of length n and let $\alpha(n)$ be the corresponding advice string. Consider $\alpha(n)$ as a part of the input of M. There is a polynomial size circuit C_n such that $C_n(x, \alpha(n)) = 1$ iff M with

[1] Sometimes, Turing machines are smarter than humans.

input x and advice $\alpha(n)$ accepts and this holds for all x of length n. Now choose C'_n to be the circuit that is obtained from C_n by considering the x as the input and treating the bits of $\alpha(n)$ as constants. C'_n is the desired circuit.

If L is decided by a nonuniform family of polynomial size circuits C_i, then the nth advice string $\alpha(n)$ is just an encoding of C_n. Circuit evaluation can be done in polynomial time. Thus, given an input x of length n, we just have to evaluate C_n at x. ∎

For each input length n, we give the Turing machine an advice $\alpha(n)$. Note that we do no restrict this sequence, except for the length. In particular, the sequence need not be to computable at all.

Lemma 11.3 *Any tally language $L \subseteq \{1\}^*$ is in $\mathsf{P/poly}$.*

Proof. For each input length n, we have an advice string of length one. This string is 1 if $1^n \in L$ and 0 otherwise. ∎

There are tally sets that are not computable, for instance

$$\{1^n \mid \text{the } n\text{th Turing machine halts on the empy word}\}.$$

11.2 The Karp–Lipton Theorem

The Karp–Lipton Theorem states that $\mathsf{NP} \subseteq \mathsf{P/poly}$ is not very likely, more precisely, this inclusion collapses the polynomial time hierarchy to the second level.

If $\mathsf{NP} \subseteq \mathsf{P/poly}$, then of course $\mathtt{SAT} \in \mathsf{P/poly}$. That is, there is a family C_i of polynomial size circuits for $\mathtt{SAT}$. C_i is a circuit that decides whether a given formula of length *exactly* i is satisfiable or not. But we can also assume that there is a family D_i of polynomial size circuits such that D_i decides whether a given formula of size *at most* i is satisfiable. D_i basically is "the union" of $C_1, \ldots, C_i$. Its size is again polynomial.

Lemma 11.4 *Let D_i be a family as described above. Then there is a polynomial time computable function h such that $h(F, D_{|F|})$ is a satisfying assignment for F iff F is a satisfiable formula.*

Proof. We use the downward self-reducibility for $\mathtt{SAT}$. Choose a variable in F and set it to zero and to one, respectively. Let F_0 and F_1 be the corresponding formulas. Their length is at most the length of F. Now evaluate $D_{|F|}$ to check whether F_0 or F_1 is satisfiable. If one of them is, say F_0, then we can construct a satisfying assignment for F by setting the chosen variable to zero and proceed recursively with F_0. ∎

Theorem 11.5 (Karp–Lipton) *If $\mathsf{NP} \subseteq \mathsf{P/poly}$, then $\Pi_2^{\mathsf{P}} \subseteq \Sigma_2^{\mathsf{P}}$.*

Proof. Let $A \in \Pi_2^{\mathsf{P}} = \forall\exists\mathsf{P} = \forall\mathsf{NP}$. Then there is a language $B \in \mathsf{NP}$ such that for all x,

$$x \in A \iff \forall^P y : \langle x, y\rangle \in B.$$

Since $B \in \mathsf{NP}$, there is a polynomial time many one reduction f from B to $\mathtt{SAT}$. In other words, for all x,

$$x \in B \iff f(x) \in \mathtt{SAT}.$$

Thus for all x,

$$x \in A \iff \forall^P y : f(\langle x, y\rangle) \in \mathtt{SAT}.$$

$f(z)$ is polynomially bounded for all z. Since y is polynomially bounded, too, $f(\langle x, y\rangle)$ is polynomially bounded in $|x|$.

Let D_i be a sequence of polynomial size circuits for $\mathtt{SAT}$ as constructed above. By Lemma 11.4, for all x,

$$x \in A \iff \forall^P y : h(f(\langle x, y\rangle), D_{|f(\langle x,y\rangle)|}) \text{ satisfies } f(\langle x, y\rangle),$$

where h is the function constructed in Lemma 11.4.

Note that we only know that the family D_i exists. It could be very hard to construct it. But we can use the power of the $\exists$ quantifier and guess the correct circuit! Since $f(\langle x, y\rangle)$ is polynomially bounded in $|x|$, the size of $D_{|f(\langle x,y\rangle)|}$ is bounded by $p(|x|)$ for some appropriate polynomial. Thus, for all x,

$$\begin{aligned} x \in A \iff &\exists^P D : D \text{ is a circuit with } |f(\langle x, y\rangle)| \text{ inputs} \\ &\forall^P y : h(f(\langle x, y\rangle), D) \text{ satisfies } f(\langle x, y\rangle). \end{aligned}$$

Note that we implicitly check whether D was the right guess since we verify that h produced a satisfying assignment. Even if this assignment is produced with a "wrong" circuit, we are still happy. Hence, $A \in \Sigma_2^{\mathsf{P}}$. ∎

Note that already $\Pi_2^{\mathsf{P}} \subseteq \Sigma_2^{\mathsf{P}}$ collapses the polynomial time hierarchy to the second level. We only showed it under the condition that $\Pi_2^{\mathsf{P}} = \Sigma_2^{\mathsf{P}}$. But $\Pi_2^{\mathsf{P}} \subseteq \Sigma_2^{\mathsf{P}}$ is sufficient to show that $\Sigma_2^{\mathsf{P}} = \Sigma_3^{\mathsf{P}}$ (apply an $\exists$). Then also $\Pi_2^{\mathsf{P}} = \Pi_3^{\mathsf{P}}$, since these are co-classes. From this, we get the collapse.

12 Relativization

In this chapter, we have a relativized look at the P versus NP question and related questions. More precisely, we will investigate the classes P^A and NP^A for some oracle A. It turns out, that there are oracles for which these two classes coincide and there are oracles for which these two classes are different. While this does not say much about the original question, it shows an important property that a proof technique should possess if it is capable to resolve the P versus NP question. This technique should not "relativize", i.e., it should not be able to resolve the question P^A versus NP^A for arbitrary A. The usual simulation techniques and diagonalization relativizes, in fact, almost all of the techniques from recursion theory relativize.

12.1 P and NP

Theorem 12.1 $\mathsf{P}^A = \mathsf{NP}^A$ *for all* PSPACE*-complete* A.

Proof. Let A be PSPACE-complete. Let L be in NP^A. Let M be a nondeterministic polynomially time bounded Turing machine with $L(M^A) = L$. M can ask only a polynomial number of questions to A on every branch. Each of them is only polynomially long. Thus $L \in \mathsf{PSPACE}^A$. But $\mathsf{PSPACE}^A = \mathsf{PSPACE}$, since instead of querying the oracle A, we can just simulate a polynomially space bounded Turing machine for A.

Since A is PSPACE-hard, there is a many one reduction from any language in PSPACE to A. In particular, $\mathsf{PSPACE} \subseteq \mathsf{P}^A$, since a many one reduction is also a Turing reduction. Putting everything together, we get

$$\mathsf{P}^A \subseteq \mathsf{NP}^A \subseteq \mathsf{PSPACE}^A = \mathsf{PSPACE} \subseteq \mathsf{P}^A. \qquad \blacksquare$$

For a language $B \in \{0,1\}^*$, let $\text{tally}(B) = \{\#^n \mid \text{there is a word of length } n \text{ in } B\}$. A nondeterministic Turing machine M with oracle B can easily decide $\text{tally}(B)$. On input $\#^n$, it just guesses a string $x \in \{0,1\}^n$ and verifies whether $x \in B$ by asking the oracle. If yes, it accepts; otherwise, it rejects.

More precisely: When M reads the input string, it can directly guess the string x on the oracle tape. This takes n steps. When it reads the first blank on the input tape, it enters the query state and gets the answer (one step). Then it just has to check whether the answer on the input tape is 0 or 1 (one step). Thus $\text{tally}(B) \in \mathsf{NTime}^B(n+2) \subseteq \mathsf{NP}^B$.

Lemma 12.2 *There is a* $B \in \mathsf{EXP}$ *such that* $\text{tally}(B) \notin DTime^B(2^n)$.

Proof. In a similar way like we have encoded Turing machines as binary strings, we can also encode oracle Turing machines. The encoding is essentially the same, we just have to mark the oracle tape and the query and answer state. By taking the lexicographic ordering on these strings, we get an ordering of the Turing machines.

We construct B iteratively. In the nth iteration, we add at most one string x_n of length n. B_n denotes the set that we have constructed in the first n iterations. We also have a set F of forbidden strings that shall not appear in B. This set is also constructed iteratively, F_n denotes the set that we have constructed in the first n iterations so far.

The nth iteration looks as follows. We simulate the nth Turing machine M_n with oracle B_{n-1} (with respect to the ordering defined above) on $\#^n$ for exactly 2^n steps. $M_n^{B_{n-1}}$ can query at most 2^{n-1} different strings, since for each query, we need one step to get the result and at least one step to write an oracle string. (In fact there are even fewer strings that the machine can query, since the oracle strings have to get longer and longer and the oracle tape is erased every time.) Let S_n be the set of strings that $M_n^{B_{n-1}}$ queries. Set $F_n = F_{n-1} \cup S_n$. If $M_n^{B_{n-1}}$ halts and rejects, we set $B_n = B_{n-1} \cup \{x_n\}$ for an arbitrary string in $\{0,1\}^n \setminus F_n$. Otherwise, we set $B_n = B_{n-1}$.

We have to check that the string x_n always exists. Since a Turing machine can query at most 2^{i-1} strings in 2^i steps,

$$|\bigcup_{1\leq i\leq n} S_i| \leq \sum_{i=1}^{n} 2^{i-1} < 2^n.$$

Hence x_n exists.

Next we show that $\text{tally}(B)$ is not accepted by a deterministic Turing machine with running time 2^n. Assume that M_n is such a machine. M_n^B behaves like $M_n^{B_{n-1}}$ on input $\#^n$, since all strings that are queried by M_n are in S_n. These strings are not in B by construction. Hence B_{n-1} and B do not contain any of the strings queried by M_n.

The assumption that M_n^B decides $\#^n \in \text{tally}(B)$ correctly within 2^n steps yields a contradiction: If $\#^n \notin \text{tally}(B)$, then $M_n^{B_{n-1}}$ would reject $\#^n$, but then B would contain a string of length n, namely x_n, and $\#^n \in \text{tally}(B)$, a contradiction. If $\#^n \in \text{tally}(B)$, then $M_n^{B_{n-1}}$ would accept, but then $B_n = B_{n-1}$ and B would not contain a string of length n which implies that $\#^n \notin \text{tally}(B)$, a contradiction.

Thus it remains to show that $B \in \mathsf{EXP}$.[1] To decide whether a given x of length n is in B we enumerate the first n Turing machines and simulate them on $\#^i$. In this way, we can compute B_n. Once we have B_n, we can decide whether $x \in B$, since in later iterations, only longer strings are added

[1] This statement is only needed to say something about the complexity of B. We want to find an "easy" oracle.

to B. The ith simulation needs time $2^{O(i)}$. One oracle query is simulated by a look up in the table for B_{i-1}. Thus the total running time is $2^{O(n)}$. ■

Note that the oracle B achieves the largest possible gap between determinism and nondeterminism: tally(B) is in linear time recognizable by a nondeterministic Turing machine with oracle B but cannot be accepted by a 2^n time bounded deterministic Turing machine.

Exercise 12.1 *Prove that* $\text{tally}(B) \in \mathsf{DTime}^B(2^{O(n)})$.

Theorem 12.3 *There is a language* $B \in \mathsf{EXP}$ *such that* $\mathsf{P}^B \neq \mathsf{NP}^B$.

Proof. Let B be the language from Lemma 12.2. We have $\text{tally}(B) \in \mathsf{NP}^B$ and $\text{tally}(B) \notin \mathsf{DTime}^B(2^n)$. But then $\text{tally}(B)$ cannot be in P^B. ■

Pitfalls

It is not clear how to speed up oracle computations by arbitrary constant factors, since this could mean asking two queries in one step.

12.2 PH and PSPACE

In this section we will show the first part of a clever construction that separates PH from PSPACE with respect to some oracle. Before we can do so, we first have to define what PH^B means! We can set $\mathsf{PH}^B = \bigcup_d (\Sigma_d^{\mathsf{P}})^B$, so it remains to define $(\Sigma_d^{\mathsf{P}})^B$. Let M be a polynomial time deterministic oracle Turing machine such that M with oracle B computes some $(d+1)$-ary relation $R^B(x, y_1, \ldots, y_d)$. Then the language L defined by

$$x \in L \iff \exists^P y_1 \forall^P y_2 \ldots Q y_d : R^B(x, y_1, \ldots, y_d) = 1$$

is in $(\Sigma_d^{\mathsf{P}})^B$.

The separation result for PH and PSPACE depends on a lower bound for the size of Boolean circuits with unbounded fanin. A Boolean circuit with unbounded fanin is a Boolean circuit that can have inner nodes of arbitrary fanin. These nodes are labeled with $\vee$ or $\wedge$ and compute the Boolean conjunction or disjunction of the inputs. The size of the circuit is the number of nodes in it plus an additional $d-2$ for every node of fanin $d > 2$. The lower bound that is needed for the separation result was shown using the polynomial method. Recall that PARITY is the set of all $x \in \{0,1\}^*$ that have an odd number of ones, i.e., the parity of x is 1.

Theorem 12.4 (Furst, Saxe & Sipser) *If for all constants $d, c \geq 1$, there does not exist any family of unbounded fanin Boolean circuits of depth d and size $2^{\mathrm{O}(\log^c n)}$ computing* PARITY*, then there is an oracle B such that $\mathsf{PH}^B \neq \mathsf{PSPACE}^B$.*

Proof overview: For a language $A \subseteq \{0,1\}^*$, let $A^{=n} := A \cap \{0,1\}^n$ be the subset of all strings of length n. For any set A let

$$\pi(A,n) = \begin{cases} 1 & \text{if } |A^{=n}| \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

In other words $\pi(A,n)$ is the parity of the number of words of length n in A. Let $P(A) = \{0^n \mid \pi(A,n) = 1\}$. We now view the characteristic vector a of all strings of length n of A as an input of length 2^n. Then $\texttt{PARITY}(a) = 1$ iff $0^n \in P(A)$. In this way, the oracle becomes the input of the circuit.

We will have the following mapping of concepts:

characteristic vectors of A	$\mapsto$	inputs of circuit
relation R	$\mapsto$	small subcircuits
quantifiers	$\mapsto$	unbounded fanin $\vee$ and $\wedge$

Proof. Let $P(A)$ be defined as above. We first show that there is a linear space bounded Turing machine M that given an oracle A, recognizes $P(A)$:

Input: $x \in \{0,1\}^*$

1. If $x \notin \{0\}^*$, then reject.
2. $c := 0$.
3. Enumerate all strings y in $\{0,1\}^{|x|}$, check whether $y \in A$, and if yes, then $c := c + 1 \mod 2$.
4. If $c = 1$ accept, else reject.

In particular, $P(A) \in \mathsf{PSPACE}^A$.

Let R be a $(d+1)$-ary relation. We call R good for length n if for all oracles A,

$$0^n \in P(A) \iff \exists^P y_1 \forall^P y_2 \ldots Q y_d : R^A(0^n, y_1, \ldots, y_d).$$

We now show that if a relation R is good for almost all lengths n (here used in the sense of "for all but finitely many") and can be computed in polynomial time, then we get a family of circuits that violates the assumption of the theorem.

Let N be a polynomial time bounded deterministic oracle Turing machine. We call N a fool if the following holds: For all n_0, for all $X \subseteq$

$\{0,1\}^{\leq n_0}$, and for infinitely many $n > n_0$, there exists a $Y \subseteq \{0,1\}^{>n_0}$ such that $N^{X \cup Y}$ errs on computing $R^{X \cup Y}(0^n, y_1, \ldots, y_d)$.

If N is not a fool, then we call N wise. For a wise N, there exists an n_0 and an X as above such that, for all but a finite number of $n > n_0$, $N^{X \cup Y}$ computes $R^{X \cup Y}(0^n, y_1, \ldots, y_d)$ correctly for all Y.

We can assume that N does not query strings of length $\ell \neq n$ and $\ell > n_0$. If N attempts to query such strings, then we can give it any value back since N has to answer correctly for all Y. Furthermore, there are only a finite number of strings of length at most n_0. Thus, instead of asking the oracle $X \cup Y$, the set X can be hard-wired into N. We will now show that wise N do not exist.

For fixed n, we can view the characteristic vector a of all strings of length n of A as an additional input of R; thus R becomes a function $r_n(y_1, \ldots, y_d, a) := R^A(0^n, y_1, \ldots, y_d)$.

Since N is $p(n)$ time bounded for some polynomial p, it can query the oracle at most $p(n)$ times for fixed n, $y_1, \ldots, y_d$. We now fix $y_1, \ldots, y_d$, and get a function $r_{n,y_1,\ldots,y_d}(a) := r_n(y_1, \ldots, y_d, a)$. The words that N queries may however depend on the answers to previous queries. We now simulate N on $0^n, y_1, \ldots, y_d$. Whenever N queries the oracle, we simulate it with possible answer 0 and with possible answer 1. We get a binary tree T of depth $p(n)$. The nodes are labeled with the corresponding strings queried and the edges are labeled with 0 or 1, the answer of the oracle to the query.

From this tree T, we get a formula $B_{n,y_1,\ldots,y_d}$ of depth two and size $\mathrm{O}(p(n) \cdot 2^{p(n)})$ that computes $r_{n,y_1,\ldots,y_d}$:

$$r_{n,y_1,\ldots,y_d}(a) = \bigvee_{\text{accepting paths } P \text{ of } T} a_{i_1^P}^{e_1^P} \wedge \cdots \wedge {}^{e_{p(n)}^P}_{i_{p(n)}^P}$$

In this disjunction, $i_1^P, \ldots, i_{p(n)}^P$ are the indices of the strings queried on the path P (i.e., the labels of the nodes along P) and $e_1^P, \ldots, e_{p(n)}^P$ are the answers on the path P (i.e., the labels of the edges along P). Above, we use the convention $x^1 = x$ and $x^0 = \neg x$ for a Boolean variable x.

$B_{n,y_1,\ldots,y_d}$ can be constructed by just simulating N for each possible outcome of the queries in space $p(n)$ (given $y_1, \ldots, y_d$).

Thus we get

$$0^n \in P(A) \iff \bigvee_{y_1} \bigwedge_{y_2} \bigvee_{y_3} \ldots B_{n,y_1,\ldots,y_d}(a).$$

But the righthand side is a circuit C_n of depth $d+2$ for PARITY. Its size is $\mathrm{O}(2^{dq(n)+p(n)})$ where q is a bound on the length of $y_1, \ldots, y_d$. It can be constructed in space $\mathrm{O}(dq(n) + p(n))$ which is logarithmic in the size. The number of inputs is $m := 2^n$. With respect to the number of inputs, the size of C_n is $2^{\mathrm{O}(\log^c m)}$ for some constant c.

Thus, if N is wise, then we can construct a family C_n that contradicts the assumption of the theorem. We only have circuits of input lengths 2^n but we can get a circuit for parity of any length ℓ out of it by rounding to the nearest power of 2 and then setting an appropriate number of inputs to 0.

