Computational Complexity and Physics

David Gross
Institute for Theoretical Physics, University of Cologne

January 13, 2016

This is an evolving draft of the lecture notes for a course offered in the fall term 2015 at the University of Cologne. It will grow as the lecture proceeds. Only parts that have been covered in the lecture should be considered stable (in particular, I recommend against printing out these notes before the end of the term). Caveat emptor.

Contents

1 Synopsis

Complexity theory is a branch of theoretical computer science. It provides quantitative statements about how hard it is to solve certain problems on a computer. Examples of such problems include:

- TRAVELING SALESMAN – Find the shortest route through a given list of cities
- INTEGER FACTORIZATION – break the secure "https" internet communication protocol
- GROUND STATE – Find the ground state energy of a physical spin system

In principle, the laws of physics enable a computer to make arbitrary predictions about the behavior of physical systems. In practice, however, this often involves solving problems for which no efficient algorithms are known to exist. Complexity theory helps us to decide when the lack of efficient algorithms reflects an insurmountable, intrinsic difficulty of the problem, rather than our limited understanding.

Conversely, physics also contributes to complexity theory. The reason is that computers are physical systems themselves and what they can and cannot do is therefore ultimately a physical question. The task here is to decide whether the mathematical models employed by computer scientists faithfully capture all computational processes allowed for by Nature (the young theory of quantum computing indicates that this may not be the case).

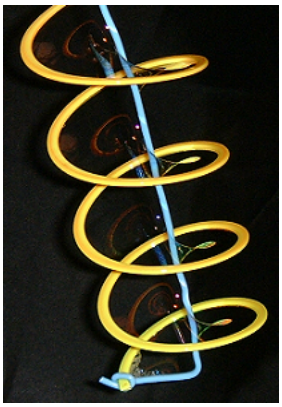
Covered topics fall into two categories:

Complexity theory for physics

- Computable and uncomputable functions: Halting Problem, Kolmogorov
- Complexity, Gdel's Incompleteness Theorem
- Complexity classes: P, NP, NP completeness, BPP, P/poly
- Important decision problems: Satisfiability, cuts in graphs
- Hard problems in physics: ground state energies, partition functions, protein folding

Physics for complexity theory

- Church-Turing thesis, billiard ball computers, DNA-computers
- Quantum computing: teleportation, Deutsch-Jozsa algorithm, Shor's algorithm, BQP, QMA
- Existence of "true randomness" from Bell's argument
- Reversibility, entropy, Landauer principle, Maxwell's demon



Picture credit: Blinking Spirit via Wikimedia, CC BY-SA 3.0.

Figure 1: The use of soap films is sometimes rumored to give a competitive computational method for finding *minimal surfaces* and related combinatorial problems. While certainly a compelling story, it doesn't seem to hold up to detailed scrutiny [1].
See also: Story on Munich Olympic Stadium roof.

Part I
Complexity Theory for Physics

2 Models of Computation

Until the Second World War, a *computer* was a person manually performing calculations. Today's computers are almost exclusively based on integrated semi-conductor circuits. But more exotic physical processes have been discussed as a means for solving computational problems. These include

- Soap films (Fig. 1) for finding *minimal surfaces*
- "DNA computers", where the problem to be solved (no pun intended) is encoded in DNA strands and the elementary steps of the computation are biochemical reactions. While these elementary steps evolve slowly, this might be counteracted by using a very large number of molecules to achieve a high degree of parallelism.
- Quantum computers whose state space and gates are based on the laws of quantum mechanics (later).
- Closed timelike curves as a hypothetical resource for computation have recently enjoyed a few years of attention.
- Billiard balls rolling on a frictionless surface and scattering elastically off each other (because an ideal such process requires no energy to run)

In order to make sense of this chaos, we'll need to define clean mathematical models of what a "computer" is. Maybe surprisingly, it has turned out that once one abstracts over the physical details, almost all the proposed physical implementations seem

to be equivalent. Their power is described essential by the mathematical model of a *Turing machine*. The sole exception are quantum computers that employ quantum mechanical primitives. There is now strong – if not overwhelming – evidence that these outperform any classical computing device.

Formalization is also necessary from a purely theoretical point of view. In order to quantify the intrinsic difficulty of computational problems, we need a clean theoretical framework. It will allow us to focus on essential questions rather than implementation details, and argue with mathematical rigor.

As a motivating example, we state an apparent paradox of computability theory. Without precise notions, it seems completely puzzling – but a clean formulation will allow us to resolve the contradiction fairly easily. (As a bonus, with a little bit of work, we can use the insights gained to explain a pop science classic – Gödel's Incompleteness Theorem).

Kolmogorov Complexity (informal version).
Let x be the smallest natural number which cannot be described using fewer than 20 words.

Feel that something's fishy?

The problem is of course that we defined x using 15 words. That's less than the 20 it ought to have by definition. So no number can consistently be assigned to x. On the other hand, the definition looks innocuous. There certainly *is* a set of numbers that can requires at least 20 words to be defined and among them, there will be a smallest one. What in the world did go wrong?

It will turn out that one can resolve the apparent paradox by carefully defining the word "to describe" in terms of a computer program generating the number.

2.1 Finite State Machines

Finite state machines (or *deterministic finite state automata, DFA*) are the simplest model of computers we'll encounter.

We start with an example: The PARITY-machine. It's defined as follows:

EVEN PARITY

INPUT:	bit string x
OUTPUT:	1 if number of 1's in x is even, 0 else.

Our strategy for solving PARITY is to read the bits of x one by one and remember whether the number of 1s seen so far was even or odd. Whenever we encounter an additional 1, we toggle the internal state. The strategy can be represented conveniently in a *state transition diagram*

With the example given, the general definition is straight-forward.

Definition 1. *A* deterministic finite state automaton *is specified by the following pieces of data:*

- *A finite set of* states Q.

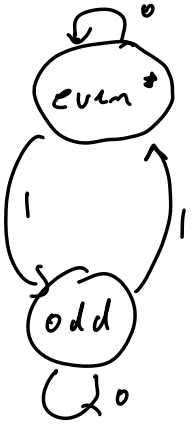


Figure 2: A *state transition diagram* of a deterministic finite automaton. Every state $q \in Q$ is represented as a node. The arrows represent the state change induced by an input $x \in \Sigma$. More precisely, there is an arrow between two states $q \to q'$ with label $x \in \Sigma$ if the state transition function δ fulfills $\delta(q, x) = q'$. The initial state is marked by a star.

