Computational Complexity Theory

Markus Bläser, Holger Dell
Universität des Saarlandes

Draft—November 14, 2016 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 $\mathtt{SAT}$ 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 $\mathtt{LEN} = \{a^n b^n \mid n \in \mathbb{N}\}$. $\mathtt{LEN}$ 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 $\mathtt{LEN}$ 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 $(Q, \Sigma, \Gamma, \delta, q_0, Q_{\text{acc}})$. 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 $s = \mathrm{o}(\log)$, 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 $s = \mathrm{o}(\log)$ 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 $\mathrm{O}(t)$ time and $\mathrm{O}(s)$ 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 $\mathrm{O}(s(n))$, 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* $L \in \mathsf{DSpace}(\log\log n)$.

A 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 A.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.

A.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

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

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 (A.1) separated by 11s. We call this string $\text{enc}(\delta)$ Finally, we encode the whole M by

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

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 A.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 (A.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 A.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.*

A.2 Deterministic space hierarchy

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

Theorem A.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 A.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 A.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 $y = \langle e, x\rangle$ 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 A.1 *Describe a log space bounded Turing machine that checks whether the input is a correct encoding of a deterministic Turing machine.*

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

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 A.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 $y = \langle e, x \rangle$ 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. ∎

A.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 A.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 $\langle e, x \rangle$, 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 A.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 A.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 A.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 $\dots$) and think for a minute how unnatural non-constructible time or space bounds are.

A.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 A.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 A.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 A.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.*

2 Robust complexity classes

Complexity classes

Good complexity classes should have two properties:

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

To understand the robustness criterion, consider that the class P of polynomial-time computable problems is the same, no matter if we define it using polynomial-time Turing machines, polynomial-time WHILE programs, polynomial-time RAM machines, or polynomial-time C++ programs – we always get the same class of problems.

Definition 2.1

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

L and NL are classes of problems that can be decided with very little space, which is important in practice when solving problems whose instance do not fit into RAM memory. 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 polynomial-time computable problems

and usually considered to be feasible or tractable. NP characterizes many important combinatorial optimization problems – as we will see, it characterizes all problems whose solutions can be verified in polynomial time. PSPACE is the class of problems that can be decided with polynomial space, which can be considered feasible if the instances are not too large. 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 is the exponential-time analogue of NP.

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.

2.1 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, *polynomial-time many-one reductions*, was introduced by Richard Karp, who used them to define the concept of NP-hardness. When Steve Cook showed that Satisfiability is NP-complete, he defined *polynomial-time Turing reductions*, which is a more permissive kind of reduction. To define this kind of reduction formally, we introduce the following notion. 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 ($L' \leq_{\log} L$).

2.2 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 2.2 (Immerman, Szelepcsényi) *For every space constructible $s(n) \geq \log n$,*

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

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

Exercise 2.1 *Show that satisfiability is* co-NP*-hard under polynomial time Turing reductions. What about polynomial-time many-one reductions?*

Exercise 2.2 *Show that if L is* C*-hard under R many-one reductions, then $\bar{L}$ is* co-C*-hard under R many-one reductions.*

3 L and NL

3.1 Logarithmic space reductions

Polynomial-time reductions are not fine-grained enough to analyze the relationship between problems in NL. This is due to the fact that $\mathsf{NL} \subseteq \mathsf{P}$, and so every problem in NL can be solved in polynomial-time. Instead, we often use logarithmic-space many-one reductions for studying L and NL.

Exercise 3.1 *Let f and g be logarithmic space computable functions $\Sigma^* \to \Sigma^*$. Then $f \circ g$ is also logarithmic space computable.*

Recall how we compute a function $\Sigma^* \to \Sigma^*$ in logspace: We use a multi-tape Turing machine with a constant number of tapes, a read-only input tape, and a write-only output tape; crucially, only the work tapes count towards our space requirements.