Finally, we construct the oracle B separating PH^B from PSPACE^B by diagonalization. Every polynomial time Turing machine N_i that computes a potential relation R_i, is a fool. We now construct B inductively. For every candidate R_i, we choose a number $n_i > n_{i-1}$ that is greater than all previously chosen numbers such that, for all $X \in \{0,1\}^{\leq n_{i-1}}$, there exists a $Y \in \{0,1\}^{>n_{i-1}}$ such that N_i errs in computing $R^{X \cup Y}$ for length n_i. In particular, for $\bigcup_{\ell \leq n_{i-1}} B_i$, there exist Y_i and n_i with that property. We set $B_i = Y_i \cap \{0,1\}^{n_i}$ and $B = \bigcup_i B_i$. N_i with oracle B computes a relation that is not good for n_i, since we can always assume that N_i on input 0^n only queries strings of length n. Thus no R_i with oracle B is good for length n_i and hence, $P(B) \notin \mathsf{PH}^B$. ∎

Remark 12.5 *We can reduce the depth of C_n to $d+1$ by choosing B to be in CNF or DNF. We can then bring it down even to d by using the distributive laws and the fact that the gates at the bottom have fanin only $p(n)$.*

Excursus: Random oracle hypothesis

Not long after Theorem 12.3 was proven, Bennett and Gill showed that with respect to a random oracle A, i.e., every word x is in A with probability $1/2$, $\mathsf{P}^A \neq \mathsf{NP}^A$ with probability 1. This observation led to the random oracle hypothesis: Whenever a separation result is true with probability 1 with respect to a random oracle, then the unrelativized result should also be true. It was finally shown that $\mathsf{IP}^A \neq \mathsf{PSPACE}^A$ with probability 1 with respect to a random oracle. We will prove that $\mathsf{IP} = \mathsf{PSPACE}$ later on (and of course define IP).

---

13 Isomorphisms and sparse sets

13.1 Isomorphism conjecture

All NP-complete languages are closely related: Any of them can be reduced to any other. But something stronger seems to hold: Many NP-complete languages (in fact, all known NP-complete languages) are *polynomially isomorphic*.

Definition 13.1 *Two languages $A, B \subseteq \Sigma^\star$ are* polynomially isomorphic *if there exists a function $f : \Sigma^\star \to \Sigma^\star$ with the following properties:*

1. *f is bijective.*
2. *f is a reduction from A to B, i.e., $x \in A$ if and only if $f(x) \in B$ for all $x \in \Sigma^\star$.*
3. *f and f^{-1} (which is well-defined and a reduction from B to A) are computable in polynomial time.*

The function f is called a polynomial-time isomorphism.

Most reductions that we have seen so far lack the property of being bijective. A somewhat trivial example is the reduction from Clique to Independent Set, which maps a pair (G, k) to $(\overline{G}, k)$, where $\overline{G}$ is the complement of the graph G.

However, there is a way to obtain bijections, given that we have reductions from A to B and vice versa. As a first step, we make a reduction injective, length-increasing, and efficiently invertible.

Definition 13.2 *Let $L \subseteq \Sigma^\star$ be any language. We say that $p : (\Sigma^\star)^2 \to \Sigma^\star$ is a* padding function for L *if the following properties hold:*

1. *p is computable in logarithmic space.*
2. *For any $x, y \in \Sigma^\star$, $p(x, y) \in L$ if and only if $x \in L$.*
3. *For any $x, y \in \Sigma^\star$, $|p(x, y)| > |x| + |y|$.*
4. *Given $p(x, y)$, y can be computed in logarithmic space.*

A padding function p is thus essentially a length-increasing reduction from L to itself with the additional feature that p encodes another string y into the instance.

Consider Clique: Given a connected graph $G = (V, E)$ and an integer $k > 2$, is there a clique in G of size k? We construct $p(G, k, y)$ as follows: We attach to G at node 1 a long tree of nodes. The tree starts with a path of $|V| + |y|$ nodes. The first $|V|$ nodes do not have other nodes adjacent to them. The remaining $|y|$ nodes are either of degree three or four, having one or two leaves attached to them. If the $|V| + i$th node has degree three, then this corresponds to $y_i = 0$. If it has degree four, then this corresponds to $y_i = 1$.

Exercise 13.1 *Prove that the function described above is indeed a padding function.*

Exercise 13.2 *Give a padding functions for SAT.*

Padding functions can be used to make reductions injective, thus solving one of the issues we have with isomorphisms.

Lemma 13.3 *Let r be a reduction from A to B, and let p be a padding function for B. Then the function mapping x to $p(r(x), x)$ is a length-increasing injective reduction. Furthermore, there is a logarithmic space algorithm r^{-1} that, given $p(r(x), x)$, recovers x.*

Proof. The lemma follows immediately from the definition of padding functions. The mapping is a reduction due to item 1 and 2. It is length-increasing due to item 3: $|p(r(x), x)| > |r(x)| + |x| \geq |x|$. Finally, given $p(r(x), x)$, x can be recovered by item 4. ∎

Given two injective functions, one from A to B and $\Sigma^\star \setminus A$ to $\Sigma^\star \setminus B$ the other in the opposite direction, we can define a bijective function mapping A to B and $\Sigma^\star \setminus A$ to $\Sigma^\star \setminus B$, similar to the proof of the Cantor–Schröder–Bernstein theorem. However, we not only need the existence of such a function, we also need that the bijective function is computable in polynomial time. To achieve this, we exploit the fact that the injective reductions obtained from padding functions are length-increasing.

Theorem 13.4 *Let $A, B \subseteq \Sigma^\star$ be languages such that there are reductions r and s from A to B and B to A, respectively. Assume that r and s are injective, length-increasing, and logarithmic-space invertible. Then A and B are polynomially isomorphic.*

Proof. Since r is invertible, there exists r^{-1} that maps strings of B to shorter strings of A. The problem is that r^{-1} is a partial function. Analogously, s is invertible, and s^{-1} maps strings of A to shorter strings of B, but may not be total.

For any string $x \in \Sigma^\star$, let the s-chain of x be the following sequence: $(x, s^{-1}(x), r^{-1}(s^{-1}(x)), s^{-1}(r^{-1}(s^{-1}(x))), \ldots)$, i.e., s^{-1} and r^{-1} are applied alternately as long as possible. The s-chain ends at an element y if either $r^{-1}(y)$ or $s^{-1}(y)$ are undefined, whatever function should be applied next. Analogously are r-chains defined. The functions r and s are length-increasing. Thus, r^{-1} and s^{-1} are length-decreasing. Therefore, each s-chain and each r-chain of x is finite and consists of at most $|x| + 1$ elements.

Now we define a function f using the chains. First, we observe that every s-chain must end either at an element $s^{-1}(r^{-1}(\ldots s^{-1}(x)\ldots))$ or at an element $r^{-1}(s^{-1}(\ldots s^{-1}(x)\ldots))$. In the former case, which we call s-type, let $f(x) = s^{-1}(x)$. In the latter case, which we call r-type, let $f(x) = r(x)$. The function f is well-defined: If the s-chain consists of only a single element x, then it is of r-type. Hence $f(x) = r(x)$, which is always defined. In all other cases, $s^{-1}(x)$ is defined.

What remains to be done is to prove that f is indeed a polynomial-time isomorphism between A and B. We have to take care of all three properties of Definition 13.1.

1. *Injectivity:* Assume that $f(x) = f(y)$ for some $x \neq y$. Since both r and s^{-1} are injective (s^{-1} is injective on its domain), x and y must be in different cases in the definition of f. Without loss of generality, we assume that the s-chain of x is of s-type and the s-chain of y is of r-type. This means $f(x) = s^{-1}(x)$ and $f(y) = r(y)$, which implies $s^{-1}(x) = r(y)$. Thus, $y = r^{-1}(s^{-1}(x))$. This, however, yields that the s-chains of x and y are of the same type – a contradiction.

   Surjectivity: Let $y \in \Sigma^\star$ be arbitrary. Consider the r-chain of y. We distinguish two cases. First, assume that this chain is of s-type, i.e., it stops somewhere because r^{-1} is undefined. Then also the s-chain of $s(y)$ is of s-type. This yields $f(s(y)) = s^{-1}(s(y)) = y$, which concludes this case. Second, assume that the s-chain of y is of r-type, i.e., it stops somewhere since s^{-1} is undefined. Let $x = r^{-1}(y)$, which is well-defined in this case. Then $f(x) = r(r^{-1}(x)) = y$, which concludes the second case.

2. This follows immediately from the fact that r and s are reductions and how f is obtained from r and s.

3. The function value $f(x)$ can be computed by first computing the s-chain of x. This requires only $|x|$ calls of r^{-1} or s^{-1} on strings of length at most $|x|$.[1] This determines whether we should apply s^{-1} or r to x, both of which can be done in logarithmic time.

[1] This is the step that is hard to perform in logarithmic space. Although concatenations of logarithmic-space computable functions are still logarithmic-space computable, this might no longer be the case if we the number of functions that we concatenate is not bounded by some constant.

The following corollary follows immediately from the theorem and the exercises above.

Corollary 13.5 *Clique and SAT are polynomially isomorphic.*

One can (more or less easily) check that all known NP-complete sets are polynomially isomorphic. All we have to do is to give a padding function for an NP-complete problem. Then we can apply the machinery from above to prove that it is polynomially isomorphic to SAT and Clique and any other NP-complete problem with a padding function.

Berman and Hartmanis conjectured in 1976 that all NP-complete sets are polynomially isomorphic. This conjecture, known as the *Berman–Hartmanis isomorphism conjecture*, remains open. If it is true, then we immediately get $\mathsf{P} \neq \mathsf{NP}$: If $\mathsf{P} = \mathsf{NP}$, then every set L with $\emptyset \subsetneq L \subsetneq \Sigma^\star$ would be NP-complete. This includes finite sets, which cannot be isomorphic to infinite sets like SAT.

13.2 Density

The density of a language L is a function mapping n to the number of elements in L of length at most n:

$$\mathrm{dens}_L(n) = |\{x \in L \mid |x| \leq n\}|.$$

There is a direct relationship between isomorphism and density as stated by the following lemma.

Lemma 13.6 *If $A, B \subseteq \Sigma^\star$ are polynomially isomorphic, then $dens_A$ and $dens_B$ are polynomially related. This means that there exist polynomials p and q such that $dens_A(n) \leq dens_B(p(n))$ and $dens_B(n) \leq dens_A(q(n))$.*

Proof. The isomorphism maps all strings of A of length at most n to strings of B of length at most $p(n)$, where p is some polynomial. Since the isomorphism is in particular injective, this yields the first inequality. The second inequality is proved analogously. ∎

Clearly, $\mathrm{dens}_L(n) \leq |\Sigma|^{n+1}$. We distinguish two types of languages: sparse and dense languages. A language L is called *sparse* if there exists a polynomial p with $\mathrm{dens}_L(n) \leq p(n)$. A language L is called *dense* if L is not sparse.

If two languages A and B are polynomially isomorphic, they either both are sparse or both are dense, which follows from the lemma above.

Finite languages are sparse. If the isomorphism conjecture holds, then no finite language can be NP-complete, as we have argued above. Moreover, unary languages, i.e., subsets of $\{0\}^\star$ are sparse since their density is bounded by $n + 1$.

Since SAT is dense, no sparse language can be NP-complete if the isomorphism conjecture holds. Moreover and intuitively, sparse languages should not be NP-complete even without the isomorphism conjecture: To make a sparse language hard, the few elements must be well-hidden. This, however, would make reductions difficult.

In the following, we aim at proving that this intuition is correct even without the assumption of the isomorphism conjecture.

13.3 Unary languages

Before dealing with arbitrary sparse languages in the next section, we prove the unary languages are not NP-hard if $\mathsf{NP} \neq \mathsf{P}$.

Theorem 13.7 *If there exists a unary language $L \subseteq \{0\}^\star$ such that that L is* NP*-hard, then* $\mathsf{P} = \mathsf{NP}$.

Proof. Assume that there exists a unary language L with $\mathsf{SAT} \leq_{\mathrm{P}} L$, and let r be a reduction from SAT to L. We can assume that $r(x) \in \{0\}^\star$ for all x. (Otherwise, we can redefine r such that it outputs any fixed string in $\{0\}^\star \setminus L$ in this case. If no such string exists, then the theorem follows easily.) We construct a polynomial-time algorithm for SAT using r, the fact that L is unary, and self-reducibility of SAT.

Let ϕ be a formula with variables $x_1, \ldots, x_n$. For a string $t \in \{0,1\}^\ell$ ($\ell \leq n$), let ϕ_t be the formula obtained by setting $x_i = t_i$ for all $i \in \{1, \ldots, \ell\}$. (We remove satisfied clauses and false literals from each clause.) The formulas ϕ_t for $t \in \{0,1\}^{\leq n}$ form a binary tree, where ϕ_t has two children ϕ_{t0} and ϕ_{t1} if $|t| < n$. If $|t| = n$, then either ϕ_t is empty, which corresponds to true, or ϕ_t contains an empty clause, which corresponds to false.

The following algorithm determines if ϕ is satisfiable on input ϕ and $t = \varepsilon$:

1. If $|t| = n$, then return "yes" if ϕ_t contains no clauses and "no" if ϕ_t contains an empty clause.

2. Return "yes" if ϕ_{t0} or ϕ_{t1} returns "yes", otherwise return "no".

This algorithm without any modification requires exponential time. But we have not exploited r yet: If $r(\phi_t) = r(\phi_{t'})$ for some t and t', then either both ϕ_t and $\phi_{t'}$ return "yes" or both return "no". (We do not have to decide if $r(\phi_t) \in L$ or not. This might not even be possible since L is only assumed to be NP-hard. L might not even be recursive. Note that we will never try membership testing with L, we only use the reduction r.)

We add a "hash table" H to the algorithm that keeps track of the values $r(\phi_t)$ that we have already seen. This yields the following modified algorithm:

1. If $|t| = n$, then return "yes" if ϕ_t contains no clauses and "no" if ϕ_t contains an empty clause.

2. Let $R = r(\phi_t)$. If $(R, v) \in H$, then return v.

3. Otherwise, return "yes" if ϕ_{t0} or ϕ_{t1} returns "yes", otherwise return "no". Update H by adding $(r(\phi_t), v)$ for $v \in \{\text{"yes"}, \text{"no"}\}$ appropriately chosen.

The algorithm decides SAT correctly since r is a reduction from SAT. It remains to analyze its running-time.

Since r is a polynomial-time reduction, the lengths of the images $r(\phi_t)$ are bounded by some polynomial p. Thus, the table H requires only polynomial space. Every execution of the algorithm requires $O(p(|\phi|))$ time for lookup in H, if we disregard the recursive calls. If we can bound the number of recursive calls by a polynomial, then we are done.

The binary tree of the recursive calls has n levels. In every level i, there are at most $p(n)$ different labels t of length $n - i$ such that two recursive calls on ϕ_{t0} and ϕ_{t1} are made. On all other incarnations with labels t of length $n - i$, we have $r(\phi_t)$ already in the hash table. Thus, overall at most $O(np(n))$ recursive calls are necessary, which yields a running-time bound of $O(np(n)^2)$. ■

Note, as mentioned in the proof, that we do not need any property of the unary language L itself. We only exploit the reduction as a hash function, not knowing if the hash values are in L or not. The same applies to Fortune's theorem (Theorem 13.8 of the next section), which states that the existence of sparse co-NP languages implies NP = P. Mahaney's theorem (Theorem 13.9), however, which states that sparse languages are unlikely to be NP-complete, needs the property that the sparse language is contained in NP.

13.4 Mahaney's theorem

Having proved that no unary language is NP-complete unless P = NP, the question arises if the same holds for arbitrary sparse languages without the isomorphism conjecture. The goal of this section is to prove that, indeed, no sparse set is NP-complete or co-NP-hard. We will start by proving Fortune's theorem, which states the latter. It is a little easier to prove.

Theorem 13.8 (Fortune) *If there exists a sparse set that is* co-NP*-hard, then* P = NP*.*

Proof. We assume that the encoding of a formula of n variables has a length of at least n. Furthermore, if we substitute a variable by 0 or 1, the

length of the encoding of the resulting formula is no greater than the length of the encoding of the original formula.

Assume that L is a sparse set and $\mathrm{dens}_L(n) \leq n^k$ for some k. Assume further that L is co-NP-hard. This yields $\overline{\mathsf{SAT}} \leq_{\mathrm{P}} L$. Let r be a polynomial-time reduction from $\overline{\mathsf{SAT}}$ to L that is computable in time n^c for some c.

For some n, let L^n be the set of all elements of L of length at most n^c. This means that L^n contains all possible images of unsatisfiable formulas of length n under r. We have $|L^n| \leq n^{ck} = N$.

Given this setting, we prove that there is a polynomial-time algorithm for SAT by self-reducibility. Let ϕ be a formula with n variables. As above, for $|t| \leq n$, we denote by ϕ_t the formula with x_i substituted by t_i. We consider the binary tree with $\phi = \phi_\varepsilon$ as the root. Node ϕ_t has children ϕ_{t0} and ϕ_{t1}. Let T_m be the set of nodes at level m. T_0 contains only the root.

Building the full tree would obviously require exponential time. We will now show how to prune the tree in order to get a polynomial time algorithm. To do this, we construct subsets $T'_m \subseteq T_m$ with the following properties:

1. $|T'_m| \leq N$ for all $m \in \{0, 1, \ldots, n\}$.

2. If T'_m is actually constructed, then T'_m contains a satisfiable formula if and only if ϕ is satisfiable. (It might happen that the algorithm terminates before reaching level m. In this case, constructing T'_m is unnecessary.)

We set $T'_0 = T_0$. Assume that we are given T'_m. Then, for every $\phi_t \in T'_m$, we compute $r(\phi_{t0})$ and $r(\phi_{t1})$ using the reduction to L. Let $S_{m+1} = \{r(\phi_{t0}), r(\phi_{t1}) \mid \phi_t \in T'_m\}$. We distinguish two cases.

First, if $|S_{m+1}| > N$, then there exists a $u \in \{0,1\}^{m+1}$ such that $r(\phi_u) \notin L$ since L is sparse. This implies that ϕ_u is satisfiable because r is a reduction. But if some ϕ_u is satisfiable, then also ϕ is satisfiable. We terminate the algorithm, and we construct no T'_ℓ for $\ell > m$.

Second, if $|S_{m+1}| \leq N$, then, for every $y \in S_{m+1}$, we select one $u \in \{0,1\}^{m+1}$ with $r(\phi_u) = y$, and we put ϕ_u into T'_{m+1}.

By construction, T'_m is either not constructed or $|T'_m|$ is bounded by N. (Otherwise, the algorithm terminates as argued above.) It remains to prove the second property. If ϕ is not satisfiable, then no formula in any T_m is. Thus, no formula in any $T'_m \subseteq T_m$ is satisfiable. If ϕ is satisfiable, then we prove that every T'_m contains a satisfiable formula by induction on m. For $m = 0$, this is clear since $T'_0 = T_0 = \{\phi_\varepsilon\} = \{\phi\}$. Assume that the property holds for m. If ϕ is satisfiable, then T'_m contains a satisfiable formula ϕ_t. Then ϕ_{t0} or ϕ_{t1} are satisfiable. Assume without loss of generality that ϕ_{t0} is satisfiable. Then all formulas in the set $M_{t0} = \{\psi \mid r(\psi) = r(\phi_{t0})\}$ are satisfiable. Since $r(\phi_{t0}) \in S_{m+1}$, we put one formula $\phi_u \in M_{t0}$ into T'_{m+1}, which completes the induction.

We now have a polynomial-time algorithm for SAT: Compute T'_m for all m, unless either $m = n$ or the algorithm terminates earlier because $|S_m| > N$ for some m. In the latter case, the formula ϕ is satisfiable. In the former, we have to evaluate at most N formulas involving only constants. If one of them evaluates to true, then ϕ is satisfiable, otherwise, ϕ is not satisfiable. ∎

Note that, throughout this proof, we made no assumption how difficult the sparse language actually is. All that we used is the existence of a polynomial-time many-one reduction. For Mahaney's theorem (that no sparse language is NP-complete unless P = NP), we not only need the NP-hardness of a sparse language, but the language also has to be in NP. The proof of Mahaney's theorem builds on Fortune's theorem and is slightly more difficult. It uses a *census function* similar to the proof of the Immerman–Szelepcsényi theorem.

Theorem 13.9 (Mahaney) *If there exists a sparse* NP*-complete set, then* P = NP*.*

Proof. Let L be a sparse set. Assume that there exists a polynomial-time many-one reduction r from SAT to L with time bound n^c. We will use this reduction to devise a polynomial-time algorithm for SAT.