- *A finite input* alphabet Σ.
- *A* transition function $\delta : Q \times \Sigma \to Q$.
- *An initial state* $q_0 \in Q$.
- *Optionally, a designated subset* $F \subset Q$ *of* accepting states.

The interpretation is that we start in state q_0. Then, we sequentially receive new input from the alphabet Σ. When encountering the input $x \in \Sigma$ while in state q, we transition to state $\delta(q, x)$. The data can equivalently be represented diagrammatically, as in Fig. 2.

How powerful are DFAs? It turns out that there are natural problems a DFA cannot solve. In order to exhibit one, we need to introduce a bunch of new terms.

A *bit string* is finite-length string of 0s and 1s, i.e. an element of

$$\{0, 1\}^* = \emptyset \cup \{0, 1\} \cup \{00, 01, 10, 11\} \cup \{000, \dots\} \cup \dots.$$

A *language* L is just a subset of all bit strings $L \subset \{0, 1\}^*$ (yes, it is that simple and no, that's not in accordance to the non-technical use of the word "language"). Examples of languages include "the bit strings with even parity", "the prime numbers (encoded in binary)", "all sequences of amino acids that fold into proteins that target cancer cells". *Recognizing* an element of one of these languages is, respectively, trivial, a difficult but well-understood task, and a completely open problem.

With every language L, one associates a *decision problem* which is maps a bit string to 1 if it is an element of L and to 0 if it is not.

Definition 2. *A language L is* DFA-decidable *(also:* DFA-computable*) if there is a DFA which will transition to an accepting state F if and only if the input x is in L.*

We have seen that EVEN PARITY is DFA-decidable. Not all languages are, though. Next, we'll describe the language PARENTHESES which is *not* decidable by a machine with finite memory (and hence not by any DFA). This will serve as a motivation for introducing a more powerful model of computation – *Turing machines*.

One should emphasize that while the models and problems introduced so far seem almost banal, it is quite remarkable that we can rigorously prove non-computability results after so little preparation.

PARENTHESES

INPUT:	a string composed of opening "(" and closing ")" parentheses
OUTPUT:	1 if they match up in the usual sense, 0 else.

Thus, elements of the language include $(()())$ or $((())())$. A non-element would e.g. be $)))$. (Strings of parentheses are not strictly bit strings, of course. But it should be clear that they can trivially be encoded as such and we will generally be cavalier about such obvious translations).

Theorem 3. *The language* PARENTHESES *is not DFA-decidable.*

We will provide the rather obvious proof in painful detail. Not because the result is surprising, but in order to show how formalization – for all the abstraction it forces as to go through – at least enables us to arrive at rigorous statements about computational problems.

Proof. Let

$$\begin{aligned} a_n &= \underbrace{((\ldots(}_{n\times}, \\ b_n &= \underbrace{))\ldots)}_{n\times}. \end{aligned}$$

Fix a DFA $\langle Q, \Sigma, \delta, q_0, F\rangle$ and let q_n be the state after the input of a_n. By the pigeonhole principle, in the sequence

$$q_1, q_2, \ldots, q_{|Q|+1},$$

there is at least one pair $x \neq y$ such that $q_x = q_y$. But that means the automaton can accept $a_x b_x$ (which is in the language) if and only if it accepts $a_y b_x$ (which is not). In any case, the automaton does not decide PARENTHESES. □

2.2 Turing Machines

Previously, we have seen that the finite number of states of a DFA limits its computational power. In a way, that's a faithful reflection of reality, where every computer's memory is finite. The state space of my laptop's memory, e.g., is roughly $|Q| = 2^{8\times 10^9} \simeq 10^{2.4\times 10^9}$. While enourmosly large, it *is* finite and there certainly are calculations where this is a limiting factor.

On the other hand, it would theoretically be much cleaner to have one generaly theory of computable problems that does not depend on a memory-size parameter. This

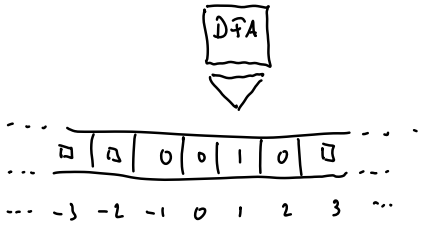


Figure 3: A *Turing machine* consists of a DFA (the *head*) with the ability to read input from and write output to an infinite tape.

is maybe analogous to the way space and time are treated in physical theories. There is no reason to believe that space is continuous rather than quantized into multiples of some finite "Planck length". One can perfectly well describe a discretized version of Newtonian mechanics where the coordinates come in finite multiples of some unit length (and, in fact, that's necessarily so if a dynamical system is simulated on a digital computer). However, having a "platonic ideal" of a theory where one doesn't have to worry about the onset of a scale where the discretization becomes noticable and the physics changes is certainly benefitial. The same is true for computability theory and as a result, our main model of computationl will be such a "platonic ideal" of a machine with infinite memory. In this respect, it is strictly stronger than any existing physical computer. Thus, problems undecidable even on a Turing machine are much less decidable in the real world.

A Turing machine is a DFA that has access to an infinite tape containing memory cells (Fig. 3). The automaton is called the "head" and thought of as being positioned above one of the memory cells at any given point of time. An elementary step of the computation consists in reading the content of the memory cell underneath the head as input, processing it by transitioning to a new state, writing a new value to the cell and possibly moving one cell to the left or right.

The ingredients more formally:

Definition 4. *A* Turing machine *is defined by the following pieces of data:*

1. *A finite set of* states Q. *The set contains an* initial state $q_0 \in Q$ *and a* halting state q_H.
2. *A finite* alphabet Σ. *It contains a designated symbol called* "blank" *and symbolically denoted by* $\square$.
3. *A* transition function

$$\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, S, R\}.$$

The cells of the tape are labeled by integers $p \in \mathbb{Z}$. Initially, the head is positioned over cell 0 and its DFA is in state q_0. We assume that all but finitely many cells of the tape are blank, i.e. contain the symbol $\square$. The non-blank cells contain any symbol for Σ – this is considered the machine's input.