Corollary 3.1 *$\leq_{\log}$ 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 3.1. 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 3.2 *Let $L \leq_{\log} L'$.*

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

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 3.1, $\chi_{L'} \circ f$ is logarithmic space computable. Thus, $L \in \mathsf{L}$.

2. This follows from a close inspection of the proof of Exercise 3.1. 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 3.3 *1. If L is NL-hard under logarithmic space many-one reductions and $L \in \mathsf{L}$, then $\mathsf{L} = \mathsf{NL}$.*

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

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 $L' \in \mathsf{NL}$. But since $L \in \mathsf{L}$, $L' \in \mathsf{L}$, too, by Lemma 3.2. Since L' was arbitrary, then claim follows. ■

3.2 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 $\mathrm{O}(\log n)$ space bounded, we can make the Turing machine $\mathrm{O}(\log n)$ space bounded by first marking the appropriate number of cells and then simulate M. Whenever M leaves the marked space, we reject.

Exercise 3.2 *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 3.3 *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 3.4 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 3.5 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 is 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.

[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.

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. ■

3.3 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 $\langle G, s, x, d, |N_d|\rangle \notin \texttt{NEGDIST}$. 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 3.6 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 3.7 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]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.

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 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 3.8 *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 3.9 *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 $v \in V$ in sequence, nondeterministically guess whether $v \in N_{i+1}$

5. If $v \in N_{i+1}$ was guessed, test whether $\langle G, s, v, i+1 \rangle \in \texttt{DIST}$.

6. If the test fails, reject, else set $c = c + 1$.

7. If $v \notin N_{i+1}$ was guessed, do the following:

8. For all $u \in V$ such that $u \neq v$ and (u, v) is an edge:

9. We run the nondeterministic algorithm in Lemma 3.7 to test $\langle G, s, u, i, |N_i| \rangle \in \texttt{NEGDIST}$

10. If it accepts, we continue the computation

11. If it rejects, we reject

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

13. 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 and continue the computation in line 10; this is because, by the induction hypothesis, M computed $|N_j|$ correctly, and so the input $\langle G, s, u, i, |N_i| \rangle$ is a valid input for NEGDIST.

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 3.10 $\overline{\texttt{CONN}} \in \mathsf{NL}$.

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 3.4 (Translation) *Complete the proof of the Immerman–Szelepcsényi Theorem by showing the following: Let s_1, s_2, and f be space constructible such that $n \mapsto s_2(f(n))$ is also space constructible and, for all n, $s_1(n) \geq \log n$, $s_2(n) \geq \log n$, and $f(n) \geq n$. If*

$$\mathsf{NSpace}(s_1(n)) \subseteq \text{co-}\mathsf{NSpace}(s_2(n))\,,$$

Then,

$$\mathsf{NSpace}(s_1(f(n))) \subseteq \text{co-}\mathsf{NSpace}(s_2(f(n)))\,.$$

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

3.4 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?

It turns out that undirected connectivity is computationally easier than directed connectivity.

Theorem 3.11 (Reingold) $\texttt{UCONN} \in \mathsf{L}$.

To appreciate this highly non-trivial result, note that, in space $O(\log n)$, we can basically only store the indices of a constant number of vertices. Now take your favourite algorithm for connectivity in undirected graphs and try to implement it with just logarithmic space. The proof of Reingold's theorem is outside the scope of this course, but we will prove that UCONN is in RL, the class of all problems that can be solved in *randomized* logspace.

Excursus: SL-complete problems

Before Reingold's result, many natural problems were known to be equivalent to UCONN under logspace reductions, and the class of these problems was called SL (which we now know to be equal to L). As a consequence of Reingold's result, these problems were shown to be computable in logspace. 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 also a compendium of SL-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 was a PhD student at University of Texas who showed that $\texttt{UCONN} \in \mathsf{DSpace}(\log n \log\log n)$[3] at the same time when Omer Reingold showed $\texttt{UCONN} \in \mathsf{L}$. This way, a remarkable result became a footnote (or an excursus).