First, to get the idea, let us assume that we know precisely how many elements of length at most n the set L contains, i.e., we assume that the function dens_L is polynomial-time computable. We can write $\overline{L}$ as

$$\overline{L} = \left\{ x \mid \exists y_1 \ldots \exists y_{\mathrm{dens}_L(|n|)} : \bigwedge_i \big(|y_i| \leq |x|^c \wedge y_i \neq x \wedge y_i \in L\big) \wedge \bigwedge_{i \neq j} y_i \neq y_j \right\}.$$

Since L is in NP, this shows that $\overline{L}$ is also in NP. Since L is NP-complete, we have $\overline{L} \leq_{\mathrm{P}} L$. But then also $L \leq_{\mathrm{P}} \overline{L}$. Thus, L is also co-NP-complete. This allows us to apply Fortune's theorem to get the desired result.

The issue is that we do not know dens_L. (This was also the main issue with the Immerman–Szelepcsényi theorem.) However, we know that $\mathrm{dens}_L(n) \leq n^k$ for some k. We try the construction for all possibilities $m \in \{0, 1, \ldots, n^k\}$ for the value of $\mathrm{dens}_L(n)$. For $m \neq \mathrm{dens}_L(n)$, we might get garbage results. But if $m = \mathrm{dens}_L(n)$, we get a satisfying assignment for ϕ if ϕ is indeed satisfiable. If ϕ is unsatisfiable, then we will never get a satisfying assignment for ϕ.

Although we might never know which value of m was correct, we know that one must be correct. But if we happen to find a satisfying assignment, we do not have to care whether m was wrong or right.

Let us carry this out more formally. Let us define a partial complement of L:

$$\hat{L} = \left\{ (x,m) \mid \exists y_1 \ldots \exists y_m : \bigwedge_i \big(|y_i| \leq |x|^c \wedge y_i \neq x \wedge y_i \in L\big) \wedge \bigwedge_{i \neq j} y_i \neq y_j \right\}.$$

If $m < \mathrm{dens}_L(|x|)$, then $(x,m) \in \hat{L}$. If $m = \mathrm{dens}_L(|x|)$, then $(x,m) \in \hat{L}$ if and only if $x \notin L$. If $m > \mathrm{dens}_L(|x|)$, then $(x,m) \notin \hat{L}$. This allows us to rewrite $\hat{L}$ as

$$\hat{L} = \big\{ (x,m) \mid m < \mathrm{dens}_L(|x|) \vee \big(m = \mathrm{dens}_L(|x|\big) \wedge x \notin L \big\}.$$

The partial complement $\hat{L}$ of L is in NP. Therefore, there exists a reduction s from $\hat{L}$ to L. We compose s and r to get a family of maps $\phi \mapsto s(r(\phi), m)$, one for each value for m. This gives us

$$\begin{aligned} s(r(\phi), \mathrm{dens}_L(|r(\phi)|)) \in L &\Leftrightarrow (r(\phi), \mathrm{dens}_L(|r(\phi)|)) \in \hat{L} \\ &\Leftrightarrow r(\phi) \notin L \\ &\Leftrightarrow \phi \in \overline{\mathsf{SAT}}. \end{aligned}$$

Now we try the construction of Theorem 13.8 for all possible m (which yield different maps $\phi \mapsto s(r(\phi), m)$) up to n^{kc}. In this way, we are certain that we hit the right value of m. The construction of Theorem 13.8 then gives us a satisfying assignment if one exists. ■

14 Probabilistic computations

14.1 Randomized complexity classes

Probabilistic Turing machines have an additional *random tape*. On this tape, the Turing machine gets an one-sided infinite $\{0,1\}$ string y. The random tape is read-only and one-way.

Right at the moment, we are considering the random string y as an additional input. The name random string is justified by the following definition: A probabilistic Turing machine accepts an input x with *acceptance probability* at least p if $\Pr[M(x,y)=1] \geq p$. Here the probability is taken over all choices of y. We define the *rejection probability* in the same way. The running time $t(n)$ of a probabilistic Turing machine M is the maximum number of steps that M performs on any input of length n and and any random string y. Note that if $t(n)$ is bounded, then we can consider y to be a finite string of length at most $t(n)$. The maximum number of random bits a Turing machine reads on any input x of length n and random string y is called the amount of randomness used by the machine.

We define $\mathsf{RTime}(t(n))$ to be the class of all languages L such that there is a Turing machine M with running time $t(n)$ and for all $x \in L$, M accepts x with probability at least $1/2$ and for all $x \notin L$, M rejects L with probability 1. Such an M is said to have a *one-sided error*. If M in fact accepts each $x \in L$ with probability $\geq 1-\epsilon \geq 1/2$, then we say that the error probability of M is bounded by ϵ.

The class $\mathsf{BPTime}(t(n))$ is defined in the same manner, but we allow the Turing machine M to err in two ways. We require that for all $x \in L$, M accepts x with probability at least $2/3$ and for all $x \notin L$, M rejects with probability at least $2/3$ (that is, accepts with probability at most $1/3$). Such an error is called a *two-sided error*. If M actually accepts all $x \in L$ with probability $\geq 1-\epsilon$ and accepts each $x \notin L$ with probability $\leq \epsilon$, then we say that the error probability is bounded by ϵ.

Definition 14.1 *1.* $\mathsf{RP} = \bigcup_{i\in\mathbb{N}} \mathsf{RTime}(n^i)$,

2. $\mathsf{BPP} = \bigcup_{i\in\mathbb{N}} \mathsf{BPTime}(n^i)$,

3. $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{co\text{-}RP}$.

The name ZPP stands for *zero error probabilistic polynomial time*. It is justified by the following statement.

Exercise 14.1 *A language L is in* ZPP *if and only if L is accepted by a probabilistic Turing machine with error probability zero and expected polynomial running time. Here the expectation is taken over all possible random strings on the random tape.*

For robust classes (such as RP and BPP) the choice of the constants $1/2$ and $2/3$ in the definitions of RTime and BPTime is fairly arbitrary, since both classes allow *probability amplification.*

Lemma 14.2 *Let M be a Turing machine for some language $L \in$ RP that runs in time $t(n)$, uses $r(n)$ random bits, and has error probability ϵ. For any $k \in \mathbb{N}$, there is a Turing machine M' for L that runs in time $O(kt(n))$, uses $kr(n)$ random bits, and has error probability ϵ^k.*

Proof. M' works as follows:

Input: $x \in \{0,1\}^*$

1. M' simulates M k times, each time using new random bits.
2. M' accepts, if in at least one of the simulations, M accepts. Otherwise, M' rejects.

The bounds on the time and randomness are obvious. If $x \notin L$, then M' also rejects, since M does not err on x. If $x \in L$, then with probability at most ϵ, M rejects x. Since M' performs k independent trials, the probability that M' rejects x is at most ϵ^k. ∎

Lemma 14.3 *Let M be a Turing machine for some language $L \in$ BPP that runs in time $t(n)$, uses $r(n)$ random bits, and has error probability $\epsilon < 1/2$. For any $k \in \mathbb{N}$, there is a Turing machine M' for L that runs in time $O(kt(n))$, uses $kr(n)$ random bits, and has error probability $2^{-c_\epsilon k}$ for some constant c_ϵ that solely depends on ϵ.*

Proof. M' works as follows:

Input: $x \in \{0,1\}^*$

1. M'simulates M k times, each time with fresh random bits.
2. M' accepts, if in at least half of the simulations (rounded up), M accepts. Otherwise, M' rejects.

Let μ be the expected number of times that a simulated run of M accepts. If $x \in L$, then $\mu \geq (1-\epsilon)k$. The probability that less than half of the simulated runs of M accept is $< e^{-\frac{(1-\epsilon)\delta^2}{2}k}$ with $\delta = 1 - \frac{1}{2(1-\epsilon)}$ by the

Chernoff bound (see below). The case $x \notin L$ is treated similarly. In both cases, the error probability is bounded by 2^{ck} for some constant c only depending on ϵ. ■

Remark 14.4 *In both lemmas, k can also be a function in n, as long as k is computable in time $O(k(n)t(n))$. (All reasonable functions k are.)*

In the proof above, we used the so-called *Chernoff bound*. A proof of it can be found in most books on probability theory.

Lemma 14.5 (Chernoff bound) *Let $X_1, \ldots, X_m$ be independent 0–1 valued random variables and let $X = X_1 + \cdots + X_m$. Let $\mu = E(X)$. Then for any $\delta > 0$,*

$$\Pr[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \quad \text{and} \quad \Pr[X < (1-\delta)\mu] < e^{-\frac{\mu\delta^2}{2}}.$$

We have

$$\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \begin{matrix}\mathsf{RP} \\ \mathsf{co\text{-}RP}\end{matrix} \subseteq \mathsf{BPP}. \tag{14.1}$$

The latter two inclusions follow by amplifying the acceptance probability once.

Exercise 14.2 *Let PP be the class of all languages L such that there is a polynomial time probabilistic Turing machine M that accepts all $x \in L$ with probability $\geq 1/2$ and accepts all $x \notin L$ with probability $< 1/2$. Show that $\mathsf{NP} \subseteq \mathsf{PP}$.*

14.2 Relation to other classes

We start with comparing RP and BPP with non-randomized complexity classes. The results in this section are the basic ones, further results will be treated in the lecture "Pseudorandomness and Derandomization" in the next semester.

Theorem 14.6 $\mathsf{BPP} \subseteq \mathsf{PSPACE}$.

Proof. Let M be polynomial time bounded Turing machine with error probability bounded by $1/3$ for some $L \in \mathsf{BPP}$. Assume that M reads at most $r(n)$ random bits on inputs of length n. Turing machine M' simulates M as follows:

Input: $x \in \{0,1\}^*$

1. M' systematically lists all bit strings of length $r(n)$.

2. M' simulates M with the current string as random string.
3. M' counts how often M accepts and rejects.
4. If the number of accepting computations exceeds the number of rejecting computations, M' accepts. Otherwise, M' rejects.

Since M is polynomial time, $r(n)$ is bounded by a polynomial. Hence M' uses only polynomial space. ■

Corollary 14.7 $\mathsf{BPP} \subseteq \mathsf{EXP}$.

Theorem 14.8 $\mathsf{RP} \subseteq \mathsf{NP}$.

Proof. Let M be a polynomial time probabilistic Turing machine with error probability bounded by $1/2$ for some $L \in \mathsf{RP}$. We convert M into a nondeterministic machine M' as follows: Whenever M would read a bit from the random tape, M' nondeterministically branches to the two states that M would enter after reading zero or one, respectively.

If M does not accept x, then there is no random string such that M on input x reaches an accepting configuration. Thus there is no accepting path in the computation tree of M' either.

On the other hand, if M accepts x, then M reaches an accepting configuration on at least half of the random strings. Thus at least half of the paths in the computation tree of M' are accepting ones. In particular, there is at least one accepting path. Hence M' accepts x. ■

NP and RP

NP: *one* proof/witness of membership
RP: *many* proofs/witnesses of membership

Next we turn to the relation between BPP and circuits.

Theorem 14.9 (Adleman) $\mathsf{BPP} \subseteq \mathsf{P/poly}$.

Proof. Let $L \in \mathsf{BPP}$. By Lemma 14.3, there is a polynomial time bounded probabilistic Turing machine with error probability $< 2^{-n}$ that accepts L. There are 2^n possible input strings of length n. Since for each string x of length n, the error probability of M is $< 2^{-n}$, M can err on x only for a fraction of all possible random strings that is smaller than 2^{-n}. Thus there must be one random string that is good for all inputs of length n. We take this string as an advice string for the inputs of length n. By Lemma 11.2, $L \in \mathsf{P/poly}$. ■

How do we find this good random string? If we amplify the error probability even further, say to 2^{-2n}, then almost all, namely a fraction of $1-2^{-n}$ random strings are good. Thus picking the advice at random is a good strategy. (This, however, requires randomness!)

14.3 Further exercises

Exercise 14.3 *Show the following: If* $\mathtt{SAT} \in \mathsf{BPP}$*, then* $\mathtt{SAT} \in \mathsf{RP}$*. (Hint: downward self-reducibility).*

The answer to the following question is not known.

Open Problem 14.10 *Prove or disprove:* $\mathsf{RP} = \mathsf{P}$ *implies* $\mathsf{BPP} = \mathsf{P}$.

15 The BP-operator

BPP extends P by adding randomness to the computation but all other properties of P stay the same. In this section, we introduce a general mechanism of adding randomness to a complexity class.

Definition 15.1 *Let* C *be a class of languages. The class* BP·C *is the class of all languages A such that there is a $B \in$ C, a polynomial p, and constants $\alpha \in (0,1)$ and $\beta > 0$ such that for all inputs x:*

$$x \in A \Longrightarrow \Pr_{y \in \{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \geq \alpha + \beta/2,$$
$$x \notin A \Longrightarrow \Pr_{y \in \{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \leq \alpha - \beta/2,$$

α is called the *threshold value* and β is the *probability gap*. Note the similarity to the $\exists$- and $\forall$-operators.

Exercise 15.1 *Prove that* BPP = BP·P.

15.1 Probability amplification

Definition 15.2 *Let* C *be a class of languages.* BP·C allows probability amplification *if for every $A \in$ BP·C and every polynomial q, there is a language $B \in$ C and a polynomial p such that for all inputs x:*

$$x \in A \Longrightarrow \Pr_{y \in \{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \geq 1 - 2^{-q(|x|)}$$
$$x \notin A \Longrightarrow \Pr_{y \in \{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \leq 2^{-q(|x|)}$$

We would like to be able to perform probability amplification. For BPP, we ran the Turing machine several times and took a majority vote. Now the key point is that the resulting machine is again a probabilistic polynomial time machine.

A sufficient condition for a class C to allow probability amplification is that for all $L \in$ C, $\mathsf{P}^L \in$ C, i.e., C is closed under Turing reductions. But this is a too strong restriction, since it is not clear whether for instances NP is closed under Turing reductions. However, a weaker condition also suffices.

Definition 15.3 *1. A* positive polynomial time Turing reduction *from a language A to a language B is a polynomial time Turing machine R*

such that $A = L(R^B)$ and R has the following monotonicity property: if $D \subseteq D'$ then $L(R^D) \subseteq L(R^{D'})$ for all languages D, D'. We write $A \leq_{\mathrm{P}}^{\mathrm{T}+} B$ in this case.

2. *Let B be a language and C be a class of languages. We define $\mathsf{Pos}(B) = \{A \mid A \leq_{\mathrm{P}}^{\mathrm{T}+} B\}$ and $\mathsf{Pos}(\mathsf{C}) = \{A \mid A \leq_{\mathrm{P}}^{\mathrm{T}+} B$ for some $B \in \mathsf{C}\}$.*

Exercise 15.2 *Prove the following:* $\mathsf{Pos}(\mathsf{NP}) = \mathsf{NP}$.

Lemma 15.4 *Let C be closed under positive polynomial time Turing reductions. Then $\mathsf{BP}\cdot\mathsf{C}$ allows probability amplification.*

Proof. Let $A \in \mathsf{BP}\cdot\mathsf{C}$ and let $D \in \mathsf{C}$, $\alpha \in (0,1)$, and $\beta > 0$ such that the conditions in Definition 15.1 holds. Let q be a polynomial.

Let $k = k(|x|)$ be some integer to be chosen later. The language B now consists of all $\langle x, y_1, \ldots, y_k\rangle$ such that for more than αk indices i, $\langle x, y_i\rangle \in D$. (I.e., the y_κ serve as fresh random strings for k independent trials. We then take a majority vote.)

Since there is a positive Turing reduction from B to D (if $D \subseteq D'$, then $\langle x, y_i\rangle \in D$ of course implies $\langle x, y_i\rangle \in D'$), $B \in \mathsf{C}$. If we now choose $k = \mathrm{O}(q(|x|))$, then the error probability goes down to $2^{-q(|x|)}$ (as already seen for BPP). ∎

Exercise 15.3 *Show the following: If $\mathsf{BP}\cdot\mathsf{C}$ allows probability amplification, then $\mathsf{BP}\cdot\mathsf{BP}\cdot\mathsf{C} = \mathsf{BP}\cdot\mathsf{C}$.*

15.2 Operator swapping

Lemma 15.5 *If C is closed under Pos, then*

1. $\exists\,\mathsf{BP}\cdot\mathsf{C} \subseteq \mathsf{BP}\cdot\exists\mathsf{C}$,
2. $\forall\,\mathsf{BP}\cdot\mathsf{C} \subseteq \mathsf{BP}\cdot\forall\mathsf{C}$,

Proof. Let $A \in \exists\,\mathsf{BP}\cdot\mathsf{C}$. By assumption, there is a language $B \in \mathsf{BP}\cdot\mathsf{C}$ such that for all x,

$$x \in A \iff \exists^P b : \langle x, b\rangle \in B. \tag{15.1}$$

In particular, $x \in A$ depends only on $B^{\leq p(|x|)}$, the strings in B of length at most $p(|x|)$ for some polynomial p.

Since $\mathsf{BP}\cdot\mathsf{C}$ allows probability amplification, for every polynomial q, there is a language $D \in \mathsf{C}$ and a polynomial r such that for all inputs y:

$$y \in B \Longrightarrow \Pr_{z\in\{0,1\}^{r(|y|)}}[\langle y, z\rangle \in D] \geq 1 - 2^{-q(|y|)},$$
$$y \notin B \Longrightarrow \Pr_{z\in\{0,1\}^{r(|y|)}}[\langle y, z\rangle \in D] \leq 2^{-q(|y|)}.$$

Set $q(n) = 2n + 3$. It follows that for all n:

$$\begin{aligned}\Pr_z[\text{for all } y \text{ with } |y| \le p(n)\text{: } y \in B \iff \langle y, z\rangle \in D] &\ge 1 - \sum_{\nu=0}^{p(n)} 2^{-q(\nu)} \cdot 2^{\nu} \\ &\ge 1 - 1/4 \\ &= 3/4,\end{aligned}$$

by the union bound. For a string z, let $Y(z) := \{y \mid \langle y, z\rangle \in D\}$. Thus for all n

$$\Pr_z[B^{\le p(n)} = Y(z)^{\le p(n)}] \ge 3/4.$$

This means that for a random z, $Y(z)^{\le p(n)}$ behaves like $B^{\le p(n)}$ with high probability. Finally set

$$\begin{aligned}E &= \{\langle x, z\rangle \mid \exists^P b\ \langle x, b\rangle \in Y(z)^{\le p(|x|)})\} \\ &= \{\langle x, z\rangle \mid \exists^P b\ \langle \langle x, b\rangle, z\rangle \in D \wedge |\langle x, b\rangle| \le p(|x|)\}.\end{aligned}$$

Since $D \in \mathsf{C}$, $E \in \exists\mathsf{C}$, as C is closed under Pos. (To test the predicate, we first check whether $|\langle x, b\rangle| \le p(|x|)$, and if it is, we then query D.) This means that in (15.1), we can replace the righthand side by

"for a fraction of $\ge 3/4$ of all z, $\langle x, z\rangle \in E$".

Thus $A \in \mathsf{BP}\text{-}\exists\mathsf{C}$.

The case of $\forall$ is shown in exactly the same way. ∎

Exercise 15.4 *Show that if* $\mathsf{NP} \subseteq \mathsf{BPP}$*, then* $\Sigma_2^{\mathsf{P}} \subseteq \mathsf{BPP}$ *(and even* $\mathsf{PH} \subseteq \mathsf{BPP}$*).*

15.3 BPP and the polynomial hierarchy

For two strings $u, v \in \{0,1\}^m$, $u \oplus v$ denotes the string that is obtained by taking the bitwise XOR.

Lemma 15.6 (Lautemann) *Let* $n > \log m$*. Let* $S \subseteq \{0,1\}^m$ *with* $|S| \ge (1 - 2^{-n})2^m$*.*

1. *There are* $u_1, \ldots, u_m$ *such that for all* v*,* $u_i \oplus v \in S$ *for some* $1 \le i \le m$*.*
2. *For all* $u_1, \ldots, u_m$ *there is a* v *such that* $u_i \oplus v \in S$ *for all* $1 \le i \le m$*.*

Proof.