At a given step of the computation, assume the head is positioned over cell p which contains the symbol s and the DFA is in state q. The action of the machine is determined by the transition function $\langle q', s', m\rangle = \delta(q, s)$ in the following way: The DFA

transitions to state q', it overwrites the current cell with symbol s' and it will move one step to the left $p \to p' = p - 1$ if $m = L$, to the right if $m = L$, or will stay at the current cell if $m = S$ respectively. The process then repeats. If the DFA ever transitions to the halting state q_H, the computation is considered done and no more action will be taken. The final state of the tape is considered the machine's output.

Why would anybody consider such an awkward mechanism?

The stated initial motivation by Alan Turing was to model human computers. These people would get a definite set or rules "δ" of the day "δ" and would mechanically apply them, having access to an unlimited supply of paper (Turing's genius is undisputed – but if he was a respectful boss, then this is not reflected in his mathematical formalization of his employees...).

More important, though, is an *a posteriori* motivation: It seems that the model of the Turing machine is borad enough to capture *any* process that one would naturally consider a "computation". This is expressed in the

(Weak) Church-Turing Thesis.
Any physical process that could reasonably be considered a "computation" can be modeled by a Turing machine.

The way its stated, it is not a mathematically precise conjecture. While we have precisely defined what a Turing machine is, we have not defined what we mean by "computation". Still, as an informal principle, it has stood the test of time.

As *physicists*, we can actually take a shot a *proving* the CTC. Proofs would work like this: We start out with a model of physical reality in terms of, say, Newtonian mechanics, non-relativistic quantum mechanics, or relativistic quantum field theory and define a "computation" to be any dynamical process compatible with the differential equations governing such theories. One can then try to prove that numerical integration techniques that can be implemented on a Turing machine are sufficiently powerful to simulate the time evolutions of these theories. We'll pursue proofs of the CTC in the second part of the lecture.

We can now define computability.

Consider a Turing machine T whose initial tape is empty, except for a finite string x of symbols written from position 0 on. If T

Definition 5 (Computable functions, decidable languages)**.**

1. *Let T be a Turing machine. Suppose we run T on an initial tape that is blank except for a finite string x of symbols starting at position 0. If T reaches the halting state q_H after finitely many steps, we say that T* halts on input x*. In that case, let y be the content of the the tape after T has halted (with leading and trailing blanks removed). We refer to y as the* output *and write $T(x) = y$.*

2. *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function taking bit strings to bit strings. If there exists a Turing machine T such that $T(x) = f(x)$ for all $x \in \{0,1\}^*$, then f is* (Turing) computable*.*

3. *Let $\mathcal{L}$ be a language. It is* (Turing) decidable *if there exists a Turing machine T such thtat $T(x) = 1$ if $x \in \mathcal{L}$ and $T(x) = 0$ otherwise.*

Turing machines are more powerful than DFAs.

Theorem 6. *The language* PARANTHESES *is Turing-decidable.*

Proof. By construction. In words, our strategy is this: We start "scanning right" (SR) for a closing paranthesis. If we find one, we overwrite it with a symbol ("-", say), that we take to indicate that the paranthesis has been processed. Having overwriten the closing one, we "scan left" (SL) until we find the matching opening paranthesis and likewise overwrite it. We skip any already-processed cells "-" that we encounter in the process. Having thus processed one pair of parantheses, we re-enter "scan right" state to repeat the process. Should we run into a blank symbol while scanning left, we declare the input invalid (☹), as it contains an unmatched ")". Once we hit the blank while scanning right, we switch into a check mode (CH) that scans left to see whether all parantheses have been eliminated. If one is left, we also declare the string invalid – otherwise, it is valid (☺).

Alphabet: $\Sigma = \{\square, (,), X\}$.

States: $Q = \{q_0 = SR, SL, CH, ☺, ☹\}$.

Transition function:

q	s	q'	s'	m
SR	$\square$	SR	$\square$	R
SR	(	SR	(	R
SR	)	SL	–	L
SL	–	SL	–	L
SL	(	SR	–	R
SL	$\square$	☹	$\square$	S
SR	$\square$	CH	$\square$	L
CH	–	CH	–	L
CH	(	☹	(	S
CH	$\square$	☺	$\square$	S

□

Strictly speaking, we'd have to add further rules that clear the tape and output 0 once we transition to ☹ and 1 if in ☺. However, the example above should convince you that (a) this would clearly be doable and (b) it would be a pain to actually write out. Turing machine transition functions are most certainly not the most digestable way of presenting algorithms. From now on, we will generally use a "pseudo code" notation for algorithms and I trust that the reader could transform these into a TM (and be loath to do so) if they wanted.

2.3 Universal Turing Machines

We come to one of the deepest realization in the theory of computing: There are *universal* Turing machines (UTM) that are as powerful as *any* other such machine. Indeed, until now, we introduced a new machine for every specific task. A universal Turing machine, on the ohter hand, can be *programmed*. It is a *general-purpose* computer.

To see how this works, we start with an arbitrary Turing machine T and show that it can be represented by a bit string. For modern-day students, this is not too surprising. After all, the PARANTHESES Turing machine was specified in my lecture notes, which are available as a pdf document, which deep down is nothing but a string of bits. Let us anyway exhibit a concrete method.

First, we need a method of representing natural numbers as bit strings. A first step is to just use their binary reprsentation, i.e. the fact that for every $n \in \mathbb{N}$, there exists a unique bit string x of length $k = \lceil \log_2 n \rceil$ such that

$$n = \sum_{i=0}^{k-1} x_i 2^i.$$

One problem with this representation is that it can't naively be concatenated. I.e. if x, y represent two numbers, then the compound string xy doesn't hold any information about when the first string ends and when the second one starts. One (somewhat wasteful) way around this is to preced the number with a string specifying its length. For example, with n, k as above, we can encode

$$n \mapsto \mathrm{enc}(n) := \underbrace{1 \dots 1}_{k\times} 0 x_1, \dots, x_k.$$

Then one can clearly recover $n_1, n_2 \in \mathbb{N}$ from the concatenation $\mathrm{enc}(n_1)\,\mathrm{enc}(n_2)$.