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

4 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.

4.1 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 4.1 *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 4.2 *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 4.3 *Every Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a Boolean circuit of size $2^{\mathrm{O}(n)}$.*[1]

4.2 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 4.1 *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 $\operatorname{size}(C_i) \leq s(i)$ and $\operatorname{depth}(C_i) \leq d(i)$ 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 4.2 *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

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

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

4.3 Simulating families of circuits by Turing machines

Theorem 4.3 *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 4.4 *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 4.4 *If $L \subseteq \{0,1\}^*$ is decided by a s size bounded and t time constructible family of circuits C, then*

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

4.4 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 4.5 *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 4.6 *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 4.5 is a size efficient construction. The following result gives a depth efficient construction.

Theorem 4.7 *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 4.5 *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 4.6 *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 4.7. 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 4.5. 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 4.8 *Theorem 4.3 and 4.7 yield an alternative proof of Savitch's theorem.*

4.5 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 4.9

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

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 4.3 and 4.7 and Remark 4.4

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

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.

5 NL, NC, and P

Lemma 5.1 *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.

5.1 Circuit evaluation

Definition 5.2 *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.

Exercise 5.1 `CVAL` *is* P*-complete.*

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 5.1, unless you believe that NC = 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 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.

6.1 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 6.1 *Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function. Show that the mutlilinear polynomial $p \in \mathbb{C}[x_1, x_2, \ldots, x_n]$ that represents f exactly is indeed unique. That is, if p and p' are two polynomials with $f(x) = p(x) = p'(x)$ for all $x \in \{0,1\}^n$, then p and p' are identical as polynomials over $\mathbb{C}[x_1, \ldots, x_n]$.*

Example 6.1

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).*

6.2 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 $O(\log n)$.

2. Boolean functions that are approximated by unbounded fanin circuits of size s and depth d have degree $O(\log^{d+1} s)$.

3. `PARITY` cannot be approximated by small degree polynomials.

We start with a technical lemma.

Lemma 6.2 *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 6.3 *Let $1 > \epsilon > 0$. There is a family of polynomials P of degree $\mathrm{O}(\log(1/\epsilon) \cdot \log n)$ 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 6.2. 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 6.2, 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 6.2 *Show that $\bigwedge_{i=1}^{n} x_i$ can be approximated in the same way.*

Lemma 6.4 *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 $\mathrm{O}(\log^d(s/\epsilon) \cdot \log^d(s))$.*

Proof. We replace every OR or AND gate by a random polynomial from a family with error probability ϵ/s. Each polynomial has degree $\mathrm{O}(\log(s/\epsilon) \cdot \log(s))$, since every gate can have at most s inputs. The degree of the polynomial at the output gate is $\mathrm{O}(\log^d(s/\epsilon) \cdot \log^d(s))$, 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 6.5 *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 6.3 *The statement of Corollary 6.5 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 6.4 and do the same construction with these polynomials.*

6.3 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 6.6 *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 6.1. 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 6.7 *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 6.7 in particular shows that $\mathsf{AC}_0 \neq \mathsf{NC}_1$. We do not know whether any of the other inclusions is strict.

Exercise 6.4 *Show that* $\texttt{PARITY} \in \mathsf{NC}_1$.

7 P and NP

Recall that we defined NP as $\bigcup_c \mathsf{NTime}(n^c)$, which in turn was defined by using non-deterministic Turing machines.

7.1 The verification characterization of NP

We first discuss a very important characterization of NP, which is arguably more natural than the definition based on non-determinism: Namely, NP contains exactly those decision problems whose yes-instances have "nifty proofs", that is, proofs that are polynomially small and that can be verified in polynomial time.