1. We have

$$\begin{aligned}
&\Pr_{u_1,\ldots,u_m\in\{0,1\}^m}[\exists v : u_1 \oplus v, \ldots, u_m \oplus v \notin S] \\
&\leq \sum_{v\in\{0,1\}^m} \Pr_{u_1,\ldots,u_m\in\{0,1\}^m}[u_1 \oplus v, \ldots, u_m \oplus v \notin S] \\
&= \sum_{v\in\{0,1\}^m} \prod_{i=1}^{m} \Pr_{u_i\in\{0,1\}^m}[u_i \oplus v \notin S] \\
&\leq 2^m \cdot (2^{-n})^m \\
&< 1,
\end{aligned}$$

   since $u_i \oplus v$ distributed uniformly in $\{0,1\}^m$ and all the u_i's are drawn independently. Since the probability that a v with the desired properties does not exist is < 1, there must be a v that fulfills the assertions of the first claim.

2. Fix $u_1, \ldots, u_m$. We have

$$\begin{aligned}
\Pr_{v\in\{0,1\}^m}[\exists i : u_i \oplus v \notin S] &\leq \sum_{i=1}^{m} \Pr_{v\in\{0,1\}^m}[u_i \oplus v \notin S] \\
&\leq m \cdot 2^{-n} \\
&< 1.
\end{aligned}$$

   Thus a v exists such that for all i, $u_i \oplus v \in S$. Since $u_1, \ldots, u_m$ were arbitrary, we are done.

■

Lemma 15.7 *Let* C *be a complexity class such that* $\mathsf{Pos}(\mathsf{C}) = \mathsf{C}$. *Then*

1. $\mathsf{BP}\text{-}\mathsf{C} \subseteq \exists\forall\mathsf{C}$ *and*
2. $\mathsf{BP}\text{-}\mathsf{C} \subseteq \forall\exists\mathsf{C}$.

Proof. Let A be a language in $\mathsf{BP}\text{-}\mathsf{C}$. Since C is closed under Pos, we can do probability amplification: There is a language $B \in \mathsf{C}$ and some polynomial p such that for all x,

$$\begin{aligned}
x \in A &\Longrightarrow \Pr_{y\in\{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \geq 1 - 2^{-n}, \\
x \notin A &\Longrightarrow \Pr_{y\in\{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \leq 2^{-n}.
\end{aligned}$$

Let $T_x = \{y \mid \langle x, y\rangle \in B\}$. If $x \in A$, then $|T_x| \geq (1 - 2^{-n})2^{p(|x|)}$. In this case, the first statement of Lemma 15.6 is true. If $x \notin A$, then $|\bar{T}_x| \geq (1 - 2^{-n})2^{p(|x|)}$. Thus the second statement of Lemma 15.6 is true for $\bar{T}_x$. But this is the negation of the first statement for T_x. Hence

$$x \in A \iff \exists^P u_1, \ldots, u_{p(|x|)} \forall^P v : u_1 \oplus v \in T_x \vee \cdots \vee u_{p(|x|)} \oplus v \in T_x.$$

The relation on the righthand side clearly is in $\mathsf{Pos}(\mathsf{C})$, and hence, $A \in \exists\forall\mathsf{C}$.

In the same way, we get $A \in \forall\exists\mathsf{C}$, since if $x \in A$, then also the second statement of Lemma 15.6 is true for T_x and if $x \notin A$, then the first statement is true for $\bar{T}_x$. ■

Corollary 15.8 (Sipser) $\mathsf{BPP} \subseteq \Sigma_2^\mathsf{P} \cap \Pi_2^\mathsf{P}$.

Corollary 15.9 *If* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{P} = \mathsf{BPP}$.

16 Testing polynomials for zero

In 1977, Solovay and Strassen proposed a new type of algorithm for testing whether a given number is a prime, the celebrated randomized Solovay-Strassen primality test. This test and similar ones proved to be very useful. This fact changed the common notion of "feasible computations" to probabilistic polynomial time algorithms with bounded error. Since then, the field of randomized algorithms flourished, with primality being one of its key problems. "Unfortunately", Agrawal, Kayal, and Saxena recently proved that primality can be decided in deterministic polynomial time, taking away one of the main arguments for probabilistic computations.

In this chapter, we look at another important problem, algebraic circuit identity testing—ACIT for short, that has a polynomial time randomized algorithm but for which we do not have a deterministic one.

Note that we do not know whether BPP and RP have complete problems. Thus we cannot show that ACIT is BPP-complete or something like that. Even a generic problem as the set of all triples $\langle M, x, 1^t \rangle$ such that M is a polynomial time probabilistic machine with error probability at most $1/3$ that accepts x within t steps is not BPP-complete. It is BPP-hard, but not in BPP, since it is even undecidable whether the error probability of M is bounded by $1/3$. Being a BPP-machine is a *semantic* property, while being an NP-machine is a *syntactic* one.[1] (One way out of this dilemma are so-called partial languages or promise problems. There is a version of BPP called promise-BPP that naturally has complete problems.)

So, can the same happen to ACIT and AFIT what happened to PRIMES? Can we show that ACIT and AFIT, the remaining prominent problems for which we do not have a deterministic polynomial time algorithm, are in P without settling whether any bounded error probabilistic polynomial time algorithm can be derandomized? Recently, Kabanets and Impagliazzo showed that derandomizing ACIT and AFIT will immediately prove circuit lower bounds, a notorious hard problem.

[1] While it is not decidable whether a Turing machine is $p(n)$ time bounded, we can enumerate all Turing machines and add to each Turing machine a counter that counts the steps up to $p(n)$. There does not seem to exist an analogue for probabilistic Turing machines.

16.1 Arithmetic circuits

Consider the following task: Given a polynomial p of degree d in n variables $X_1, \ldots, X_n$ over $\mathbb{Z}$, decide whether p is identically zero. If the coefficients of p are given, then this task is of course easy. Representing a polynomial in such a way might not be that clever, since it has $\binom{d+n}{n}$ coefficients. Representing polynomials by arithmetic formulas or circuits often is a much better alternative. We can efficiently evaluate the determinant, for instance by Gaussian elimination. Gaussian elimination treats the entries of the matrix as entities and computes the determinant using only arithmetic operations. This gives an arithmetic circuit for the determinant. (This is not quite true since there is the issue of pivoting and we would also need equality tests. But if we consider the entries of the input matrix as indeterminates, then no pivoting is necessary.)

An *arithmetic circuit* is an acyclic directed graph with exactly one node of outdegree zero, the *output gate*. Each gate has either indegree zero or two. A gate with degree zero is either labeled with a constant from $\mathbb{Z}$ or with a variable X_i. A gate of indegree two is either labeled with "+" (addition gate), "×" (multiplication gate), or "/" (division gate). In the latter case, we have to ensure that there is no division by zero (as a rational function). For simplicity, we will solely deal with division-free circuits in the following. (There is a general way to efficiently eliminate divisions when one wants to compute a polynomial due to Strassen.) An *arithmetic formula* is an arithmetic circuit where all gates except the output gate have outdegree one, i.e., the underlying graph is a tree. (Note that several input gates may be labeled with the same variable.)

The *size* of a circuit or formula is the number of nodes. A *description* of a circuit or formula is a binary encoding of it. The length of the description is the length of it as a binary string.

With each node in the circuit, we can associate a polynomial that is computed at this node. For an node with indegree zero, it is the polynomial that the node is labeled with. For a node v with degree two, we define this polynomial inductively: If p and q are the polynomials of the predecessors of v, then we associate the polynomial $p + q$ or $p \cdot q$ with this node. The polynomial at the output gate is the polynomial computed by the circuit.

Definition 16.1 *1. Arithmetic circuit identity testing problem* (`ACIT`)*: Given an (encoding of an) arithmetic circuit computing a polynomial p in $X_1, \ldots, X_n$, decide whether p is identically zero.*

2. Arithmetic formula identity testing problem (`AFIT`)*: Given an (encoding of a) arithmetic formula computing a polynomial p, decide whether p is identically zero.*

16.2 Testing for zero

How do we check whether a polynomial p given by a circuit or formula is identically zero? We can of course compute the coefficients of p from the circuit. The output may be exponential in the size of the circuit, so this is not efficient. A better way to solve this problem is provided by randomization. We simply assign random values to the variables and evaluate the circuit. If p is nonzero, then it is very unlikely that p will evaluate to zero at a random point. This intuition is formalized in the following lemma.

Lemma 16.2 (Schwartz–Zippel) *Let $p(X_1, \ldots, X_n)$ be a nonzero polynomial of degree d over a ring F. Let $S \subseteq F$ be finite. Then*

$$\Pr_{r_1,\ldots,r_n \in S}[p(r_1, \ldots, r_n) = 0] \leq d/|S|.$$

Proof. The proof is by induction in n. The case $n = 1$ is easy: A univariate polynomial $p \neq 0$ of degree d has at most d roots. The probability of picking such a root from S is at most $d/|S|$. For the induction step $n \to n+1$, we write p as an element of $F[X_1, \ldots, X_n][X_{n+1}]$. Let d' be the degree of X_{n+1} in p. We have

$$p(X_1, \ldots, X_{n+1}) = \sum_{\delta=0}^{d'} p_\delta(X_1, \ldots, X_n) X_{n+1}^\delta \qquad \text{with } p_\delta \in F[X_1, \ldots, X_n].$$

Obviously, $d' \leq d$ and $\deg p_{d'} \leq d - d'$. By the induction hypothesis applied to $p_{d'}$,

$$\begin{aligned}
&\Pr_{r_1,\ldots,r_{n+1} \in S}[p(r_1, \ldots, r_{n+1}) = 0] \\
&\leq \Pr_{r_1,\ldots,r_n \in S}[p_{d'}(r_1, \ldots, r_n) = 0] \\
&\quad + \Pr_{r_1,\ldots,r_{n+1} \in S}[p(r_1, \ldots, r_{n+1}) = 0 \mid p_{d'}(r_1, \ldots, r_n) \neq 0] \\
&\leq \deg p_{d'}/|S| + d'/|S| \\
&\leq d/|S|. \qquad \blacksquare
\end{aligned}$$

Now assume we are given a description of an arithmetic formula for a polynomial p of size s. Let $\ell \geq s$ be the length of the description.

Lemma 16.3 *The degree of a polynomial p computed by an arithmetic formula of size s is at most s.*

Proof. The claim is shown by induction on s:
Induction base: If $s = 1$, then the degree is either zero or one.
Induction step: If $s > 1$, then we decompose the given formula into two formulas by removing the output gate. Let the resulting subformulae have

sizes s_1 and s_2 and compute polynomials p_1 and p_2. We have $s_1+s_2+1 = s$. By the induction hypothesis, $\deg p_1 \leq s_1$ and $\deg p_2 \leq s_2$. We have $p = p_1 \cdot p_2$ or $p = p_1 + p_2$. In both cases, $\deg p \leq \deg p_1 + \deg p_2 \leq s$. ■

Now we choose the set $S = \{1, 2, \ldots, 2s\}$, from which we randomly pick the values and assign them to the variables. If p is zero, then p will evaluate to zero no matter what. If p is nonzero, then the Schwartz–Zippel lemma assures that our error probability is less than $s/(2s) = 1/2$.

What remains to be addressed is how to evaluate the formula after assigning values to the variables. If the largest absolute value of the constants in the formula is c, then the absolute value of the output is at most $(\max\{c, 2s\})^s$.

Exercise 16.1 *Prove the last claim.*

Its bit representation has at most $s \cdot \log \max\{c, 2s\}$ many bits. Since $\log c \leq \ell$ (the bit representation of c is somewhere in the encoding), this is polynomial in the length of the input. This show the following results.

Theorem 16.4 AFIT $\in$ co-RP.

The case where p is given by an arithmetic circuit is somewhat trickier. Here the degree of the computed polynomial might be almost as large 2^s but never more.

Exercise 16.2

1. *Prove the following: Any arithmetic circuit of size s computes polynomials of degree at most 2^{s-1}.*
2. *Construct an arithmetic circuit of size s that computes a polynomial of degree 2^{s-1}*
3. *Consider a circuit of size s and let c be an upper bound for the absolute values of the constants in C. Assume we evaluate the circuit at a point $(a_1, \ldots, a_n)$ with $|a_\nu| \leq d$, $1 \leq \nu \leq n$. Then $|C(a_1, \ldots, a_n)| \leq \max\{c, d\}^{2^s}$.*

Theorem 16.5 ACIT $\in$ co-RP.

Proof. The following Turing machine is a probabilistic Turing machine for ACIT.

Input: a description of length ℓ of a circuit C of size s.

1. Choose random values $a_1, \ldots, a_n \in \{1, \ldots, 8 \cdot 2^s\}$.
2. Let $m = 2^s \cdot \max\{\log c, s + 3\}$.
3. Choose a random prime number $q \leq m^2$ (Lemma 16.6.

4. Evaluate the circuit at $a_1, \ldots, a_n$ modulo q
5. Accept if the result is 0, otherwise reject.

Assume that in step 3, q is a prime number with probability $\geq 7/8$. q has $O(s \cdot \max\{\log\log c, \log s\})$ many bits. By Exercise 16.2, this is bounded by $O(s \log \ell)$. Thus we can evaluate the circuit modulo q in polynomial time, since we can perform the operation at every gate modulo q.

If C is zero, then the Turing machine will always accept C. (If we do not find a prime q in step 3, we will simply accept.)

Now assume that C is nonzero. By the Schwartz-Zippel lemma, the probability that C evaluates to zero is $\leq 2^s/(8 \cdot 2^s) = 1/8$. The probability that we do not find a prime in step 3 is 1/8, too. We have $|C(a_1, \ldots, a_n)| \leq \max\{c, 2^{s+3}\}^{2^s}$. Thus there are at most $2^s \cdot \max\{\log c, s+3\} = m$ different primes that divide $C(a_1, \ldots, a_n)$. The prime number theorem tells us that there are at least $m^2/(2\log m)$ many primes smaller than m^2. The probability that we hit a prime that divides $C(a_1, \ldots, a_n)$ is $(2\log m)/m \leq 1/8$ for s large enough. Thus the probability that the Turing machines accepts C is $\leq 3/8$. ∎

Lemma 16.6 *There is a probabilistic algorithm that given $M = 2^\mu$, returns with probability $\geq 3/4$ a random prime and "failure" otherwise.*

Proof. Consider the following Turing machine:

Input: $M = 2^\mu$

1. Do $c \cdot \mu$ times:
2. Choose a random number r with μ bits
3. Check deterministically whether r is prime
4. If r is prime, return r
5. return "failure"

By the prime number theorem, r is prime with probability $\geq 1/\mu$. Thus, the probability that we do not find a prime is $(1 - \frac{1}{\mu})^{c\mu} \leq e^{-c}$. Choosing $c = \ln 4$ proves the lemma. ∎

17 The isolation lemma

Deciding whether a bipartite graph has a perfect matching is one of the problems in P for which we do not know whether it is in NC. In this chapter we show that this problem can be decided by randomized circuits of polynomial size and polylogarithmic depth.

17.1 Probabilistic circuits

In the same way we added randomness to Turing machines, a sequential model of computation, we can also add randomness to circuits. A circuit now gets two inputs x and y where $|y| = p(|x|)$ for some polynomial p. We think of y as a random string.

Definition 17.1 *1. A language L is in the class RNC_i if there exists a logarithmic space uniform family of circuits C (with two inputs) of polynomial size and depths $\mathrm{O}(\log^i n)$ such that*

$$x \in L \Longrightarrow \Pr_y[C(x,y) = 1] \geq 1/2,$$
$$x \notin L \Longrightarrow \Pr_y[C(x,y) = 1] = 0.$$

2. $\mathsf{RNC} = \bigcup_{i \in \mathbb{N}} \mathsf{RNC}_i$.

One could also define BPNC and ZPNC but these classes are rarely needed.

17.2 Matchings, permanents, and determinants

Let $G = (U \cup V, E)$ be a bipartite graph with bipartition $U = \{u_1, \ldots, u_n\}$, $V = \{v_1, \ldots, v_n\}$ and edges $E \subseteq U \times V$. A perfect matching M in G is a subset of E such that each $u \in U$ appears in exactly one $e \in M$ and each $v \in V$ appears in exactly one $e \in M$. Computing one perfect matching (or deciding whether one exists) can be done in polynomial time. It is an open problem whether there is an efficient parallel deterministic algorithm for this problem, i.e., it is unknown whether this problem is in NC. In this chapter we will show that it is in RNC.

Let $A_G = (a_{i,j})$ be the $n \times n$-matrix that has a 1 in position (i,j) if $(u_i, v_j) \in E$ and a 0 otherwise. The determinant of A_G is

$$\det A_G = \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}.$$

S_n is the symmetric group, i.e., the group of all permutations of $\{1, \ldots, n\}$. Any perfect matching defines a bijection from $\{u_1, \ldots, u_n\}$ to $\{v_1, \ldots, v_n\}$. We can interpret this as a permutation on $\{1, \ldots, n\}$. $\prod_{i=1}^{n} a_{i,\sigma(i)}$ equals 1, if $(u_i, v_{\sigma(i)}) \in E$ for all $1 \leq i \leq n$, i.e., the matching that corresponds to σ is in fact present in G. Thus the determinant counts all perfect matchings but either with 1 or -1 depending on the sign of the permutation. The problem is that the determinant can be zero even if the graph contains a perfect matching, since the matchings with 1 and -1 can cancel out.

If we really want to count all perfect matchings in G, then the *permanent* does it:

$$\operatorname{perm} A_G = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}.$$

The permanent of A_G is precisely the number of perfect matchings in G. While the determinant can be computed efficiently, even in parallel, evaluating the permanent seems to be hard. (We will formalize this in one of the next chapters). Thus the omission of $\operatorname{sign}(\sigma)$ makes the problem harder not easier!

We can still use the determinant to check whether a graph has a perfect matching or not. Instead of writing a 1 if there is an edge from u_i to v_i, we write an indeterminate $x_{i,j}$ instead. In this way, every perfect matching in G give rise to a unique monomial of the determinant (as a polynomial in the indeterminates) and hence they cannot cancel out.

Lemma 17.2 *Let G be a bipartite graph as above and let X_G be the matrix that has an indeterminate $x_{i,j}$ in position (i,j) if (u_i, v_j) is an edge of G and 0 otherwise. Then G has a perfect matching iff $\det X_G \neq 0$.*

Exercise 17.1 *The characteristic polynomial of a matrix A is defined as $c_A(X) = \det(A - X \cdot I)$ where I is the identity matrix. Let $c_A(X) = s_{A,0}X^n + s_{A,1}X^{n-1} + \cdots + s_{A,n}$.*

1. *Show that*

$$s_{A,0} = (-1)^n$$
$$s_{A,k} = \frac{1}{k} \sum_{\kappa=1}^{k} (-1)^{\kappa-1} s_{k-\kappa} \operatorname{trace}(A^\kappa), \qquad 1 \leq k \leq n.$$

2. *Show that $s_{A,n} = \det A$.*

3. *Show that there is a logarithmic space uniform family of Boolean circuits of polynomial size and polylogarithmic depth that computes the determinant of a matrix A. (Assume that A has dimension $n \times n$ and entries with $p(n)$ bits for some polynomial p.)*

While we can efficiently compute the determinant in parallel if the entries of the matrix are numbers, we cannot compute the determinant of a matrix with indeterminates, even not in polynomial time, since the number of monomial might be too large. But the Schwartz-Zippel lemma gives an efficient way to test whether a polynomial is zero or not, just plug in random values from a large enough (but not too large) set.

Now Lemma 17.2 and Exercise 17.1 together with the Schwartz-Zippel Lemma gives an efficient randomized parallel algorithm for deciding whether a given bipartite graph has a perfect matching.