Now consider the data $Q, q_0, q_H, \Sigma, \delta$ that specifies a Turing machine T. Clearly, the way a particular state $q \in Q$ or symbol $s \in \Sigma$ are represented is immaterial for the functioning of the machine. So we can without loss of generality assume that the states are just binary numbers from 0 to $|Q| - 1$ and likewise for the alphabet. What is more, we may also always assume that $q_0 = 0$ and $q_H = 1$. Thus, the machine is specified by the data $|Q|, |\Sigma|, \delta$. The firs two, we can just encode and concatenate $\mathrm{enc}(|Q|)\,\mathrm{enc}(|\Sigma|)$. Thus, we are left to define a way of turning δ into a bit string. But this is also simple, using the table notation for δ employed in the proof of Theorem 6. First we encode the number of rows and then, one by one, for every row encode the numbers q, s, q', s', m (where, for m, we may use the convention $L = 0, S = 1, R = 2$). Concatenating all these encodings toghether, we arrive a one long bitstring that uniquely represents the Turing machine T. What is more, if $x \in \Sigma^*$ is a finite input to T, it can likewise be specified by first encoding $|x|$ and then each symbol x_i. Thus, we arrive a the following invertible map from Turing machines and inputs to bit strings:

$$\mathrm{enc}\langle T, x\rangle := \mathrm{enc}\langle |Q|, |\Sigma|, |\delta|, q_1, s_1, q_1', s_1', m_1, \dots, |x|, x_1, x_2, \dots \rangle.$$

The resulting bit string can of course also be read as the binary representation of a natural number. This is called the *Turing number* $\mathrm{TN}(T)$ associated with the machine T.

Definition 7. *A* universal Turing machine *is a Turing machine U such that $U(\mathrm{enc}\langle T, x\rangle) = \mathrm{enc}(T(x))$ for every Turing machine T and input x on which T halts.*

Thus a UTM can "emulate" any other Turing machine. It is unclear that a UTM exists (indeed, there exists *no* universal DFA! (why?)). However, one can show that this is indeed the case:

Theorem 8 (Turing '36 (!))**.** *There exists a univeral Turing machine.*

TBD: Discuss the impact.

2.4 Kolmogorov Complexity Revisited

Recall the puzzling paradox posed earlier.

Kolmogorov Complexity (informal version).
Let x be the smallest natural number which cannot be described using fewer than 20 words.

We can now precisely define what we mean by "to describe".

Definition 9. *Fix a UTM U. The* Kolmogorov complexity $K(x)$ *of a bit string $x \in \{0,1\}^*$ is the length of the smallest input p to U such that $U(p) = x$.*

The complexity $K(x)$ is always smaller than some linear function of x. That's because one can easily design a Turing machine that executes the program "PRINT 'x", i.e. a machine that has x "hard-coded" into its transition function δ. For some "compressible" x, however, much shorter programs are possible. TBD: expand on that.

Thus, the paradox could be formulated more precisely using the following piece of pseudo-code:

Algorithm 1: MIN: A proposed precise version of the Kolmogorov paradox.

Input:
$n \in \mathbb{N}$ minimal complexity

Output:
$x \in \mathbb{N}$ smallest number with $K(x) \geq n$

MIN(n):
for $k = 0$ *to* ∞ **do**
  if $K(x) \geq n$ **then**
    return x
  end
end

Theorem 10. *The Kolmogorov complexity K is not computable.*

Proof. Assume (for the sake of reaching a contradiction) that K is computable. Then the pseudo-code in Algorithm 1 defines a Turing machine M which computes MIN.

Let $n = |\operatorname{enc}(M)|$ and let

$$n_0 = n + 2\lceil \log_2 n \rceil + 4, \qquad x = \textsc{Min}(n_0).$$

Then we have on the one hand

$$K(x) \quad < \quad n_0,$$

because with $p = \operatorname{enc}(M, n_0)$ we have that $U(p) = x$ and $|p| < n_0$ (why?). On the other hand, by definition of MIN,

$$K(x) \quad \geq \quad n_0$$

This is a contradiction and therefore the assumption (that K be computable) must be wrong. □

TBD: discuss.

We can identify further uncomputable functions by trying to propose algorithms for K and understanding why they fail.

Algorithm 2: A first proposed algorithm for the Kolmogorov complexity.

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

Output:
$K(x) \in \mathbb{N}$ length of shortest program computing it

K(x):
for $p = 0$ *to* ∞ **do**
  if $U(p) = x$ **then**
    return $|p|$
  end
end

Why does that not work? ... Answer: there might be programs such that $U(p)$ never halts. Thus the algorithm might "hang" indefinitely at the first such program before it ever has a chance of computing x. So, why don't we work around that problem by excluding the infinite loops?

Algorithm 3: A second proposed algorithm for the Kolmogorov complexity.

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

Output:
$K(x) \in \mathbb{N}$ length of shortest program computing it

K(x):
for $p = 0$ *to* ∞ **do**
  if WILL-HALT*(p)* **then**
    if $U(p) = x$ **then**
      return $|p|$
    end
  end
end

Wow! That's an algorithm for $K(x)$. Unfortunately, we already know that no such algorithm can exist. Hence the subroutine checking whether or not a program will halt cannot exist.

Corollary 11. *The* Halting Problem *is undecidable. I.e. there does not exist a Turing machine that can decide the language of inputs x to a UTM U such that U will halt on x.*

3 Gödel's Incompleteness Theorem

With fairly little effort, we have been able to show the truely remarkable fact that there are functions that are mathematically well-defined, yet cannot be computed! However, our examples so far all seemed somewhat self-referential: Turing machines can't decide how Turing machines behave. The theory only seems to solve problems it has introduced itself.

However, with a bit of work one can identify functions that appear in other contexts than computer science (and, indeed, partly precede the development of c.s.) that also admit no computational solution. Over the past few years, there have been several papers pointing out that problems in many-body quantum mechanics are undecidable (the present lecturer was recently involved in one such publication). We won't have time to explain any of these in class (but ask me if you're interested).

Instead, we will sketch the proof of the best-known statement of the foundation of mathematics: *Gödel's Incompleteness Theorem*.

Gödel's Incompleteness Theorem (informal version).
There are algebraic statements about natural numbers that are true but not provable.