Definition 7.1 *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 two properties hold for all $x \in \Sigma^*$:*

- **Completeness.** *If $x \in L$, then there is a string $c \in \{0,1\}^*$ of length $|c| \leq p(|x|)$ such that M accepts $\langle x, c\rangle$.*
- **Soundness.** *If $x \notin L$, then, for all $c \in \{0,1\}^*$ of length $|c| \leq p(|x|)$, the machine M rejects $\langle x, c\rangle$.*

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

For yes-instances $x \in L$, the string c serves as a *certificate* (or *witness* or *proof*) of the fact that x is in L. The completeness says that, whenever a yes-instance and its certificate are given, the verifier can determine efficiently that the instance is indeed a yes-instance. We do not require an analogous certificate to exist for the no-instances; however, due to the soundness condition, we do require that no purported certificate gets accepted by M even though x is a no-instance. That is, any purported proof for a no-instance $x \notin L$ must contain a mistake, and the verifier will catch this mistake.

Note that the language $V(M)$ verified by M is generally different from the language $L(M)$ of strings accepted by the Turing machine M. In particular, $L(M)$ is the binary relation that contains precisely the pairs (x, c) for which $|c| \leq p(|x|)$ holds and M accepts $\langle x, c\rangle$. The following theorem is proven in the "Theoretical Computer Science" lecture (as Theorem 26.5).

Theorem 7.2 *$L \in$ NP if and only if 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 of writing $(x, c) \in R$, we may also write $R(x, c) = 1$. Thus $R(x, c) = 1$ holds if and only if $M(\langle x, c\rangle) = 1$. 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 for all $x \in \Sigma^*$:

$$x \in L \iff \exists c \in \{0,1\}^{p(|x|)} : R(x, c) = 1. \tag{7.1}$$

The string c is the certificate for $x \in L$; in terms of non-deterministic Turing machines, c models the non-deterministic choices that lead the non-deterministic Turing machine to an accepting computation path.

7.2 NP-complete problems

NP characterizes the complexity of an abundance of relevant problems. The *satisfiability problems* form the most prominent family of NP-complete problems.

Definition 7.3

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

Because ℓSAT is just a special case of SAT, which in turn is a special case of CSAT, we have

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

We previously proved that 3SAT is NP-complete, which implies that all three problems are NP-complete. Note that 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.

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

7.3 Branching algorithms for SAT

The most straightforward *branching algorithm* for SAT works as follows: Let F be the input CNF formula and let x be a variable of F. Then we know that

the variable is either set to true or it is set to false in a satisfying assignment of F. Hence we can define the two formulas F_0 and F_1, which are obtained from F by setting x either to 0 or to 1: In F_b we set x to $b \in \{0,1\}$, and furthermore, we remove all literals that are equal to 0 and we remove all clauses that are already satisfied because they contain a literal set to 1. Then F is satisfiable if and only if F_0 is satisfiable or F_1 is satisfiable. Note that, in any reasonable encoding, the length of F_0 and F_1 is smaller than the length of F. If we recursively apply this algorithm to F_0 and F_1, we end up with a formula that contains no variables; in this case, either the formula contains the empty clause and is not satisfiable, or the formula is itself empty and thus satisfiable.

We use this opportunity to remark a small improvement to the algorithm above: Whenever we observe a unit clause $\{\ell\}$, we can set the literal ℓ to true instead of branching on the underlying variable. This additional rule is called *unit propagation*. Now the following is an important class of SAT-algorithms: we use unit propagation whenever possible, and if it is not possible, we choose a variable according to some rule and branch on it. Any such algorithm is called a DPLL-algorithm (named after their inventors Davis, Putnam, Logemann, and Loveland). DPLL-algorithms have a degree of freedom: In each branching step, the algorithm gets to choose which variable to branch on next. Practical SAT-solvers are based on the DPLL-paradigm, and they turn out to be extremely efficient on industrial SAT-instances. On the other hand, from a theoretical perspective we cannot explain with current theoretical methods why DPLL-algorithms are so good.