Theorem 17.3 *Deciding whether a given bipartite graph contains a perfect matching is in* RNC.[1]

17.3 The isolation lemma

We will design an algorithm that will find a perfect matching in a bipartite graph with exactly one perfect matching and will even find a perfect matching of minimum weight in a weighted graph provided that the minimum weight perfect matching is unique. The result in this chapter shows that if we assign random weights to the edges of an unweighted graph, then with high probability, there is a unique minimum weight perfect matching.

Let S be a set and assume that each element $s \in S$ has a weight $w_s \in \mathbb{N}$. For $T \subseteq S$, the weight of T is $\sum_{t \in T} w_t$.

Lemma 17.4 *Let S be a set of size n and let F be a nonempty set of subsets of S. If we assign to each $s \in S$ a weight $w_s \in \{1, \ldots, 2n\}$ independently and uniformly at random, then*

$$\Pr[\textit{there is a unique minimum weight set in } F] \geq 1/2.$$

The amazing part of this lemma is that it is completely oblivious: It does not care about how F actually looks like, the same random process does the job for *every* F!

Proof. We call an $s \in S$ bad if it is contained in a minimum weight set but not in all of them. It is obvious that there is a bad element s iff the minimum weight set is not unique.

[1] We showed that ACIT $\in$ co-RP, so one might expect that the problem is in co-RNC. But note that elements in ACIT are circuits that compute the zero polynomial whereas determinant $= 0$ means that the graph has *not* a perfect matching.

Suppose that all weights have been assigned except the one for s. Let $W_s := \min\{w(T) \mid T \in F,\ s \notin T\}$ and $V_s := \min\{w(T \setminus \{s\}) \mid T \in F,\ s \in T\}$.

We claim that s can only be bad if we assign to s the weight $W_s - V_s$. If $w_s < W_s - V_s$, then the sets that do not contain s are not minimum weight sets, hence s cannot be bad. If $w_s > W_s - V_s$, then the sets that contain s cannot be minimum weight sets, again s cannot be bad.

Since s is chosen from a set of size $2n$, the probability that s is bad is $\leq 1/(2n)$. There are n elements, thus, the probability that none of them is bad is $\geq 1 - 1/(2n) \cdot n = 1/2$. ∎

In our algorithm, F will be the set of all matchings of a graph. After assigning weights from $\{1, \ldots, 2|E|\}$ to the edges, we know that with probability $\geq 1/2$, there is a unique minimum weight perfect matching.

17.4 Constructing perfect matchings

We already saw how to compute a satisfying assignment given a procedure that decides `SAT`. But self-reducibility seems to be inherently sequential. For perfect matching, there is a way of doing self reducibility in parallel:

Input: an incidence matrix A of a bipartite graph $G = (V, E)$

1. Choose random values $w_e \in \{1, \ldots, 2|E|\}$ for each $e \in E$ and give each edge the weight 2^{w_e}. Let B be the resulting matrix.
2. Compute $\det B$. If $\det B = 0$ return "no matching".
3. Compute the largest R such that 2^R divides $\det B$.
4. Do the following in parallel:
   (a) For each edge e, let B_e the matrix that is obtained from B by setting the entry of e to 0.
   (b) Compute $\det B_e$. If $\det B_e = 0$, then output e.
   (c) Otherwise, compute the largest R_e such that 2^{R_e} divides $\det B_e$.
   (d) If $R < R_e$, then output e.

Every matching corresponds to a monomial $\operatorname{sign}(\pi) x_{1,\pi(1)} \cdots x_{n,\pi(n)}$. Assume that π corresponds to an actual matching M in G. If we replace x_e by 2^{w_e}, then the term above becomes $\operatorname{sign}(\pi) 2^{w(M)}$. By the isolation lemma, with probability $1/2$, the minimum weight perfect matching is unique. Let W be the weight of the minimum weight perfect matching. Then $2^W \mid \det B$. All other matchings contribute terms $\pm 2^{W'}$ with $W' > W$. Thus all other terms are divided by 2^{W+1}. Hence, $\det B = 2^W(1+2b)$ for some odd number $(1 + 2b)$, which is potentially negative, and the number R computed by the

circuit is indeed the weight of a minimum weight perfect matching. Note that all the weights have a polynomial number of bits.

So far everything could be done in logarithmic depth, we just needed to compute a determinant. Finding R is a little tricky. If we assume that all number are stored in signed representation (and not in 2-complement), then we just have to find the first 1 in the binary expansion of $\det B$. It is an easy exercise to do this in logarithmic depth.

Next we remove each edge e in parallel and again compute the weight of a minimum weight perfect matching: If e is not in the unique minimum weight perfect matching, then $R_e = R$. If e is in the minimum matching, then this term is canceled and $\det B_e = 2^R(2b')$ and hence, $R_e > R$. Thus for every edge, we correctly compute whether it is in the unique minimum weight perfect matching or not. The only technicality is how to output the matching. We have m subcircuits, n of which computed an edge and the other ones reported a failure. We have to find out which circuits actually found edges and move their result to the right output gates. But this is again a not too hard exercise.

Theorem 17.5 *There is a logarithmic space uniform family of probabilistic circuits of polynomial size and polylogarithmic depth that given an incidence matrix of a bipartite graph computes with probability* $1/2$ *a perfect matching of* G*, if one exists, and reports failure otherwise.*

18 The Valiant–Vazirani Theorem

Suppose we have a polynomial time bounded deterministic Turing machine A for SAT that always finds a satisfying assignment for a formula ϕ provided that ϕ has exactly one satisfying assignment. (And we do not care what A does on other formulas.) We will show that this is already sufficient to efficiently solve all problems in NP, namely, NP = RP follows.

Proof overview: The key idea is to design a randomized reduction that maps ϕ to formulas $\psi_0, \ldots, \psi_n$ with the following properties: If ϕ is not satisfiable, then none of the ψ_ν is. If ϕ is satisfiable, then with high probability at least one ψ_ν has exactly one satisfying assignment. We run A on the formulas and check whether the computed assignments satisfy the formula or not. Each ψ_ν has the form $\phi \wedge f(x)$ where f is an additional formula that is satisfied by one out of $\approx 2^\nu$ assignments. Hence if ϕ has about 2^ν assignments, then ψ_ν will have exactly one assignment (with high probability).

18.1 Pairwise independent hash functions

Definition 18.1 *1. A family H of functions $\{0,1\}^n \to \{0,1\}^m$ is a* family of universal hash functions *if for every two different $x, y \in \{0,1\}^n$,*

$$\Pr_{h \in H}[h(x) = h(y)] = \frac{1}{2^m}.$$

2. H is a family of pairwise independent hash functions *if for every two different $x, y \in \{0,1\}^n$ and every $u, v \in \{0,1\}^m$,*

$$\Pr_{h \in H}[h(x) = u \wedge h(y) = v] = \frac{1}{2^{2m}}.$$

If H is a family of pairwise independent hash functions, them by summing over all v, we get that $\Pr_{h \in H}[h(x) = u] = \frac{1}{2^m}$. An easy calculation now shows that $\Pr_{h \in H}[h(x) = u | h(y) = v] = \Pr_{h \in H}[h(x) = u]$.

The idea of the above reduction is as follows. Assume that ϕ has about 2^m satisfying assignments. Then the probability that an assignment a satisfies ϕ and $h(a) = (0, \ldots, 0)$ for some randomly chosen h from a family of pairwise independent hash functions would roughly be 2^{-m}. This means, we expect about one such assignment.

We define a family A of functions $\{0,1\}^n \to \{0,1\}^m$ as follows: For m vectors $a_1, \ldots, a_m \in \{0,1\}^n$ and m number $b_1, \ldots, b_m \in \{0,1\}$, define $h_{a_1,\ldots,a_m,b_1,\ldots,b_m}$ by $x \mapsto (a_1x+b_1, \ldots, a_mx+b_m)$. Here a_ix denotes the scalar product of a_i and x and all computations are modulo 2, i.e., we compute over GF(2). In particular, we view $\{0,1\}^n$ as a vector space over the field GF(2).

Theorem 18.2 *A is a family of pairwise independent hash functions.*

Proof. Let $x, y \in \{0,1\}^n$, $x \neq y$ and $u, v \in \{0,1\}^m$. Consider the event

$$a_1x + b_1 = u_1 \wedge a_1y + b_1 = v_1.$$

(Here u_i and v_i are the entries of the vectors u and v.) The event above is the same as

$$a_1(x+y) = u_1 + v_1 \wedge b_1 = v_1 - a_1y.$$

(We added the equation on the righthand side to the one on the lefthand side. This is an invertible linear transformation.) Its probability is

$$\Pr_h[b_1 = v_1 - a_1y | a_1(x+y) = u_1 + v_1] \Pr_h[a_1(x+y) = u_1 + v_1].$$

The conditional probability is $1/2$, since the righthand side of the first equation determines b_1. Since everything else is independent of b_1, b_1 fulfills this equation with probability $1/2$. The other probability is also $1/2$: Since $x \neq y$, there is one component where x and y differ, say $x_1 \neq y_1$. Then $a_1(x+y) = u_1 + v_1$ determines $a_{1,1}$ via

$$a_{1,1}a_{1,1}(x_1 + y_1) = u_1 + v_1 - \sum_{i=2}^{n} a_{1,i}(x_i + y_i).$$

Thus

$$\Pr_h[a_1(x+y) = u_1 + v_1 \wedge b_1 = v_1 - a_1y] = \frac{1}{4}.$$

Since the events

$$a_ix + b_i = u_i \wedge a_iy + b_i = v_i$$

are all independent,

$$\Pr_h[h(x) = u \wedge h(y) = v] = \frac{1}{4^m}. \qquad \blacksquare$$

18.2 Making solutions unique

We use hash functions to make a satisfying assignment unique.

Lemma 18.3 *Let $S \subseteq \{0,1\}^n$ with $2^k \leq |S| < 2^{k+1}$ and let H be a family of pairwise independent hash functions $\{0,1\}^n \to \{0,1\}^{k+2}$. Then*

$$\Pr_{h \in H}[|\{x \in S \mid h(x) = (0,\ldots,0)\}| = 1] \geq \frac{1}{8}.$$

Proof. Fix some $s \in S$. The probability that s is the only element in S that is mapped to $z := (0,\ldots,0)$ is

$$\begin{aligned} & \Pr_h[h(s) = z \wedge \forall t \in S \setminus \{s\} : h(t) \neq z] \\ = \; & \Pr_h[h(s) = z] \Pr_h[\forall t \in S \setminus \{s\} : h(t) \neq z | h(s) = z]. \end{aligned}$$

We have

$$\Pr_h[h(s) = z] = \frac{1}{2^{k+2}}$$

and

$$\Pr_h[\forall t \in S \setminus \{s\} : h(t) \neq z | h(s) = z] = 1 - \Pr_h[\exists t \in S \setminus \{s\} : h(t) = z | h(s) = z].$$

Now,

$$\begin{aligned} \Pr_h[\exists t \in S \setminus \{s\} : h(t) = z | h(s) = z] &\leq \sum_{t \in S \setminus \{s\}} \Pr_h[h(t) = z | h(s) = z] \\ &\leq \sum_{t \in S \setminus \{s\}} \Pr_h[h(t) = z] \\ &= \frac{|S| - 1}{2^{k+2}} \\ &\leq \frac{1}{2}. \end{aligned}$$

Here the second line follows from the pairwise independency of the family H. Putting everything together, we get that

$$\Pr_h[h(s) = z \wedge \forall t \in S \setminus \{s\} : h(t) \neq 0] \geq \frac{1}{2^{k+3}}.$$

The probability that there is a unique element that is mapped to z is

$$\sum_{s \in S} \Pr_h[h(s) = z \wedge \forall t \in S \setminus \{s\} : h(t) \neq 0] \geq \frac{|S|}{2^{k+3}} \geq \frac{1}{8}. \qquad \blacksquare$$

Lemma 18.4 *There is a polynomial time bounded probabilistic Turing machine that given a formula ϕ and an integer k outputs a formula ψ such that*

1. *if ϕ is not satisfiable, so is ψ, and*
2. *if ϕ is satisfiable and the number of satisfying assignments of ϕ is in $[2^k, 2^{k+1})$, then ψ has a unique satisfying assignment with probability at least $1/8$.*

Proof. Let $x_1, \ldots, x_n$ be the variables of ϕ and $x = (x_1, \ldots, x_n)$. The Turing machine M picks $a_1, \ldots, a_{k+2} \in \{0,1\}^n$ and $b_1, \ldots, b_{k+2} \in \{0,1\}$ uniformly at random. The formula ψ is equivalent to $\phi(x) \wedge (a_1 x + b_1 = 0) \wedge \cdots \wedge (a_{k+2} x + b_{k+2} = 0)$. The term $a_1 x + b_1 = 0$ might have an exponentially large CNF. We circumvent this problem by viewing the expression as a circuit and let ψ be the formula obtained as in the reduction from `CSAT` to `SAT`. (Exercise: Check that this reduction preserves the number of satisfying assignments.)

By construction, the number of satisfying assignments of ψ is the number of these assignments that satisfy ϕ and are mapped to $(0, \ldots, 0)$ by the hash function. Thus, if ϕ is not satisfiable, so is ψ. If the number of satisfying assignments is in $[2^k, 2^{k+1})$, then by Lemma 18.3, ψ has exactly one satisfying assignment with probability $\geq 1/8$. ∎

Theorem 18.5 (Valiant–Vazirani) *If there is a polynomial time Turing machine M that given a formula having exactly one satisfying assignment finds this assignment, then $\mathsf{NP} = \mathsf{RP}$. (We do not make any assumption about the behavior on M on other formulas.)*

Proof. It is sufficient to show that $\texttt{SAT} \in \mathsf{RP}$. We claim that the following probabilistic Turing machine decides `SAT`:

Input: a formula ϕ in CNF

1. Construct formulas $\psi_0, \ldots, \psi_n$ as in Lemma 18.4 for all values $k = 0, \ldots, n$.
2. Simulate M on $\psi_0, \ldots, \psi_n$
3. Check whether M produces a satisfying assignment for at least one ψ_i.
4. If no, then reject ϕ. If yes, accept ϕ.

If ϕ is not satisfiable, then none of $\psi_0, \ldots, \psi_n$ is. Thus we will always reject ϕ. If ϕ is satisfiable, then it has between 2^k and 2^{k+1} satisfying

assignments for some k. With probability $\geq 1/8$, ψ_k will have a unique satisfying assignment. (We need the satisfying assignment produced by M to check whether ϕ is indeed satisfied, since we do not know the value of k.) In this case, M will accept. Thus we accept ϕ with probability $\geq 1/8$. Using probability amplification, we can amplify this probability to $1/2$. ■

18.3 Further exercises

Exercise 18.1 *Show that* $\mathsf{NP} = \mathsf{RP}$ *if there is a polynomial time bounded deterministic Turing machine that given a formula* ϕ *that has either zero or one satisfying assignments accepts this formula iff it is satisfiable. (Again we do not care what the machine does on other inputs.)*

19 Counting problems

19.1 #P

Definition 19.1 *1. Let R be an* NP *relation. Then $\#R : \Sigma^* \to \mathbb{N}$ is the function defined by*

$$\#R(x) = |\{y \mid R(x,y) = 1\}|.$$

2. $\#\mathsf{P} = \{\#R \mid R$ is an NP*-relation $\}$. (#P is usually pronounced as "sharp P" or "number P".)*

Unlike previous classes, $\#P$ is not a class of languages but of functions. When we decide $L(R)$, we want to check for a given input x whether there is a y such that $R(x,y)$. When we compute $\#R(x)$, we count the number of y such that $R(x,y) = 1$.

Exercise 19.1 *Show the following: $f \in \#\mathsf{P}$ iff there is a polynomial time nondeterministic Turing machine M such that for all x, $f(x)$ is the number of accepting paths of M.*

Exercise 19.2 *Show that if $f, g \in \#\mathsf{P}$, so are $f+g$, $f \cdot g$, and $x \mapsto f(x)^{p(|x|)}$ for every polynomial p.*

Of course, we want to talk about #P-completeness. One has to be a little careful about the kind of reduction.

Definition 19.2 *Let $f, g : \Sigma^* \to \mathbb{N}$. Let $s : \Sigma^* \to \Sigma^*$ and $t : \mathbb{N} \to \mathbb{N}$ be polynomial time computable; for computing t, we use the binary encoding.*

1. *(s,t) is a* polynomial time many one reduction *from f to g, if $f(x) = t(g(s(x)))$ for all $x \in \Sigma^*$. f is* polynomial time many one reducible *to g, denoted by $f \leq_{\mathrm{P}} g$, if a polynomial time many one reduction from f to g exists.*

2. *s is a* parsimonious reduction *from f to g if $f(x) = g(s(x))$ for all $x \in \Sigma^*$. In this case, we write $f \leq_{\mathrm{par}} g$.*

3. *f is called* polynomial time Turing reducible *to g, denoted by $f \leq_{\mathrm{P}}^{\mathrm{T}} g$, if there is a polynomial time oracle Turing machine M such that $f(x) = M^g(x)$ for all $x \in \Sigma^*$.*

Here a many one reduction consists of two functions. s maps instances of f to instances of g. t recovers the answer of f from the answer of g. A parsimonious reduction is a many one reduction where t is the identity. Let $f = \#R$ and $g = \#S$ for two NP relations R and S. If $f \leq_{\text{par}} g$, then

$$f(x) = |\{y \mid R(x,y) = 1\}| = g(s(x)) = |\{z \mid S(s(x),z) = 1\}|,$$

i.e., x and $s(x)$ have the same number of solutions. We have $f \leq_{\text{par}} g \Longrightarrow f \leq_{\text{P}} g \Longrightarrow f \leq_{\text{P}}^{\text{T}} g$.

Lemma 19.3 *$\leq_{\text{par}}$, $\leq_{\text{P}}$, and $\leq_{\text{P}}^{\text{T}}$ are transitive.*

Proof. Let $f \leq_{\text{P}} g$ and $g \leq_{\text{P}} h$ with reductions (s,t) and (u,v). Then $(u \circ s, t \circ v)$ is a polynomial time many one reduction from f to h:

$$t(v(h(u(s(x))))) = t(g(s(x))) = f(x)$$

for all x.

If v and t are the identity, so is $v \circ t$. Thus the same holds for parsimonious reductions.

For $\leq_{\text{P}}^{\text{T}}$, the proof works the same as for decision problems. ∎

Theorem 19.4 *#CSAT is #P-complete under parsimonious reductions.*

Proof. Let R be an NP-reduction. R is computable in polynomial time. When we showed that CSAT is NP-complete, we constructed a polynomial time computable mapping that maps each x to a circuit C_x such that $C_x(y) = 1$ iff $R(x,y) = 1$. This is obviously a parsimonious reduction from $\#R$ to #CSAT. ∎

Exercise 19.3 *Show that #3SAT is #P-complete under parsimonious reductions.*

Reductions for counting problems

$$\begin{array}{ccc} x & \xrightarrow{s} & s(x) \\ \downarrow & & \downarrow \\ f(x) & \xleftarrow{t} & g(s(x)) \end{array}$$

19.2 The permanent

Let R, S be NP-relations. If $\#R$ is #P-hard under parsimonious reductions, then $L(R)$ is NP-hard under many-one reductions. If s is a parsimonious reduction from $\#S$ to $\#R$, then $|\{y \mid S(x,y) = 1\}| = |\{z \mid R(s(x),z) = 1\}|$. In particular, $x \in L(S)$ iff $s(x) \in L(R)$.

But if we look at many one reductions (or even Turing reductions), then there are problems that are $\#P$-complete and the corresponding search problem is in P. An example is the problem of counting the perfect matchings in a bipartite graph. Let $G = (U \cup V, E)$ be a bipartite graph with bipartition $U = \{u_1, \ldots, u_n\}$, $V = \{v_1, \ldots, v_n\}$ and edges $E \subseteq U \times V$. Computing one perfect matching (or deciding whether one exists) can be done in polynomial time. But counting all perfect matchings, that is, computing the permanent of the adjacency matrix A_G, is $\#P$-hard, as will we see in the following.