As was true for our first encounter with Kolmogorov complexity, there are several terms in that statement that have an intuitive, but not-yet precisely defined meaning. Like "provable" and "algebraic statements about natural numbers". With only the few notions of computer science that we have introduced so far, we can already give a more modern, stronger, and precise version – Theorem 13. (We'll comment on the original version of Gödel in terms of *sound* and *complete* axiomatic proof systems below).

Definition 12. *An* arithmetic formula *is any string that can be composed of the following symbols:*

- *the number* 1*,*
- *addition, multiplication, and power*[1]*:* $+, \times, **$*,*
- *numerical comparisions:* $=, <, >$*,*
- *logical AND, OR, and NOT:* $\wedge, \vee, \neg$*,*
- *parantheses:* $(,)$*,*
- *existence and for-all quantifiers over natural numbers:* $\exists_k$*,* $\forall_k$*,*
- *variables* $k, l, x, y, \ldots$*, as long as they are bound to a quantifier.*

Let $\mathcal{T}$ *be the language of arithmetic formulas that are well-formed and true as statements about natural numbers.*

Here is an example of an element of $\mathcal{T}$:

$$\forall_k \exists_l \big((k = (1+1) \times l) \vee (k = (1+1) \times l + 1)\big).$$

[1] Including the "power" symbol is not strictly necessary, but will make our constructions easier.

It says that every natural number k is either even (i.e. $k = 2l$) or odd (i.e. $k = 2l + 1$). It also shows that our language is quite cumbersome. E.g., in order to keep the set of allowed symbolds finite, we included just one number – 1. But since

$$\underbrace{(1 + 1 + \cdots + 1)}_{n\times} \in L,$$

we can effectively express statements involving any natural number n. Thus, in the following, we will use standard mathematical symbols withtout further comment, as long as it is clear how to express them in terms of those listed in the definition of arithmetic formula.

There are many formulas whose truth value is not known to us, even though mathematicians have tried hard for centruries to establish them. For example *Goldbach's Conjecture*:

$$\forall_k \, (k = 1) \vee (\exists_p \exists_q (2 \times k = p + q) \wedge PRIME(p) \wedge PRIME(q)) \,, \qquad (1)$$

where we have used the abbreviation

$$PRIME(n) := \neg\big(\exists_k \exists_l (k > 1) \wedge (l > 1) \wedge (n = k \times l)\big).$$

At the beginning of the 20th century, significant efforts were expended by mathematicians to find an algorithm that could systematically decide whether statements like (1) are in fact true. Well, no such luck:

Theorem 13. *The language $\mathcal{T}$ is not decidable.*

The proof strategy is a first instance of the core concept of *reduction*. We will show that the Halting problem *reduces* to the proble of deciding L: Given the Turing number $\mathrm{enc}(T)$ of a Turing machine T, one can algorithmically construct an arithmetic formula ϕ_T that is true if and only if T halts. Thus, if one could decide the truth of such formulas, one could also solve the Halting problem. But we already know that the Halting problem is undecidable and thus, we are done.

Typical for such arguments, the proof has a "constructive" feel. We have to "write the Halting problem into an arithmetic formula". In yet other words, we have to show that the "expressive power" of arithmetic is as strong as the one of arbitrary Turing machines. (For reasons explained later, reductions are *the* central tool for time complexity analysis. So listen up!).

The construction works as follows: Start by noticing that we can represent the current state of the Turing machine as a number (or, equivalently, as a bit string). Here, by "state", we mean the state of the DFA and all non-blank symbols on the tape. We will come up with a precise construction in a minute. Now comes the central insight:

> Fix a Turing machine T. Then there exists an arithmetic formula NEXT_T in two variables such that $\mathrm{NEXT}_T(n_1, n_2)$ is true if and only if the state encoded by n_1 will be followed by the state n_2 after one application of the transition function of T.

This insight will help us transform a construction that has a "dynamical" feel – like the evolution of a TM – to a more "static" object – in this case an artihmetic formula.

To complete the construction, we'll require a slew of further formulas (which all turn out to be relatively simple, when compared to NEXT). The outline is this:

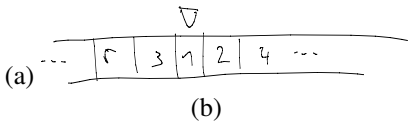


Figure 4: (a) Method for turning the content of a tape into a bit string. Encode cell by cell, starting from the cell underneath the head and moving outwards, alternatingly to the right and to the left. The numbers in the cell indicate the position of the cell in the resulting bitstring.
(b) TBD: Re-labeling after head moved to the left.

- $\mathrm{INIT}(n)$ is true if and only if n represents T with empty tape and in the starting state q_0.
- $\mathrm{HALT}(n)$ is true if and only if n represents a halting state of T.
- $\phi_T(t)$ is true if and only if T halts after t steps. It will (roughly) be patched together from the previous formulas as in

  $$\text{“}\phi_T(t) = \exists_n \Big(\mathrm{INIT}(n_1) \wedge \forall_k \big((k{+}1 > t) \vee \mathrm{NEXT}(n_k, n_{k+1}, k) \big) \wedge \mathrm{HALT}(n_t) \Big)\text{”}.$$

- Then the statement “T halts” is equivalent to $\exists_t \phi_T(t)$ and we're done.

Everyone must understand the idea so far. The following (gory) details are, in comparision, somewhat less important.

First, we have to fix the encoding. Choose a number r that is big enough that all states $q \in Q$ and all symbols $s \in \Sigma$ can be encoded using r bits. The first r bits of n will just be the current state q of the DFA. We use a “big-endian” convention, i.e. the leas significant bits come first (that's the opposite way from how we normally write out digits of numbers, but it allows us to read everything from left to right in a more consistent way). Encoding the tape is a bit more problematic for two reasons: (i) There is no bound on the number of non-blank symbols and (ii) It extends to infinity in *two* directions, whereas we can only append further bits to n in one direction. We solve these problems as follows. The number n_t that will encoded the tth time step will include $2t$ cells. (In every step, the number of non-blank symbols can increase only by one – but that one symbol can be added either to the left or to the right of the current non-blank symbols, hence the factor of two.) There are two ways to tackle the second problem. One can prove that a TM whose tape extends only to one direction is still universal. However, we don't want to show that and enforcing that the tape never moves “into the negative” comes with its own overhead. Instead, we opt to enumerate the cells starting from the current position of the head in a “zig-zag fashion” depicted in Fig. 4. Thus, we map the state of T after t steps to the $(2t+1)r$-bit number n with bits $q, s_1, \ldots, s_{2t}$, where s_i is the symbol of the ith cell relative to the head position in zig-zag ordering.