In particular, we can only seem to bound the running time of the DPLL-algorithm by $2^n \cdot \mathrm{poly}(n)$ in the worst case. Surprisingly, perhaps, it is possible to prove a better bound for kSAT: In each branch step, the algorithm chooses the next branch variable uniformly at random from the set of unset variables. This randomized algorithm has an expected running time of at most $2^{\left(1-\frac{1}{k}\right)n} \cdot \mathrm{poly}(n)$ which was proved by Paturi, Pudlák, and Zane (1999). Note that as k tends to ∞ (and the problem kSAT comes closer to SAT), the growth rate of this running time tends to 2. The *strong exponential time hypothesis* (SETH) is that, for every $\epsilon > 0$, there is a k such that kSAT does not have a $(2-\epsilon)^n$-time algorithm. Clearly if SETH is true, then $\mathsf{P} \neq \mathsf{NP}$; the converse is not known and maybe SETH is false.

7.4 Self-reducibility

Search versus Decision

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

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

In practice, we want to *find concrete solutions* to our problems, and it is not usually sufficient merely to know that a solution exists. From the branching algorithm for SAT, we can generalize the concept of "branching" to other decision problems as follows.

Definition 7.4 *A language A is called* downward self-reducible*, if there is a deterministic polynomial-time oracle Turing machine M that satisfies $A = L(M^A)$ and, on input x, the machine M only queries oracle strings of length $< |x|$.*

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. Using the branching rule from the basic branching algorithm for SAT, we immediately observe the following.

Theorem 7.5 *k*SAT*,* SAT*, and* CSAT *are downward self-reducible.*

Exercise 7.1 *Show that, if A is downward self-reducible, then $A \in$ PSPACE.*

Exercise 7.2 *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: *Prove by induction over n that $A^{\leq n} = B^{\leq n}$ holds.*

For each problem $A \in$ NP, there is a relation R that is polynomial-time computable such that (7.1) 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 R an NP*-relation* or *polynomially bounded relation.* Given an 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 7.6 *Let R be the relation corresponding to* SAT*, i.e., $R(x, y) = 1$ if x encodes a CNF-formula F and y is a satisfying assignment of F.*

1. $L(R) = \texttt{SAT}$.

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 deterministic polynomial-time oracle Turing machine M. Then M is a search-to-decision reduction, that it, it reduces the problem of finding a certificate to the problem of deciding whether there exists a certificate.

Recall that $M^{L(R)}$ denotes that function that is computed by M with oracle $L(R)$. In case $L(R)$ is NP-complete, we will prove below that search(R) is self-reducible. In the example $L(R) = \texttt{SAT}$, we get that, if $\texttt{SAT} \in \mathsf{P}$ holds, then we can find satisfying assignments of satisfiable formulas in polynomial time.

Theorem 7.7 *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 at most polynomially harder than deciding membership. In practical applications, even just a polynomial overhead may be very disadvantageous; however, many algorithms automatically compute a certificate without additional overhead.

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

7.5 co-NP

Recall that co-NP is the class of all languages L for which $\bar{L} \in \mathsf{NP}$ holds. Thus L is in co-NP if there is a polynomial time bounded nondeterministic Turing machine M such that, for all $x \in L$, every path in the computation tree of $M(x)$ is accepting, and, for all $x \notin L$, there is at least one rejecting path in the computation tree of $M(x)$. This gives us a characterization in terms of certificates for co-NP: A language L is in co-NP if and only if, for all $x \in \Sigma^*$, we have

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

Here, R is the polynomial-time computable relation that determines whether the computation path y in the computation tree of $M(x)$ accepts. Hence, co-NP contains precisely those languages whose no-instances can be verified efficiently.