For the proof of the #P-hardness of the permanent it is more convenient to think in terms of *cycle covers*. For a given G, we define a directed graph $G' = (V, A)$, possibly having loops. There is a directed edge $(v_i, v_j) \in A$ iff $(u_i, v_j) \in E$. A *cycle cover* in G' is a collection of node disjoint cycles such that each node is contained in exactly one cycle. (Loops count as cycles.) It is easy to see that cycle covers of G' stand in one-to-one correspondence with perfect matchings of G. Thus instead of counting perfect matchings of G we can count cycle covers of G' instead.

Theorem 19.5 (Valiant) perm *is* #P*-complete under many-one reductions and even for* $\{0,1\}$ *matrices.*

Proof. We reduce #3SAT to perm, i.e., to couting the number of cycle covers in a directed graph. A given formula ϕ with n variables and m clauses is mapped to a graph G_ϕ. This graph will have $O(m)$ edges. We normalize the given formula in such a way that every variable appears as many times negated as unnegated. We can achieve this by adding clauses of the form $x \vee x \vee \bar{x}$ or $x \vee \bar{x} \vee \bar{x}$.

For every variable x, there is a selector gadget, see Figure 19.1 (left-hand side). There are two ways to cover this gadget by a cycle cover, taking the left-hand edge will correspond to setting x to zero and taking the right-hand edge will correspond to setting x to one.

For every clause, there is a clause gadget as depicted in Figure 19.1 (right-hand side). Each of the three outer edges corresponds to one literal of the clause. Taking one of the three outer edges corresponds to setting the literal to zero. For every subset of the outer edges, except for the one consisting of all three outer edges, there is exactly one cycle cover, see Figure 19.2. Call the graph constructed so far G'_ϕ. A cycle cover of G'_ϕ is called *consistent*, if the chosen edges in the selector gadgets and the clause gadgets are consistent, that is, whenever we chose the left-hand edge in the selector

Figure 19.1: Left-hand side: The selector gadget. In all Figures, edges without explicitely stated weights have weight 1. Right-hand side: The clause gadget. In the gadget as it is, there is a double edge between the two nodes at the bottom. The lower edge is however subdivided when we introduce the equality gadgets.

gadget for x (i.e, $x = 0$), then we choose all corresponding edges in the clause gadgets in which x appears positively and vice versa.

Fact 19.6 *Satisfying assignments of ϕ and consistent cycle covers of G'_ϕ stand in one-to-one correspondence.*

The last step is to get rid of inconsistent cycle covers. This is done by connecting the edge of a literal ℓ in a clause gadget to the edge in the selector gadget corresponding to setting $\ell = 0$ using an equality gadget, see Figure 19.3. The edge of the selector gate and the edge of the clause gadget are subdivided, let x and z be the newly introduced vertices. These two vertices are connected as depicted in Figure 19.3. When a literal may appear in several clauses, the edge of the selector gadget is subdivived as many times.

Every consistent cycle cover of G'_ϕ can be extended to several cycle covers of G_ϕ. If the two edges connected by a equality gadget are both taken, then we take both path $u - x - v$ and $u' - z - v'$ in G_ϕ. The interior vertex y is covered by the self-loop, yielding a weight of -1. If both edges are not taken, then we take none of the corresponding paths. There are six possibility to cover the interior nodes x, y, and z; four of them have weight 1, two of them have weight -1. This sums up to 2. (The six different covers are shown in Figure 19.4.) Therefore, every consistent cycle cover is mapped to several cycle covers with a total weight of $(-1)^p 2^q$ where p is the number of literals set to zero and q is the number of literals set to one. Since we normalized ϕ, $p = q = 3m/2$.

There are also cycle covers that do not cover equality gadget consistently. This can either mean that the path $u - x - v$ is taken but not $u' - z - v'$ or that we enter the gadget via u but leave it via v'. One can prove that all cycle covers in which at least one equality gadget is not covered consistently sum up to zero.

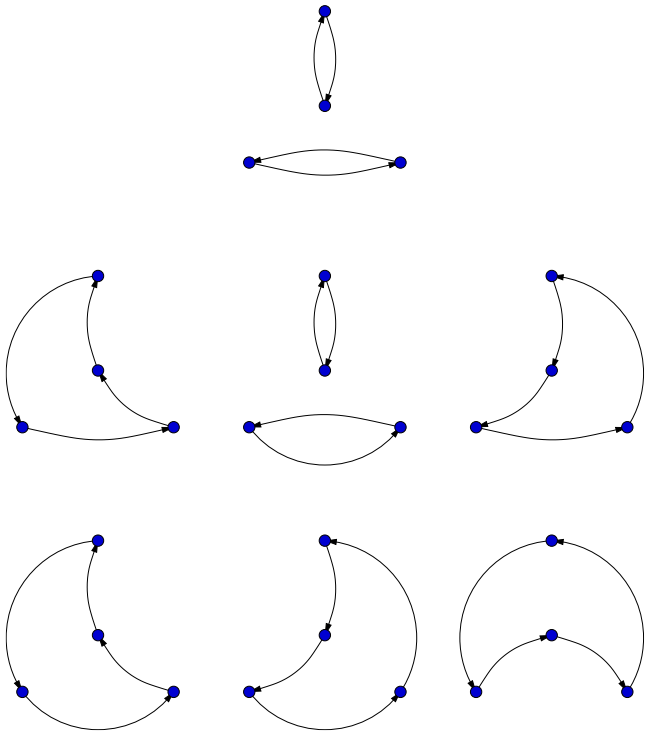

Figure 19.2: Every proper subset of the outer edges can be extended to a unique cycle cover of the clause gadget.

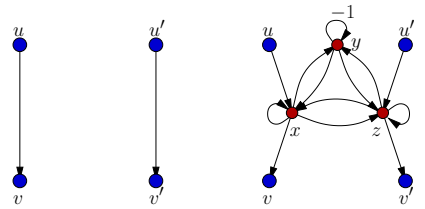

Figure 19.3: The equality gadget. The pair of edges (u, v) and (u', v') of the left-hand side, one of them is an edge of the selector gadget and the other is the corresponding outer edge of a clause gadget, is connected as shown on the right-hand side.

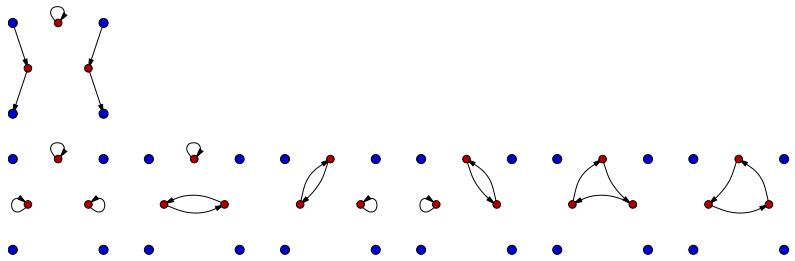

Figure 19.4: First row: The one possible configuration if both edges are taken. Second row: The six possible configurations if none of the edges is taken.

We will define an involution I without fixed points on the set of all inconsistent cycle covers of G_ϕ such that the weight of C and $I(C)$ cancel each other. From this it follows that the total contribution of the inconsistent cycle covers is zero. To define I, take an inconsistent cycle cover. We order the equality gadgets arbitrarily. Let C be an inconsistent cycle cover and consider the first inconsistent equality gadget. Then either C uses the path $u - x - v$ in this gadget but not $u' - z - v'$ or it enters the gadget via u and leaves it via v'. (The cases where $u' - z - v'$ is used but not $u - x - v$ or the gadget is entered via u' and left via v are symmetric.) Figure 19.5 shows how I pairs inconsistent cycle covers.

In the first case, C and $I(C)$ only differ in how y and z are covered. On the lefthand side, we use two cycles of total weight -1, on the righthand side, we use one cycle of weight 1. So the weights of the cycle covers are the same, but the signs differ. The symmetric case is treated in the same way.

In the second case, we either use one edge and cover y by a cycle of weight -1 (lefthand side), or we use two edges. Again, the contribution of both cycle covers is the same but with different signs.

Altogether, we get that $\operatorname{perm} G_\phi = (-2)^{3m/2} \cdot \#\texttt{3SAT}(\phi)$, where $\#\texttt{3SAT}(\phi)$

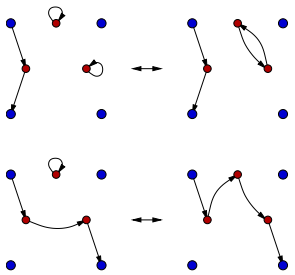

Figure 19.5: The involution I. I maps the configuration on the left-hand side to the corresponding configuration on the right-hand side and vice versa.

denotes the number of satisfying assignments of ϕ.

Finally it remains to simulate the weights -1. To this aim, we replace the edge weight -1 by 2^k where $k = 3m$. The weight 2^k can be simulated by a directed path of length k with a double edge between every pair of adjacent nodes. Then we replace every double edge by a path of length two to make the adjacency matrix of the final graph $\{0, 1\}$.

Let P be the permanent of the new graph. Let $0 \leq r < 2^k + 1$ the remainder when dividing P by $2^k + 1$. We claim that r is the number of cycle covers in the original graph. Notice that modulo $2^k + 1$, 2^k equals -1. The number of cycle covers is at most $2^n \cdot 2^{3m/2} < 2^{3m} < 2^k + 1$. Thus r is the right number. ■

Open Problem 19.7 *I do not know of any* NP*-complete problem such that its counting version is not known to be* #P*-hard. However, one always needs an individual proof for every such problem. It is open whether the following is true: For every* NP*-relation R such that $L(R)$ is* NP*-hard, $\#R$ is* #P*-hard.*

20 Toda's theorem

How much more powerful is counting the number of witnesses compared to deciding whether there exists a witness? Astonishingly, counting is much more powerful than deciding: We will show that $\mathsf{P}^{\#\mathsf{P}}$ contains the whole polynomial time hierachy![1]

Proof overview: First we introduce the class $\oplus\mathsf{P}$, a class that corresponds to a very simply way of counting, namely counting modulo two. It turns out that already this class is very powerful: From the Valiant–Vazirani Theorem, it essentially follows that $\mathsf{NP} \subseteq \mathsf{BP}\text{-}\oplus\mathsf{P}$. We can even stronger show that $\exists\oplus\mathsf{P} \subseteq \mathsf{BP}\text{-}\oplus\mathsf{P}$. Once we have this, an easy induction shows that $\mathsf{PH} \subseteq \mathsf{BP}\text{-}\oplus\mathsf{P}$. Finally, we show that $\mathsf{BP}\text{-}\oplus\mathsf{P} \subseteq \mathsf{P}^{\#\mathsf{P}}$.

20.1 $\oplus\mathsf{P}$

Definition 20.1 *The class $\oplus\mathsf{P}$ (read "parity P" or "odd P") is the class of all languages L such that there is an NP-relation R and for all x we have*

$$x \in L \iff \#R(x) \text{ is odd.}$$

Compared to NP, we do want to know whether there is an odd number of accepting paths and not just one. Compared to $\#\mathsf{P}$, we have a very rudimentary form of counting, namely of counting modulo two.

Lemma 20.2 *1.* $\mathsf{P} \subseteq \oplus\mathsf{P}$.

2. $\oplus\mathsf{P} = \mathsf{co}\text{-}\oplus\mathsf{P}$.

Proof.

1. 0 is even, 1 is odd.
2. Consider the following NP-relation:

$$S(x,y) = \begin{cases} 1 & \text{if } y = 0 \\ R(x,y') & \text{if } y = 1y' \\ 0 & \text{otherwise} \end{cases}$$

We have $\#S(x) = \#R(x) + 1$. ∎

[1] We cannot directly compare NP with $\#\mathsf{P}$, since they contain different objects, namely sets and functions.

$\oplus$P has complete problems under polynomial time many-one reductions. For instance $\oplus$3SAT is, where a formula ϕ in 3-CNF is in $\oplus$ 3SAT iff it has an odd number of satisfying assignments. This follows from the fact that #3SAT is #P-complete under parsimonious reductions.

Exercise 20.1 *Let R be an* NP *relation. Let $\oplus R$ be the language $\{x \mid$ the number of y with $R(x,y) = 1$ is odd$\}$. Prove that if $\#R$ is* #P*-hard under parsimonious reductions, then $L(R)$ is* NP*-hard under many-one reductions and $\oplus R$ is $\oplus$*P*-hard under many-one reductions.*

Exercise 20.2 *Show that we can deterministically decide in polynomial time whether the permanent of an integer matrix is odd.*

From the proof of the Valiant-Vazirani theorem, we get the following result with only little extra work.

Theorem 20.3 NP $\subseteq$ BP-$\oplus$P.

Before we can prove it, we first show three useful results about $\oplus$3SAT.

Lemma 20.4 *There are deterministically polynomial time computable functions f, g, h such that for all formulas $\phi_1, \ldots, \phi_m$,*

1. $\phi_1 \in \oplus\mathtt{3SAT} \wedge \cdots \wedge \phi_m \in \oplus\mathtt{3SAT} \iff f(\phi_1, \ldots, \phi_m) \in \oplus\mathtt{3SAT}$,
2. $\phi_1 \in \oplus\mathtt{3SAT} \iff g(\phi_1) \notin \oplus\mathtt{3SAT}$,
3. $\phi_1 \in \oplus\mathtt{3SAT} \vee \cdots \vee \phi_m \in \oplus\mathtt{3SAT} \iff h(\phi_1, \ldots, \phi_m) \in \oplus\mathtt{3SAT}$.

Proof.

1. Let t_i be the number of satisfying assignments of ϕ_i. $\psi = \phi_1 \wedge \cdots \wedge \phi_m$ has exactly $t_1 \cdots t_n$ satisfying assignments (we use disjoint sets of variables). This is odd iff each t_i is odd.
2. Let $x_1, \ldots, x_n$ be the variables of ϕ_1. Then $(\phi_1 \vee y) \wedge ((x_1 \vee \neg y) \wedge \cdots \wedge (x_n \vee \neg y))$ has one satisfying assignment more than ϕ_1. (Check this!) This formula is not in 3-CNF, though, but there is a parsimonious reduction from #SAT to #3SAT.
3. Follows from 1. and 2. and de Morgan's law. ∎

Proof of Theorem 20.3. Given a formula ϕ in 3-CNF, there is a probabilistic polynomial time algorithm that computes formulas $\psi_0, \ldots, \psi_n$ such that:

- If ϕ is satisfiable, then there is an i such that ψ_i has a unique satisfying assignment with probability $\geq 1/8$.

- If ϕ is not satisfiable, then no ψ_i is satisfiable.

Now take the function h from Lemma 20.4: If the formula ϕ is satisfiable, then $h(\psi_0, \ldots, \psi_n) \in \oplus\mathtt{3SAT}$ with probability 1/8. If ϕ is not satisfiable, then $h(\psi_0, \ldots, \psi_n) \notin \oplus\mathtt{3SAT}$. Since 3SAT is NP-complete and $\oplus\mathtt{3SAT} \in \oplus\mathsf{P}$, this completes the proof. ■

Next, we strengthen Theorem 20.3.

Theorem 20.5 $\exists\oplus\mathsf{P} \subseteq \mathsf{BP}\text{-}\oplus\mathsf{P}$.

Proof. The following problem is $\exists\oplus\mathsf{P}$ complete. Given a formula in 3-CNF in variables $y_1, \ldots, y_n$ and $z_1, \ldots, z_m$, is there an assigment to $y_1, \ldots, y_n$ such that the resulting formula has an odd number of satisfying assignments to $z_1, \ldots, z_m$?

Exercise 20.3 *Prove this!*

We now use essentially the same theorem as in the ValiantVazirani theorem, but with one exception: We will require that $h(y) = 0$, where h is the hash function (and not the function h of Lemma 20.4), i.e, only the assignments to the y's is hashed to zero. Let ϕ be the given input formula and let $\psi_0, \ldots, \psi_n$ be the resulting formulas.

Assume that ϕ has an assignment to the y's such that there is an odd number of satisfying assignments to the z's. Then there is an index i such that with probability 1/8, ψ_i has exactly one assigments to the y's such that there is an odd number of satisfying assignments to the z's. Then the total number of satisfying assignments of ψ_i is odd, since for all other assignments to the y's, there is an even number of satisfying assignments to the z's. Hence, $h(\psi_0, \ldots, \psi_n) \in \oplus\mathtt{3SAT}$.

If ϕ has no assignments to the y's such that there is an odd number of satisfying assignments to the z's, then every ϕ_i has an even number of satisfying assignments, no matter what. Hence, $h(\psi_0, \ldots, \psi_n) \in \oplus\mathtt{3SAT}$. ■

Finally, we show that an $\oplus\mathsf{P}$ oracle does not add any power to $\oplus\mathsf{P}$. In particular, this show that $\mathsf{BP}\text{-}\oplus\mathsf{P}$ allows probability amplification.

Theorem 20.6 $\oplus\mathsf{P}^{\oplus\mathsf{P}} = \oplus\mathsf{P}$.

Proof overview: Instead of asking the oracle queries directly, we store the queries and guess the answers. After the computation, we have to verify the answers. Using the functions f and g of Lemma 20.4, we can verify all the queries by just one query. We can have one query for free after the simulation by running a Turing machine for the oracle.

Proof. We only have to prove the $\subseteq$-direction. Let $L \in \oplus\mathsf{P}^{\oplus\mathsf{P}}$ Let M be a nondeterministic polynomial time Turing machine such that for all x, $M^{\oplus\mathtt{3SAT}}$ has an odd number of satisfying assignment iff $x \in L$.

We have to get rid of the oracle. Since M is polynomial time bounded, it asks at most $p(n)$ queries for some polynomial p. We simulate M as follows:

Input: x

1. Guess the answers to the queries.

2. Simulate M.
   Whenever M wants to query the oracle, use the guessed answer instead and store the query of M on an extra tape.

3. If at some point, M rejects, reject, too.

4. If M accepts, test whether we guessed the right queries:
   This can be done via the functions f and g from Lemma 20.4. If the guessed answer to a query ϕ was yes, then we have to test whether $\phi \in \oplus\mathtt{3SAT}$. If it was no, we test whether $g(\phi) \in \oplus\mathtt{3SAT}$. All the tests can be done together via the function f. Let ψ the formula that we get in this way.

5. Simulate a Turing machine for $\oplus\mathtt{3SAT}$ to test whether $\psi \in \oplus\mathtt{3SAT}$. Accept, if this Turing machine accepts, otherwise reject.

The computation tree of the simulation consists of many subtrees, each one corresponding to one sequence of guessed answers. If the guesses are wrong, then this subtree always has an even number of accepting path, since to each accepting path of M, we append a tree that verifies the guessed answers.

Let $x \in L$. Consider the subtree that corresponds to the right guesses. It has an odd number a of accepting paths (of M). To each accepting path, we append a tree where we check our guesses. Since we guessed right, each of the appended trees has an odd number b of accepting path. The total number of accepting paths is ab which is odd. In all other subtrees, the number of accepting path is even. Thus the total number is odd.

If $x \notin L$, then a above is even. Thus ab is even, too, and so is the total number of accepting paths. ∎

Corollary 20.7 $\mathsf{BP}\text{-}\oplus\mathsf{P}$ *allows probability amplification.*

Proof. The amplification procedure in Lemma 15.4 is easily seen to be in $\oplus\mathsf{P}^{\oplus\mathsf{P}}$. Thus it is in $\oplus\mathsf{P}$. ∎

20.2 Toda's theorem

We start with the following result.

Theorem 20.8 $\mathsf{PH} \subseteq \mathsf{BP}\text{-}\oplus\mathsf{P}$.