We now define a few “helper formulas” (in any language other than algebera, these would be “subroutines”; in the context of such reductions, they are sometimes known as “gadgets”).

First and most importantly, the number n_t arose from concatenating $2t{+}1$ numbers.

We need a way of getting them out again. We'd need something like this:

$$\mathrm{EXTRACT}(x,n,i,d) = \begin{cases} \text{true} & \text{if } x \text{ is the } d\text{-bit number starting from bit } i \text{ of } n \\ \text{false} & \text{else.} \end{cases}$$

Can one design such a formula? Yes:

$$\mathrm{EXTRACT}(x,n,i,d) = \exists_{k,l}(k < 2^i) \wedge (x < 2^d) \wedge (n = k + x * 2^i + 2^{i+d}l).$$

(Some people include 0 in the natural numbers, while some don't. The above formula is more natural with 0 included, as a sub-string of a non-zero number might be 0 and we'll thus adopt this convention. It's not essential, though. If you prefer $0 \notin \mathbb{N}$, you have to make sure never to use the above formula on a number n where either the number to be extracted, nor the parts to the left or to the right are zero. That's doable, e.g. by padding with extra 1s before and after the actual "payload" of the bitstring...).

Now we can construct the first formulas from the list above.

$$\begin{aligned} \mathrm{INIT}(n) = \quad & \big(\forall_x\big(\neg\,\mathrm{EXTRACT}(x,n,1,r) \vee (x = q_0)\big)\big) \\ \wedge \quad & \big(\forall_x\big(\neg\,\mathrm{EXTRACT}(x,n,r+1,r) \vee (x = \square)\big)\big) \\ \wedge \quad & \big(\forall_x\big(\neg\,\mathrm{EXTRACT}(x,n,2*r+1,r) \vee (x = \square)\big)\big)\,, \end{aligned}$$

$$\mathrm{HALT}(n) = \quad \forall_x\big(\neg\,\mathrm{EXTRACT}(x,n,1,r) \vee (x = q_h)\big).$$

Now comes the monster formula: NEXT. A state n_1 turns into n_1 iff there is one row of the transition table that matches n_1 to produce n_2. Thus, NEXT will be a disjunction (logical "or") over one formula for every line of the transition table. Let's take the i row

q_i	s_i	q_i'	s_i'	m_i

and define the formla $\mathrm{NEXT}_i(n_1,n_2,t)$ that tests whether $n_1 \to n_2$ under that rule. Note that we snug in an argument of t. The reason is that our encoding of the state of the turing machine depends on the number of time steps t that have passed between the start of the calculation and us reaching n_1. This information must be supplied.

To make our lives slightly less miserable, we define

$$\begin{aligned} \mathrm{STATE}(x,n) &= \mathrm{EXTRACT}(x,n,1,r), \\ \mathrm{SYMBOL}(x,n) &= \mathrm{EXTRACT}(x,n,r+1,r). \end{aligned}$$

Then:

$$\begin{aligned} \mathrm{NEXT}_i(n_1,n_2,t) = \quad & \big(\forall_x\big(\neg\,\mathrm{STATE}(x,n_1) \vee (x = q_i)\big)\big) \\ \wedge \quad & \big(\forall_x\big(\neg\,\mathrm{SYMBOL}(x,n_1) \vee (x = s_1)\big)\big) \\ \wedge \quad & \big(\forall_x\big(\neg\,\mathrm{STATE}(x,n_2) \vee (x = q_i')\big)\big) \\ \wedge \quad & \ldots. \end{aligned}$$

The first two lines test whether the ith rule match n_1, while the next two lines test whether the state change has been implemented correctly. Now we need test whether

the symbol on the tape has been changed correctly. But the head might have moved and what used to be the symbol under the head no longer is. Where the previous position ended up depends on the direction the head moved into. The next line of NEXT_i will thus depend on m_i. If e.g. $m_i = L$, we must look at what is now cell number 2 (c.f. Fig. 4). For lines i with $m_i = R$, we thus add

$$\begin{aligned}\mathrm{NEXT}_i(n_1, n_2, t) = \quad & \ldots \\ \wedge \quad & \big(\forall_x \big(\neg \mathrm{EXTRACT}(x, n_2, 2 * r + 1, r) \vee (x = s_i')\big)\big) \\ \wedge \quad & \ldots,\end{aligned}$$

with the cases $m_i = R, S$ treated analogously. The next bunch of conjunctions have to ensure that the cells are re-ordered correctly and that the two new cells that have moved into our view are initialized as blank. For the blank test (this one, e.g., requires knowing t):

$$\begin{aligned}\mathrm{NEXT}_i(n_1, n_2, t) = \quad & \ldots \\ \wedge \quad & \big(\forall_x \big(\neg \mathrm{EXTRACT}(x, n_2, r * (2 * t + 1) + 1, r) \vee (x = \square)\big)\big) \\ \wedge \quad & \big(\forall_x \big(\neg \mathrm{EXTRACT}(x, n_2, r * (2 * t + 2) + 1, r) \vee (x = \square)\big)\big) \\ & \ldots,\end{aligned}$$

The re-ordering test for the case $m_i = L$ can (and should! ☹) be checked to read

$$\begin{aligned}\forall_x \Big(\\ & (x = 1) \vee (x > 2 * t) \\ \vee \quad & \big(\mathrm{EVEN}(x) \wedge \\ & \forall_y (\neg \mathrm{EXTRACT}(y, n_1, r * (2 * x + 1) + 1, r) \vee \mathrm{EXTRACT}(y, n_2, r * (2 * x + 3) + 1, r)) \\ & \big) \\ \vee \quad & \big(\mathrm{ODD}(x) \wedge \\ & \forall_y (\neg \mathrm{EXTRACT}(y, n_1, r * (2 * x + 3) + 1, r) \vee \mathrm{EXTRACT}(y, n_2, r * (2 * x + 1) + 1, r)) \\ & \big) \\ \Big)\end{aligned}$$

(TBD: re-do alignment).