Let UNSAT be the encodings of all unsatisfiable CNF-formulas. Now UNSAT may not be equal to the complement of SAT because the complement of SAT contains all CNF-formulas from UNSAT as well as all strings that are not an encoding of a formula in CNF. However, this is merely a technicality since we can determine in polynomial time whether a given string corresponds to a CNF formula. Thus we always have $\overline{\texttt{SAT}} \leq_\mathrm{P} \texttt{UNSAT} \leq_\mathrm{P} \overline{\texttt{SAT}}$, that is, UNSAT and $\overline{\texttt{SAT}}$ are equivalent under polynomial-time many-one reductions.

A formula F is unsatisfiable if and only if $\neg F$ is a tautology. If F is in CNF, then we can compute the DNF of $\neg F$ in polynomial time by applying De Morgan's law. Hence we have the reduction $\texttt{UNSAT} \leq_\mathrm{P} \texttt{TAUT}$, where TAUT is the following *tautology problem*.

Definition 7.8 TAUT *is given a DNF formula F to decide whether F is a tautology, i.e., whether all assignments satisfy F.*

Note that, if we replaced DNF with CNF here, the resulting problem would be polynomial-time computable. Since SAT is NP-complete, $\overline{\texttt{SAT}}$ is co-NP-complete and so is TAUT.

7.6 NP and co-NP

Since P is closed under complementation, one approach to show that $\mathsf{P} \neq \mathsf{NP}$ would be to show that NP is not closed under complementation, that is, $\mathsf{NP} \neq$

co-NP. On the other hand, to show that NP is closed under complementation, it is sufficient to show that an NP-complete problem is contained in co-NP.

Exercise 7.3 *If* co-NP *contains an* NP*-complete problem, then* NP = co-NP*.*

Most researchers who have an opinion on the matter conjecture that NP $\neq$ co-NP. Furthermore, the relationship between P and NP $\cap$ co-NP is open. The following problem is in the intersection of NP and co-NP, but it is not known to be in P. Hence this problem is a candidate for a problem in (NP $\cap$ co-NP) $\setminus$ P.

Definition 7.9 `FACTOR` *is given two positive integers x and c in binary to decide whether x has a factor b with $2 \leq b \leq c$.*

Exercise 7.4 *Prove the following:*

1. `FACTOR` $\in$ NP*.*
2. `FACTOR` $\in$ co-NP*. (You can use the result by Agrawal, Kayal, and Saxena that* `PRIMES` $\in$ P*; that is, we can determine in polynomial time whether a given positive integer x is a prime number or not, where the integer x is encoded in binary)*

8 The polynomial time hierarchy

Alternating quantifiers

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

8.1 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, $\mathsf{co\text{-}NP}$ 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 8.1 *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 8.1 *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 8.2 *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 8.3 *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

$$B' = \{ \langle x, \langle y, b \rangle \rangle \mid \langle \langle x, y \rangle, b \rangle \in B \}. \tag{8.1}$$

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 (8.1). 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 8.2. 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 8.1 and Lemma 8.3, $\Sigma_{i+1}^\mathsf{P} = \Sigma_i^\mathsf{P}$. 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.

8.2 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 8.4 *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. $\mathtt{QBF}\Pi_k$ *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 $\mathtt{QBF}$ is PSPACE-complete. $\mathtt{QBF}\Sigma_k$ and $\mathtt{QBF}\Pi_k$ are complete problems for Σ_k^{P} and Π_k^{P}, respectively.

Exercise 8.2 *Show that* $\mathtt{QBF}\Sigma_k$ *is* Σ_k^{P}*-complete.*

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

Exercise 8.3 *If* PH *has complete problems, then* PH *collapses.*

8.3 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 8.5 *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.

9 P, NP, and PSPACE

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

Exercise 9.1 *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 9.1 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 if and only if 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 deterministic polynomial-space Turing machine with $L(M) = L$. Without loss of generality, 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, this encoding is an injective function. 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 if and only if $x \in L$. 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 if and only if 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 if and only if 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 if and only if $x \in L$. ∎

Exercise 9.2 *Construct a formula S such that $S(\xi, \eta)$ is true if and only if $\xi \vdash_M \eta$. 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.