Proof. We will prove the following claim: For all k, $\Sigma_k^{\mathsf{P}} \cup \Pi_k^{\mathsf{P}} \subseteq \mathsf{BP}\text{-}\oplus\mathsf{P}$. The proof is by induction in k.
Induction base: The case $k = 0$ is clear.
Induction step: Now assume that the claim is valid for some k. We have to prove it for $k+1$. Since $\mathsf{BP}\text{-}\oplus\mathsf{P}$ is closed under complementation (check this!) is is sufficient to proof that $\Sigma_{k+1}^{\mathsf{P}} \in \mathsf{BP}\text{-}\oplus\mathsf{P}$. Let $L \in \Sigma_{k+1}^{\mathsf{P}} = \exists\Pi_k^{\mathsf{P}}$. By the induction $L \in \exists\,\mathsf{BP}\text{-}\oplus\mathsf{P}$. Since $\mathsf{BP}\text{-}\oplus\mathsf{P}$ allows probability amplification, $L \in \mathsf{BP}\text{-}\exists\oplus\mathsf{P}$ by Lemma 15.5. By Theorem 20.5, $L \in \mathsf{BP}\text{-}\,\mathsf{BP}\text{-}\oplus\mathsf{P}$. Thus $L \in \mathsf{BP}\text{-}\oplus\mathsf{P}$ by Exercise 15.3. ∎

Exercise 20.4 *Show that if* C *is closed under complementation, then* $\mathsf{BP}\text{-}\mathsf{C}$ *is closed under complementation.*

Theorem 20.9 $\mathsf{BP}\text{-}\oplus\mathsf{P} \subseteq \mathsf{P}^{\#\mathsf{P}}$.

Proof. Let $A \in \mathsf{BP}\text{-}\oplus\mathsf{P}$. Then there is some language $B \in \mathsf{P}$ such that if $x \in A$, then for a fraction of at least $2/3$ of all y's there is an odd number of z such that $\langle x, y, z\rangle \in B$, where y and z are polynomially bounded. And if $x \notin A$, then for a fraction of at most $1/3$ of the y's there is an odd number of z such that $\langle x, y, z\rangle \in B$.

Now consider a nondeterministic Turing machine M that consists of three stages: It first guesses a y, then a z, and finally accepts iff $\langle x, y, z\rangle \in B$. Let p be the running time of M. Then M has w.l.o.g. at most $2^{p(|x|)}$ computation paths.

Let $a(x, y)$ denote the number of accepting paths that M has on input x in the subtree that corresponds to a particular y guessed in the first stage. Next we modify M as follows: Whenever the test $\langle x, y, z\rangle \in B$ is positive, we guess a new z and repeat. We do this $p := p(|x|)$ times. Then we add an artifical accepting path and repeat the whole process another p times. Thus in the subtree corresponding to y, the number of accepting paths is now $(a(x, y)^p + 1)^p$. Let the resulting Turing machine be N. N is a polynomial time nondeterministic Turing machine. Thus it defines a function acc_N in $\#\mathsf{P}$.

What is this good for? We claim that the following holds:

$$a(x, y) \text{ odd} \Longrightarrow (a(x, y)^p + 1)^p = 0 \mod 2^p$$
$$a(x, y) \text{ even} \Longrightarrow (a(x, y)^p + 1)^p = 1 \mod 2^p$$

Let $a := a(x, y)$. If a is even, then

$$\begin{aligned} a &= 0 \mod 2 \\ a^p &= 0 \mod 2^p \\ a^p + 1 &= 1 \mod 2^p \\ (a^p + 1)^p &= 1 \mod 2^p \end{aligned}$$

If a is odd then

$$\begin{aligned} a &= 1 \mod 2 \\ a^p &= 1 \mod 2 \\ a^p + 1 &= 0 \mod 2 \\ (a^p + 1)^p &= 0 \mod 2^p. \end{aligned}$$

Thus in order see whether $x \in A$ in $\mathsf{P}^{\#\mathsf{P}}$, we query $\mathrm{acc}_N(x)$ and reduce this value modulo $2^{p(|x|)}$. We have

$$\begin{aligned} \mathrm{acc}_N(x) &= \sum_y a(x, y) \\ &= |\{y \mid \langle x, y, z\rangle \in B \text{ for an even number of } z\text{'s}\}| \mod 2^{p(|x|)}. \end{aligned}$$

Thus, if this reduced value is at most a fraction of $1/3$ of all possible y's, then we accept. Otherwise, we reject. The total number of all y's is $2^{q(n)}$ for some polynomial q. ∎

Remark 20.10 *1. Note that we do not need the probability gap of the* BP*-operator. We could replace the threshold of* $1/3$ *safely by* $1/2$ *in the last step of the proof.*

2. We can replace the query to $\#\mathsf{P}$ *by a query to* PP*, i.e.,* $\mathsf{BP}\text{-}\oplus\mathsf{P} \subseteq \mathsf{P}^{\mathsf{PP}}$*, since we are essentially only interested in the highest bit of the* $\#\mathsf{P}$*-function.*

21 Interactive proofs (and zero knowledge)

21.1 Interactive proofs

Languages $A \in \mathsf{NP}$ have the property that they have polynomially long proofs of membership, that is, for all $x \in A$ there is a polynomially long proof that shows "$x \in A$" and for all $x \notin A$, there is no such proof for the fact "$x \in A$".

One can see this as a game between two players, a prover and a verifier. The prover wants to convince the verifier that "$x \in A$". For NP, this game is easy. The prover sends the proof to the verifier and the verifier checks it in polynomial time. Now we make this process *interactive*: The verifier may also send information to the prover ("ask questions") and the prover may send answers several times.[1]

Formally, an *interactive proof system* consists of two Turing machines, a *prover* P and a *verifier* V. These two machines share a read-only input tape and a communication tape. The verifier is a polynomial time probabilistic Turing machine. The prover is computationally unbounded but it must halt on all inputs and is only allowed to write strings of polynomially length on the communication tape.

The protocol proceeds in rounds: On the verifier's turn, it runs for a polynomial number of steps and finally writes some string on the communication tape and enters a special state. Then the prover takes over and computes as long as he wants and finally writes a polynomially long string on the communication tape. Then it is again the verifier's turn and so on until the verifier finally accepts or rejects. In the first case, we write $(P,V)(x) = 1$, in the second case $(P,V)(x) = 0$. The number of rounds, i.e, the number of times the control changes between the prover and the verifier, is always polynomial in $|x|$.

Definition 21.1 *A pair (P,V) as above is an interactive proof system for a language A if*

1. *for all $x \in A$, $\Pr[(P,V)(x) = 1] \geq 2/3$ and*
2. *for all $x \notin A$, $\Pr[(\hat{P},V)(x) = 1] \leq 1/3$ for all provers $\hat{P}$.*

[1] Such situations arise for instance in cryptography where one person has to convince an other person that some information is correct but both persons do not trust each other.

Above, the probability is taken over the random strings of V.

If $x \in A$, then there is a prover P that convinces V to accept with probability $\geq 2/3$. If $x \notin A$, then no prover $\hat{P}$ can make V accept with probability $> 1/3$.

21.2 Examples

Example 21.2 *As already mentioned, every language $A \in \mathsf{NP}$ is in IP, too. Since $\mathsf{NP} = \exists\mathsf{P}$, there is a language $B \in \mathsf{P}$ such that for all x, $x \in A$ iff there exists a polynomially long y such that $\langle x, y\rangle \in B$. The following IP protocol is an interactive proof for A:*

1. *The prover uses exhaustive search to find a y such that $\langle x, y\rangle \in B$. If he finds one, then he sends such a y to the verifier. If he does not find such a y, then he send anything.*

2. *The verifier checks whether $\langle x, y\rangle \in B$. If it is, then he accepts, otherwise he rejects.*

It is clear that the protocol is correct. Since the prover is computationally unbounded he has all the time in the world to find a proof y if it exists. Note that the verifier does not need any randomization.

Two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* there is a bijective mapping $\pi : V_1 \to V_2$ such that

$$\text{for all } u, v \in V,\ u \neq v\text{: } \{u, v\} \in E_1 \iff \{\pi(u), \pi(v)\} \in E_2.$$

Graph isomorphism GI is the following problem: Given a pair of graphs $\langle G_1, G_2\rangle$, decide whether they are isomorphic. GI is obviously in NP, a mapping π is a polynomially long proof that G_1 and G_2 are isomorphic.

Graph nonisomorphism $\overline{\text{GI}}$, the complement of GI, has an IP protocol. This is an interesting fact, as we do not know that $\overline{\text{GI}} \in \mathsf{NP}$ (or $\text{GI} \in \mathsf{co\text{-}NP}$):

Example 21.3 *Here is an IP protocol for $\overline{\text{GI}}$: The input are two graphs G_1 and G_2 both with n nodes. (If they have a different number of nodes, then they cannot be isomorphic.) We can assume that the set of nodes of G_1 and G_2 is $\{1, \ldots, n\}$.*

1. *The verifier chooses a random $i \in \{1, 2\}$ and a random permutation of $\{1, \ldots, n\}$. (How do you guess a random permutation cleverly?). The verifier now sends $H := \pi(G_i)$ to the prover, where $\pi(G_i)$ is the graph that one gets when replacing each edge $\{u, v\}$ by $\{\pi(u), \pi(v)\}$.*

2. *The prover now checks whether G_1 and H are isomorphic or G_2 and H are isomorphic (by running through all permutations) and sends a j such that G_j and H are isomorphic to the verifier.*

3. *The verifier accepts if $i = j$.*

If G_1 and G_2 are not isomorphic, then the j that a suitable prover will find will always be the i that the verifier sent, that is,

$$\langle G_1, G_2\rangle \in \overline{\mathrm{GI}} \Longrightarrow \textit{there is a prover } P\textit{: } \Pr[(P,V)(\langle G_1, G_2\rangle) = 1] = 1.$$

If G_1 and G_2 are isomorphic, then no matter what the prover will do, he has just a chance of $1/2$ that $i = j$, that is,

$$\langle G_1, G_2\rangle \notin \overline{\mathrm{GI}} \Longrightarrow \textit{for all provers } \hat{P}\textit{: } \Pr[(\hat{P},V)(\langle G_1, G_2\rangle) = 1] \leq 1/2.$$

We can bring down the probability $1/2$ to $1/4$ running the protocol a second time.

Remark 21.4 *In the protocol for $\overline{\mathrm{GI}}$ above, it is crucial that the random tape of the verifier is* private*, i.e, the prover does not have access to it. (Otherwise, the prover would know i.) But one can get rid of this restriction, more general, one can show that whenever there is an IP protocol for some language A with private random tape, then there is also one with* public *random tape which the prover can read.*

Remark 21.5 GI*, by the way, is a candidate for a problem in $\mathsf{NP} \setminus \mathsf{P}$ that is not NP-complete. In fact, it can be shown that if* GI *was NP-complete, then the polynomial time hierarchy would collapse.*

Excursus: Zero knowledge

Look at the following protocol for GI, where we assume that the prover is a probabilistic Turing machine, too.:

1. The prover randomly guesses an $i \in \{1,2\}$ and a permutation $\pi \in S_n$. He sends the graph $H := \pi(G_i)$ to the verifier.
2. The verifier now randomly selects a $j \in \{1,2\}$ and sends j to the prover.
3. The prover now computes a $\tau \in S_n$ such that $\tau(G_j) = H$ and sends τ to the verifier.
4. The verifier accepts if $\tau(G_j) = H$.

If the graphs are isomorphic, then (P,V) will accept. If the graphs are not isomorphic, then i and j will differ with probability $1/2$ and no prover P' will be able to find a permutation τ such that $\tau(G_j) = H = \pi(G_i)$ in this case.

Note that prover does not need to be able to compute any isomorphism τ, it is sufficient if he knows the isomorphism σ between G_1 and G_2. If $i = j$, then $\tau = \pi$. If $i \neq j$, then $\tau = \pi \circ \sigma$ or $\tau = \pi \circ \sigma^{-1}$.

The protocol does not reveal anything about the isomorphism (if one exists); just sending $\pi(G_j)$, j and π for random π and j would not be distinguishable from the actual communication.

Definition 21.6 (Goldwasser, Micali & Rackoff) *Let $A \in$ IP via a prover–verifier pair (P, V). Then (P, V) is called a* zero knowledge protocol *if there is a polynomial time bounded probabilistic Turing machine M such that M on any input x outputs a tuple $(m_1, \ldots, m_\ell)$ such that the distribution produced by M is exactly the distribution of the communication messages of (P, V) on x.*

Strictly speaking the definition above is "perfect zero knowledge with fixed verifier". There are other definitions of zero knowledge. Under a weaker definition, one can show that every language in NP has a zero knowledge protocol.

Zero knowledge means that the verifier cannot learn anything about the proof that $x \in A$, since the only thing he sees are the messages. But with the resources he can use, he can produce these sequences without the help of the prover.

21.3 IP ⊆ PSPACE

Even though the prover is computationally unbounded, the power of IP is bounded.

Theorem 21.7 IP ⊆ PSPACE.

Proof. Let (P, V) be an interactive proof system for a language A. Let $p(|x|)$ denote the number of rounds on inputs of length $|x|$ and let $q(|x|)$ be an upper bound for the length of the messages written by P. We construct a polynomial space bounded Turing machine M that accepts A.

Input: x

1. Systematically enumerate all possible messages $m_1, \ldots, m_{p(|x|)}$ of the prover.
2. Systematically enumerate all possible random strings for V.
   (a) Simulate V using the current random string pretending that the messages of P are $m_1, \ldots, m_{p(|x|)}$.
   (b) Count how often V accepts and rejects.
3. If V had more accepting paths than rejecting paths, then accept x. Otherwise, go on with the next sequence of messages.
4. If all sequences of messages have been tried, then reject x.

If $x \in A$, then the prover P will make V accept. In particular, for the sequence of messages that P produces, V will have more accepting path than rejecting paths (in fact at least a fraction of $2/3$ of the paths will be accepting). If $x \notin A$, then no prover $\hat{P}$ will make V accept. Thus M will not accept x, since otherwise, there will be a prover $\hat{P}$ that will convince V. (Note that $\hat{P}$ is computationally unbounded and can just simulate M to find the right set of messages that he has to send.) ■

Exercise 21.1 *Prove that if the verifier is deterministic, then any interactive proof system can only accept languages in* NP. *(Hint: Guess the communication.)*

22 Interactive proofs and PSPACE

In this chapter, we will prove the converse of Theorem 21.7, namely that PSPACE $\subseteq$ IP. In the view of Exercise 21.1, this is remarkably. It is completely unexpected that by just giving V a random string, the power of the system jumps from NP to PSPACE.

Proof overview: It is sufficient to show QBF $\in$ IP. The proof has several steps:

1. First we need to normalize the quantified Boolean formulas in a certain way.

2. Then we will *arithmetize* the formulas in a way similar to Chapter 7. Every formula F will be represented by a number a_F with a succinct representation (i.e., a polynomial size circuit C_F that computes a_F), such that $a_F = 0$ iff F is false. (Remember that the formulas F are closed, so a_F will be a number. Some intermediate a_F will however be (non-constant) polynomials.)

3. Now the prover has to convince the verifier that $a_F \neq 0$. The problem is that a_F is too large and the verifier is not able to evaluate a_F on its own. The evaluation will be done modulo a small prime and uses the structure of the formula. The prover will help the verifier with evaluating the subformulas.

Excursus: The race for IP = PSPACE

N. Nisan observed that one could exploit the techniques used in the proof of Toda's theorem together with arithmetization to show that PH $\subseteq$ IP. This started a race in December 1989 to finally show that IP = PSPACE, which was finally proven by A. Shamir. You can read about this—at least for computer science standards—exciting race in the following article by László Babai. The proceedings should be available in the library.

L. Babai. E-mail and the unexpected power of interaction. In *Proc. 5th IEEE Structure in Complexity Theory Conference*, 1990, 30–44.
(Nowadays, the conference is called IEEE Computational Complexity Conference.)

22.1 Simple quantified formulas

We will show that QBF $\in$ IP. Since QBF is PSPACE-complete, the result follows. First we will define a restricted version of QBF.

Definition 22.1 *A quantified Boolean formula is called* simple *if*

1. *all negations are only applied to variables and*
2. *for every occurrence of a variable x, the number of universal quantifiers between the occurrence and the binding quantifier of x is at most one.*

Let QBF$'$ be the set of all $F \in$ QBF that are simple.

Lemma 22.2 QBF $\leq_{\mathrm{P}}$ QBF$'$

Proof. Let F be a closed quantified Boolean formula. We first move all negation to the variables by using De Morgan's laws and the rules $\neg\exists G = \forall\neg G$ and $\neg\forall G = \exists\neg G$.

For the second property, we scan the formula from the outside to the inside stopping at each universal quantifier. Assume that the current formula is $\forall x G(x, y_1, \ldots, y_\ell, \ldots)$ and $y_1, \ldots, y_\ell$ are these variables of G that are bounded outside of G, i.e, the variables that are affected by the universal quantifier. We replace the current formula by the following equivalent formula:

$$\forall x \exists y'_1 \ldots \exists y'_\ell : (\bigwedge_{\lambda=1}^{\ell} y_\lambda \leftrightarrow y'_\lambda) \wedge G(x, y'_1, \ldots, y'_\ell, \ldots).$$

Then we go on with G. Since we go from the outside to the inside of the formula, it is easy to see that $\forall x$ is the only universal quantifier between $y_1, \ldots, y_\ell$ and their binding quantifiers.

The transformation is polynomial time computable and a many-one reduction from QBF to QBF$'$. ∎

22.2 Arithmetization

We will map simple quantified Boolean formulas F to arithmetic circuits C_F that compute polynomials a_F with the property that for all closed formulas

$$F \text{ is true} \iff a_F \neq 0.$$

Our circuits will have two new operations, namely $\sum_x$ and $\prod_x$ where x is a variable. These two new gates will have fanin one. If C is a circuit whose top gate is $\sum_x$ or $\prod_x$, then C computes $P|_{x=0} + P|_{x=1}$ or $P|_{x=0} \cdot P|_{x=1}$,

respectively. Here, P is the polynomial computed by the subcircuit of C that is obtained by removing the top gate of C and $P|_{x=0}$ and $P|_{x=1}$ are the polynomials that we get when replacing x by 0 and 1, respectively.[1]

We define a_F inductively. This implicitly will define C_F, too.

1. If $F = x$, then $a_F = x$. [2]
2. If $F = \neg x$, then $a_F = 1 - x$.
   (Remember that negations are only applied to variables.)
3. If $F = G \wedge H$, then $a_F = a_G \cdot a_H$.
4. If $F = G \vee H$, then $a_F = a_G + a_H$.
5. If $F = \exists x G$, then $a_F = \sum_x a_G$.
6. If $F = \forall x G$, then $a_G = \prod_x a_F$.

Exercise 22.1 *Show by structural induction that for all closed simple quantified Boolean formulas F: F is true iff $a_F \neq 0$.*
It might be easier to show the following more general statement: If F is a simple quantified Boolean formula with k free variables. Then for all $\xi_1, \ldots, \xi_k \in \{0,1\}$, $F(\xi_1, \ldots, \xi_k)$ is true iff $a_F(\xi_1, \ldots, \xi_k) \neq 0$.

Lemma 22.3 *Let F be a simple quantified formula F and let x be a variable of F. Then $\deg_x a_F \leq 2\ell$ where ℓ is the length of F.*

Proof. Since F is simple, x is in the scope of at most one universal quantifier. This corresponds to the $\prod_x$ operation which can double the degree. If $F = G \wedge H$, then the degrees of a_G and a_H add up, but the size of G and H together is less than the size of F. All other operations do not change the degree. ∎

22.3 The protocol

The input is a closed simple quantified Boolean formula F of length n. It is easier to consider a more general task: Given a simple quantified Boolean formula F of length n, and a circuit D_F that is obtained from C_F be replacing some of the free variables by constants from $\{1, \ldots, 2^{O(n)}\}$, convince the verifier that the value of D_F is not zero.

[1] Instead of introducing this new operation, one could just take two copies of the subcircuit of C and replace in one of them x by 0 and in the other one x by 1 and add or multiply this results. This however yields an exponential blow up in the description which we cannot afford, since then the verifier could not even read the circuit. But note that it is not clear how to evaluate such a circuit in polynomial time, since one still has to evaluate the subcircuit twice. So the verifier still needs the help of the prover.