From here, it should be trivial to piece together ϕ_T along the hints given.

We close the discussion by returning to the original statement of Gödel (his First Theorem, more precisely). An *axiomatic system* is a list of axioms and rules for logically deriving further statements from these axioms. The system is said to haven an *effective procedure* if there is some algorithm that can enumerate all statements that are provable by that system. An axiomatic system is *consistent* (or *sound*) if one cannot both prove a statement and its negation. It is *complete* if all true statements (for the *model* the system applies to – in our case the arithmetic of natural numbers) are provable. Gödel's First Theorem says that no axiomatic system for arithmetic of the natural numbers can be sound, complete, and possess an effective procedure.

For if such a system existed, it would give rise to an algorithm deciding $\mathcal{T}$: For any formula ϕ, just enumerate all true statements (possible by completeness and effectiveness) and check one-by-one whether that statement equals either ϕ or $\neg\phi$.

4 Time Complexity Classes

Computability Theory proved rather easy to understand. Within three lectures, we could show some of the deepest results about the limits of computation and mathematical reasoning. However, in practical terms, we're mostly interested in questions that do have one obvious, but impossibly difficult algorithm. Optimization problems fall into this class: Find the minimum value of an easy-to-compute function on a finite but huge domain. One *could* try all inputs to find smallest one, but often, this results in expected runtimes that exceed the age of the universe. Thus, it would be interesting to decide whether a failure to come up with a faster algorithm is indicative of our lack of imagination, or whether the problem is intrinsically hard. Answering these questions is the focus of the field of *computational complexity theory*.

Unfortunately, these questions turned out to be *much* more difficult than the theory of computation (which was basically settled before the first computers were even built). Strictly speaking, almost no lower bounds on the runtime of algorithms solving concrete problems are known. This is a devastating conclusion to draw after decades of computer science... Fortunately, computer scientists have come up with a second best solution: While we can almost never show a specific problem to be hard, it is often possible to show that the problem is *as hard* as an entire class of other, well-studied problems. Whence the large number of *conditional* or *relative* statements that come out of computer science, along the lines of "The problem is hard (unless "P=NP" or "the polynomial hierarchy collapses" – i.e. until a large class of problems is much easier than expected). Before introducing the theoretical notions, we start by looking at a physical example: The ground state energy of the classical Ising model.

4.1 Time complexity classes

In the case of the Ising model, we will see that sometimes, one can find highly surprising algorithms that may solve a problem efficiently that seemed not to allow for any such solution. Clearly, if we fail to solve a problem efficiently, it'd be good to have methods for proving that no fast algorithm exists – so that we don't have to waste our time trying to find one.

To this end, we define the following *time-complexity* classes.

Definition 14. *Let $f : \mathbb{R} \to \mathbb{R}$ be a monotonous function. A language L is in the class $\mathrm{DTIME}(f)$ if there exists a Turing machine T that decides L and such that $T(x)$ halts after no more than $c\, f(|x|)$ steps, where $c > 0$ is a constant and $|x|$ the length of the input.*

Common choices for f would be:

- Constant runtime: $f = \mathrm{const}$. Note that there's not even time to read the input, so this is not very relevant in practice. The language of even numbers in binary encoding can be decided in constant time: just look at the final bit.
- Linear runtime: $f(n) = n$. Read the input and do some simple calculations on the fly. Finding the Ising ground state on a tree fits this category.
- Quadratic runtime: $f(n) = n^2$. Occurs commonly in practice. Many simple sorting algorithms have this behavior (even though $f(n) = n \log n$ is achievable). Can already hurt in practice.

- High-degree polynomials $f(n) = n^6$. Few optimized, natural algorithms exhibit higher-degree polynomial runtimes. Can be *very* bad in practice.
- Quasi-polynomial: $f(n) = n^{\log^c n}$ for some constant $c > 0$. During the lecture on the 4th of November, I talked a bit about the Graph Isomorphism problem. On the evening of the 4th of November, a quasi-polynomial algorithm for GI was announced. Exciting times! `http://www.scottaaronson.com/blog/?p=2521`.
- Sub-exponential: $f(n) = 2^{n^c}$ for some constant $c \in (0, 1)$. The best known classical algorithm for integer factorization has sub-exponential runtime with $c = 1/3$. (That's the general number field sieve, which is quite hard to understand. There are much simpler algorithms achieving $c = 1/2$, e.g. the quadratic sieve).
- Exponential: $f(n) = 2^{cn}$ for $c > 0$. General optimization problems, where the best-known algorithms just enumerate all possible inputs fall into this class. An exact solution for the general problem is often hopeless in practice.

A highly important class for theoretical consideration is this:

Definition 15. *The class*

$$\mathrm{P} = \bigcup_k \mathrm{DTIME}(n^k)$$

is the set of languages that can be decided in polynomial runtime. By convention, a problem is called (computationally) tractable *if and only if it is in* P.

So, now that we have set up the formal definitions, can we start associating problems with complexity classes? E.g. is the general ising ground state problem in $\mathrm{DTIME}(n^3)$? Or in P?

The sad state of affairs is that we don't know. Never. We're not even close.

Lower bounds on runtime: The humbling truth.
Despite the most intense efforts, science has failed to develop effective methods for lower-bounding the runtime required to solve natural computational problems.

This is as bad as it sounds. Do something about it!

However, there is a second-best option. What computer science *can* do is *compare* the computational difficulty of a given problem to other, ideally well-studied problems. So while we presently cannot prove natural problems hard, we *can* often prove them to be as hard as an entire class of tasks that withstood all attempts at solving them efficiently so far. This approach – while less than ideal from a theoretical perspective – has turned out to be extremely useful in practice.

The basic tool of comparing the complexity of problems is the notion of *polynomial reduction*.

Definition 16. *Let A, B be languages. We say that $A \leq_p B$, or that A is* (poly-time) reducible *to B, if there exists a polynomial-time Turing machine T such that*

$$x \in A \Leftrightarrow T(x) \in B.$$

Thus, if $A \leq_p B$, we can turn an efficient algorithm M for B into one for A by computing $M(T(x))$. Conversely, if no efficient algorithm for A exists, but $A \leq_p B$, then it follows that there can't be an efficient algorithm for B either.