[2] We do not distinguish between Boolean and integer variables here. You are old enough.

1. The prover evaluates D_F. Let d_F be the value. He selects a prime number $k \in \{2^n, \ldots, 2^{2n}\}$ such that if $d_F \neq 0$, then k does not divide d_F. If $d_F = 0$, then he can select any prime number k in the given range. The prover sends k and $\hat{d} = d_F \mod k$ to the verifier.

2. The verifier now checks whether k is prime.[3] He rejects if k is not prime.

3. Now the prover convinces the verifier that $d_F = \hat{d} \mod k$.

   (a) If $F = x$ or $F = \neg x$, then the verifier can check this on its own.

   (b) If $F = G \vee H$ or $F = G \wedge H$, then the prover goes on recursively.

      i. He computes the values e and f of D_G and D_H, reduces them mod k and send the values $\hat{e}$ and $\hat{f}$ to the verifier.

      ii. The verifier now checks whether $\hat{e} + \hat{f} = \hat{d} \mod k$ or $\hat{e} \cdot \hat{f} = \hat{d} \mod k$, respectively.

      iii. If this is true, the prover now recursively has to convince that indeed $\hat{e} = e \mod k$ and $\hat{f} = f \mod k$.

   (c) $F = \exists x G$ and $F = \forall x G$ are the interesting cases. We only treat the first one, the second one is treated in a similar manner. The circuit D_G that corresponds to G computes a univariate polynomial P in x of degree $\delta \leq 2n$.

      i. The prover computes the coefficients of P, reduces them mod k, and sends the reduced coefficients $\hat{a}_0, \ldots, \hat{a}_\delta$ to the verifier.

      ii. The verifier now computes $P(0) = \hat{a}_0 \mod k$ and $P(1) = \hat{a}_0 + \cdots + \hat{a}_\delta \mod k$ and checks whether $P(0) + P(1) = \hat{d} \mod k$.

      iii. Now the prover has to convince the verifier that $\hat{a}_0, \ldots, \hat{a}_\delta$ are indeed the coefficients of $P \mod k$. Since we are working over $\mathrm{GF}(k)$, which is a field of size $2^{\Omega(n)}$, and the degree of P is bounded by $\mathrm{O}(n)$, the event that $P(z) = a'_0 + a'_1 z + \cdots + a'_\delta z^\delta \mod k$ holds for a random $z \in \mathrm{GF}(k)$ but $a'_0, \ldots, a'_\delta$ are not the coefficients of $P \mod k$ is tiny. Thus the verifier chooses a random $z \in \mathrm{GF}(z)$ and sends it to the verifier.

      iv. Now the verifier has to convince that the value of D_G, where D_G is the circuit C_G with the free variable x replaced by the constant z, is indeed $\hat{a}_0 + \hat{a}_1 z + \ldots \hat{a}_\delta z^\delta$. This is done recursively.

4. The verifier finally accepts if he accepted all of the subtasks.

[3] This can be done in polynomial time with the algorithm by Agrawal, Kayal, and Saxena. Another method is having the verifier selecting a prime at random. If this is done appropriately, then, with high probability, k does not divide d_F.

Theorem 22.4 $\mathsf{PSPACE} \subseteq \mathsf{IP}$.

Proof. $\mathtt{QBF}'$ is PSPACE-complete, so it suffices to show $\mathtt{QBF}' \in \mathsf{IP}$. If $F \in \mathtt{QBF}'$, then the prover–verifier pair above will accept C_F with probability 1. The only problem is whether we can find a prime k. d_F is bounded by 2^{2^n} (see Chapter 16). Thus there are at most 2^n many primes that divide d_F. By the prime number theorem, there are that many primes between 2^n and 2^{2n}.

It remains to bound the probability that if $F \notin \mathtt{QBF}$, a prover $\hat{P}$ can convince the verifier. The only place where a prover can cheat and the verifier does not detect this is when in step 3.c.iii, the prover can convince the verifier that a wrong sequence of coefficients are the coefficients of P mod k. But since P has degree $2n$ but z is drawn from a set of size $\geq 2^n$, this can happen with probability at most $2n/2^n$. The probability that in any round an error happens is then $2n^2/2^n$ which tends to zero. ∎

23 Graph isomorpishm

Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. A *graph isomorphism* from G to G' is a bijection $\pi : V \to V'$ such that for all pairs of vertices u and v,

$$\{u, v\} \in E \text{ if and only if } \{f(u), f(v)\} \in E'.$$

In other words, G and G' are the same graphs up to renaming the nodes. We say that G and G' are *isomorphic*. An isomorphism from G to G is called an *automorphism*. $\mathrm{Aut}(G)$ denotes the set of all automorphisms of G.

The graph isomorphism problem is defined as

$$\mathrm{GI} = \{\langle G, G' \rangle \mid G \text{ and } G' \text{ are isomorphic}\}.$$

The subgraph isomorphism problem SubGI generalizes GI:

$$\mathrm{SubGI} = \{\langle G, G' \rangle \mid G \text{ is isomorphic to a subgraph of } G'\}.$$

We will show that it is very unlikely that GI is NP-complete.

Theorem 23.1 *If* GI *is* NP*-complete, then* $\mathsf{PH} = \Sigma_2^{\mathsf{P}} = \Pi_2^{\mathsf{P}}$.

On the other hand, we do not know whether GI is in P or BPP. SubGI, however, is NP-complete.

Exercise 23.1 *Show that* SubGI *is* NP*-complete.*

The proof of Theorem 23.1 will be done in two steps:

1. First we will show that $\overline{\mathrm{GI}} \in \mathsf{BP}\cdot\mathsf{NP}$. If GI were NP-complete and consequently, $\overline{\mathrm{GI}}$ were co-NP-complete, this would imply $\mathsf{co}\text{-}\mathsf{NP} \subseteq \mathsf{BP}\cdot\mathsf{NP}$.
2. Second we will show that $\mathsf{co}\text{-}\mathsf{NP} \subseteq \mathsf{BP}\cdot\mathsf{NP}$ implies $\mathsf{PH} = \Sigma_2^{\mathsf{P}} = \Pi_2^{\mathsf{P}}$.

Lemma 23.2 *There is a language* $A \in \mathsf{NP}$ *and a polynomial* p *such that for all graphs* G *and* G' *with* n *nodes:*

1. *If* G_1 *and* G_2 *are isomorphic, then* $|\{y \in \{0,1\}^{\leq p(n)} \mid \langle G_1, G_2, y \rangle \in A\}| = (n!)^3$.
2. *If* G_1 *and* G_2 *are not isomorphic, then* $|\{y \in \{0,1\}^{\leq p(n)} \mid \langle G_1, G_2, y \rangle \in A\}| = 8 \cdot (n!)^3$.

Proof. We first interpret y as a graph $G = (V, E)$ with n nodes and a mapping $f : V \to V$. $\langle G_1, G_2, y\rangle$ is accepted if G is isomorphic to G_1 or isomorphic to G_2 and $f \in \mathrm{Aut}(G)$, see the next lemma.

If G_1 and G_2 are isomorphic, then there are $n!/k$ graphs that are isomorphic to G_1 where $k = \mathrm{Aut}(G_1)$. Thus there are $n!$ pairs (G, f) such that G is isomorphic to G_1 (and hence to G_2) and $f \in \mathrm{Aut}(G)$.

If G_1 and G_2 are not isomorphic, then there are $n!$ pairs that are isomorphic to G_1 and $n!$ different pairs that are isomorphic to G_2 yielding $2n!$ pairs in total.

Now we just interpret y not as one pair (G, f) but as three pairs $(G, f), (G', f'), (G'', f'')$ such that each of the three graphs is isomorphic to G_1 or G_2 and each of the three functions if a automorphims of the corresponding graph.

If G_1 and G_2 are isomorphic, we have there $(n!)^3$ y's and otherwise, there are $8(n!)^3$ y's. A is just the language of all $\langle G_1, G_2, y\rangle$ such that y has the property above. A is in NP since we just have to guess three isomorphisms to check whether G, G', and G'' are isomorphic to one of G_1 and G_2. ∎

Lemma 23.3 *Let G be a graph with n nodes and $\mathrm{Aut}(G) = k$. Then there are $n!/k$ different graphs isomorphic to G.*

Proof. We can order the set of nodes of G in $n!$ ways. For any one such ordering, there are another $k - 1$ orderings such that the resulting graph is the same. Thus there are $n!/k$ different graphs.[1] ∎

Let H be a set of pairwise independent hash functions $\{0,1\}^n \to \{0,1\}^{m+1}$ with $2^m \le 8 \cdot (n!)^3 \le 2^{m+1}$. Let $Y = \{y \mid \langle G_1, G_2, y\rangle \in A\}$.

- If G_1 and G_2 are isomorphic, then

$$\Pr_{h\in H}[\exists y : h(y) = 0] \le \sum_{y\in Y} \Pr[h(y) = 0] = |Y|/2^{m+1} \le 1/8$$

- If G_1 and G_2 are not isomorphic, then

$$\begin{aligned}
\Pr_{h\in H}[\exists y : h(y) = 0] &\ge \sum_{y\in Y} \Pr[h(y) = 0] - \sum_{y,z\in Y, y\neq z} \Pr[h(y) = 0 \wedge h(z) = 0] \\
&= \frac{|Y|}{2^{m+1}} - \frac{\frac{1}{2}|Y|(|Y|+1)}{(2^{m+1})^2} \\
&= \frac{|Y|(2^{m+1} - \frac{1}{2}|Y| - \frac{1}{2})}{(2^{m+1})^2} \\
&\ge \frac{2^m(2^{m+1} - 2^m - \frac{1}{2})}{(2^{m+1})^2} \\
&\ge \frac{1}{5}
\end{aligned}$$

[1] $\mathrm{Aut}(k)$ acts on the set of all graphs that we get from G by relabeling the nodes. The lemma now follows from Lagrange's theorem.

for $m \geq 2$. The first inequality follows from the inclusion-exclusion principle, the second equality from the fact that H is pairwise independent.

Now let $B = \{\langle G_1, G_2, h\rangle \mid \exists y : \langle G_1, G_2, y\rangle \in A \wedge h(y) = 0\}$. The predicate on the right hand side of the definition of B is in NP. The calculations above show that $\overline{\mathrm{GI}} \in \mathsf{BP}\text{-}\,\mathsf{NP}$.

Theorem 23.4 $\overline{\mathrm{GI}} \in \mathsf{BP}\text{-}\,\mathsf{NP}$.

The second step of the proof of Theorem 23.1 is done by "operator magic".

Lemma 23.5 *If* $\mathsf{co}\text{-}\mathsf{NP} \subseteq \mathsf{BP}\text{-}\,\mathsf{NP}$, *then* $\mathsf{PH} = \Sigma_2^{\mathsf{P}} = \Pi_2^{\mathsf{P}} = \mathsf{BP}\text{-}\,\mathsf{NP}$.

Proof. $\mathsf{BP}\text{-}\,\mathsf{NP}$ allows probability amplification. Thus,

$$\Sigma_2^{\mathsf{P}} = \exists \mathsf{co}\text{-}\mathsf{NP} \subseteq \exists\, \mathsf{BP}\text{-}\,\mathsf{NP} \subseteq \mathsf{BP}\text{-}\,\exists \mathsf{NP} = \mathsf{BP}\text{-}\,\mathsf{NP} \subseteq \Pi_2^{\mathsf{P}}. \quad \blacksquare$$

24 Arthur–Merlin games & interactive proofs

24.1 Arthur–Merlin protocols

In the protocol for graph non-isomorphism (Example 21.3), it is crucial that the prover has no access to the verifier's random bits. But what happens if the prover also gets access to the random string? Remark 21.4 says, somewhat surprisingly, that this does not help the prover. The goal of this chapter is to formalize and to prove this. (Although we will not give a full proof.)

Let us first formalize what it means for an interactive proof system that the prover knows the random bits. To distinguish between private random bits (Definition 21.1) and public random bits, proof systems using the latter are called *Arthur–Merlin protocols*: Arthur is the (polynomial-time bounded probabilistic) verifier and Merlin is the (powerful) prover.

Definition 24.1 (Arthur–Merlin protocol) *A k-round interactive proof with public random coins or* k-round Arthur–Merlin protocol *is an interactive proof system with a prover of unlimited computational power (here called* Merlin*) and a verifier that is polynomial-time restricted and probabilistic (called* Arthur*).*

In the ith round for odd i, Arthur generates a random string of polynomial length and sends it to Merlin. In the ith round for even i, Merlin sends a message to Arthur. Finally, Arthur decides (in deterministic polynomial time) to accept or reject. Arthur is not allowed to use any randomness besides the random strings he sends to Merlin.

Let $\mathsf{AM}[k]$ *be the set of languages that can be decided by a k-round Arthur–Merlin game with error probability bounded by* $1/3$*. Let* $\mathsf{AM} = \mathsf{AM}[2]$*.*

Since Arthur must send all his random strings to Merlin, Arthur–Merlin games correspond to interactive proofs with public coins.

Similar to $\mathsf{AM}[k]$, we denote by $\mathsf{IP}[k]$ the set of languages that can be decided by an interactive protocol (with private coins) that needs at most k rounds of interaction. Note that $\mathsf{AM} = \mathsf{AM}[2]$, whereas $\mathsf{IP} = \mathsf{IP}[\mathrm{poly}]$.

24.2 Graph non-isomorphism revisited

From the definitions of Arthur–Merlin games and interactive proof systems, we immediately have $\mathsf{AM}[k] \subseteq \mathsf{IP}[k]$. (Merlin does not need to get any extra information from Arthur in addition to the random strings: Since he has arbitrary computational power, he can just simulate Arthur.) Somewhat surprisingly, replacing private coins by public coins is possible at the expense of two extra rounds of interaction.

Theorem 24.2 *Let $k : \mathbb{N} \to \mathbb{N}$ be any function with $k(n)$ computable in time poly(n). Then*

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

We will not give the full proof of this theorem here. Instead, we will revisit the graph non-isomorphism problem. In Example 21.3, it was crucial that the prover had no access to the verifier's random bits.

Theorem 24.3 $\overline{\mathrm{GI}} \in \mathsf{AM}[k]$ *for some constant $k \in \mathbb{N}$.*

In the following, we are going to prove this theorem. The key idea to prove this is to rephrase $\overline{\mathrm{GI}}$: Consider the set $S = \{H \mid G_1 \equiv H \vee G_2 \equiv H\}$. Roughly speaking, the size of this set S depends on whether G_1 and G_2 are equivalent: If they are equivalent, then the cardinality of S is $n!$. If they are not, then $|S| = 2n!$. (Strictly speaking, G_1 or G_2 may have less than $n!$ equivalent graphs. We can cope with this by changing the definition of the set S to $S = \{(H, \pi) \mid H \equiv G_1 \vee H \equiv G_2 \text{ and } \pi \in \mathrm{aut}(H)\}$, where $\mathrm{aut}(H)$ is the automorphism group of H: $\pi \in \mathrm{aut}(H)$ if $\pi(H) = H$.) Membership in S can be certified easily by provided a suitable permutation π.

Now we have set up the stage for a more complicated looking problem: Instead of proving $G_1 \not\equiv G_2$, Merlin just has to convince Arthur that $|S| = 2n!$, i.e., that S is large. To do this, we use a set lower bound protocol.

In a *set lower bound protocol*, the prover wants to convince the verifier that a set S has a cardinality of at least K. The verifier should accept if this is indeed the case, he should reject if $|S| \leq K/2$, and he can do anything if $K/2 < |S| < K$. If we use the set S from above and set $K = 2n!$, then a set lower bound protocol is just a Arthur–Merlin protocol for $\overline{\mathrm{GI}}$.

The tool for designing a set lower bound protocol are hash functions, similar to the Valiant–Vazirani theorem (Chapter 18). In Section 18.1, pairwise independent hash functions have been introduced (Definition 18.1).

We use the following protocol:

Setup: membership in $S \subseteq \{0,1\}^m$ can be certified efficiently. Arthur and Merlin both know $K \in \mathbb{N}$. Merlin's goal is to convince Arthur of $|S| \geq K$. Arthur should reject if $|S| \leq K/2$. Let $k \in \mathbb{N}$ such that $2^{k-2} \leq K \leq 2^{k-1}$.

Arthur: Randomly pick $h : \{0,1\}^m \to \{0,1\}^k$ from a set $\mathcal{H}$ of pairwise independent hash functions. Randomly pick $y \in \{0,1\}^k$. Send h and y to Merlin.

Merlin: Try to find a $x \in S$ with $h(x) = y$. Let π_x be the certificate of $x \in S$. Send x and π_x to Arthur.

Arthur: If π_x proves $x \in S$ and $h(x) = y$, then accept. Otherwise, reject.

In the following, let $p = K2^{-k}$. If $|S| \leq K/2$, then $|h(S)| \leq K/2 = \frac{p}{2}2^k$. Thus, Arthur will accept with a probability of at most $p/2$. It remains to analyze the acceptance probability.

Lemma 24.4 *Let $\mathcal{H}$ be a family of pairwise independent hash functions $\{0,1\}^m \to \{0,1\}^k$. Let $S \subseteq \{0,1\}^m$ be any set with $|S| \leq 2^k/2$. Then*

$$\Pr_{h\in\mathcal{H},y\in\{0,1\}^k}\big(\exists x \in S : h(x) = y\big) \geq 3|S|2^{-k-2}.$$

Proof. We prove $\Pr_h(\exists x \in S : h(x) = y) \geq \frac{3}{4}p$ for any fixed y. Then it follows also for a random y. By the inclusion-exclusion principle, we have

$$\begin{aligned}
&\Pr_{h\in\mathcal{H}}\big(\exists x \in S : h(x) = y\big) \\
&\geq \sum_{x\in S} \Pr\big(h(x) = y\big) - \frac{1}{2} \sum_{x,x'\in S, x\neq x'} \Pr\big(h(x) = h(x') = y\big) \\
&\geq 2^{-k}|S| - \frac{1}{2}|S|^2 2^{-2k} = |S|2^{-k}\left(1 - \frac{|S|}{2^{k+1}}\right) \geq \frac{3}{4}p.
\end{aligned}$$

■

Altogether, Arthur accepts with a probability of at least $3p/4$ if $|S| \geq K$ and at most $p/2$ if $|S| \leq K/2$. Note that $p \in [\frac{1}{4}, \frac{1}{2}]$ is known to both Arthur and Merlin.

Given the set lower bound protocol, the protocol for $\overline{\text{GI}}$ follows easily: The set S defined for $\overline{\text{GI}}$ allows efficient certification. Arthur accepts if Merlin succeeded to convince him that $|S| \geq 2n! = K$. Using Chernoff bounds, the number of rounds necessary until both error probabilities are bounded by $1/3$ can be shown to be constant.

How does the Arthur–Merlin protocol for $\overline{\text{GI}}$ relate to the interactive proof system of Example 21.3? The set S corresponds to the set of messages that the verifier possibly sends in the protocol of Example 21.3. The Arthur–Merlin protocol can be viewed as Merlin trying to convince Arthur that his chance of convincing the private-coin verifier is large. This is also the idea behind the proof of the more general Theorem 24.2.

Unlike in the private-coin protocol, the completeness of the Arthur–Merlin protocol is not complete, i.e., smaller than 1 (the *completeness* is

the probability that the verifier accepts an input from the language, the *soundness* is the probability that the verifier rejects an input not from the language). We note that also this can be patched: There exists also an Arthur–Merlin protocol for $\overline{\mathrm{GI}}$ with perfect completeness, and also this can be generalized to arbitrary $\mathsf{AM}[k]$, maintaining a similar number of rounds.

Finally, any constant number of rounds is as good as two rounds of interaction in Arthur–Merlin protocols: $\mathsf{AM} = \mathsf{AM}[2] = \mathsf{AM}[k]$ for any fixed $k \in \mathbb{N}$.