What are good problems to compare a new one to, if one wants to argue that the new problem hard? A key role in this question is played by the class NP, defined below. If the definition sounds overly technical and the concept slightly bizar: don't worry. The great utility of the class NP emerged only after many years of collective experience by the field. It is natural to be initially somewhat puzzled by it.

Informally, NP is the set of problems for which we can convince someone that we have found a solution. Formally:

Definition 17. *A language L is in* NP *if the following is true: There exists polynomials p, q and a Turing machine V such that for every $x \in \{0,1\}^*$ we have that*

$$x \in T \Leftrightarrow \exists\, u \in \{0,1\}^{p(|x|)} \text{ s.t. } V(x,u) = 1.$$

We also require that for every $u \in \{0,1\}^{p(|x|)}$, the runtime of $V(x,u)$ is bounded above by $q(|x|)$.

The machine V is referred to as the *verifier*. A string u such that $V(x,u) = 1$ is a *witness* or *certificate* for x.

Confused? Here are some examples.

- ISING, the set of coupling matrices J and energies E such that there exists a configuration of the spins with energy less than or equal to E:

  $$\{\langle J, E\rangle \,|\, E_0(J) \leq E\}.$$

  Given someone claims that a concrete pair $x = \langle J, E\rangle$ is an instance of ISING, how could they convince us of that fact? Well, an obvious certificate would be a configuration u of spins that achieves an energy not larger than E. Given x and u, we can obviously calculate the energy of u with respect to the couplings J and compare that energy to E in polynomial time. That's what the verifier would do.

- TRAVELING SALESMAN, a list of coordinates of cities C and a maximal range R for a car such that there is a route visiting all the cities not exceeding the range. A certificate would be such a route and the verification protocol is obvious.

- FACTOR, the integer factorization problem

  $$\{\langle n, m_1, m_2\rangle \,|\, \exists\, q \in [m_1, m_2] \text{ such that } q|n\}.$$

  (The relation $q|n$ is read as "q divides n" and means that q is a factor of n). A certificate is the factor q. It can be verified, since there is a poly-time integer division algorithm (not a priori obvivous, actually, but long devision taught in elementary school does the trick).

TBD: list some problems not obviously in NP, as discussed in class.

We can now state one of the most notorious open problems in all of science and math:

A million-dollar question (literally).
Decice whether $\mathrm{P} = \mathrm{NP}$.

There is an overwhelming consensus that the two classes are distinct. The lack of a proof thereof is generally interpreted as a testament to our embarassing inability of proving complexity lower bounds, rather than as an indication of equatlity. But then, who knows....

Already at this point, it should be highly intuitive that the two classes are the same. Indeed, one would believe that *finding* the solution to a problem is in general much harder than *verifying* that a solution has been found. Some examples: Not everyone who can appreciate good music is a composer (verification seems easier than generation). For a provable mathematical statement, the proof acts as a certificate of that statement's truth. Still, mathematicians keep banging their heads against walls trying to come up with proofs of many statements (or their negation). If $\mathrm{P} = \mathrm{NP}$, finding these proofs would not be significantly harder than verifying them. Anyone can identify a well-performing stock trading strategy after the fact. Can you find one?

Since the 1970s, an enormous number of problems in NP has been studied. Many of them have resisted intense efforts at finding a poly-time algorithm. Using the concept of reduction, we can leverage the joint experience of these people to argue that a given problem is indeed computationally intractable. For suppose we could show that *any* problem in NP is poly-time reducible to ours. Then in order to find an efficient algorithm for the problem at hand, we'd have to be at least as ingenious as the army of scientists who failed at all the other problems in NP before us. No shame in not living up to that standard! A priori, it is unclear that we can make such an argument for any given problem. However, it turns out that this is a suprisingly effective strategy.

Definition 18. *A language L is* NP-hard *if every problem in* NP *reduces to it: $K \leq_p L \, \forall_K \in \mathrm{NP}$.*

A language is NP-complete *if it is* NP*-hard and contained in* NP.

Again, it is not clear that there exists an NP-hard problem. In the 1970s, Cook (in the West) and Levin (in the East) independently identified one such problem: SATISFIABILITY.

To introduce it, recall that a *Boolean formula over variables $x_1, \ldots, x_n$* is an expression involving these variables and the logical operators $\wedge$ (and), $\vee$ (or), and $\neg$ (not). The variables take the values true/false (or $1, 0$). E.g. the formula $(x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$ is true if and only if $x_1 = x_2$.

The SATISFIABLITLY or SAT-problem asks: given a Boolean formula ϕ, is there an assignment to the variables $x_1, \ldots, x_n$ such that $\phi(x_1, \ldots, x_n)$ is true? It is obviously contained in NP. What is more:

Theorem 19 (Cook-Levin)**.** SAT *is* NP*-complete.*

Proving NP-hardness is an (important!) homework assignment.

Since the discovery of the Cook-Levin Theorem, thousands of problems have been proven to be NP-hard. If today, one faces a problem L that one suspects of being computationally intractable, the standard strategy is to browse the list of known NP-hard problems to find one, say A, and to show that $A \leq_p L$. As NP-hardness is clearly transitive (why?), this proves that L can be included in the list of NP-hard problems via reduction from A. The result is an entire *tree of reductions* (sometimes: *web of*

reductions) that is growing ever further. At the root of the tree, we need one problem that is proven to be NP hard by directly appealing to the definition, rather than by reduction from another one. This problem is SAT. That's what makes the Cook-Levin argument one of the most fundamental in theoretical computer science.

4.2 Ising model on trees

TBD: Intro, algorithm for trees.

4.3 Graph Theory, Perfect Matchings, and the planar Ising model

This chapter has three purposes: We will give an introduction to basic notions of graph theory, which play a prominent role not only in computational complexity, but in many areas of discrete math; We will appreciate that there are cases where a seemingly hard problem can fall to a highly non-obvious algorithm; and lastly, we will complete the positive part of our study of the Ising model.

We start by introducing some of the basic graph-theoretic notions that will be important in what folllows.

A *graph* is a collection of vertices V and edges $\{i, j\}$ for $i, j \in V$. It can be visualized by representing every vertex as a point in space and connecting those vertices i, j for which there is an edge $\{i, j\}$ in E. Edges can be associated with a *weight*. A *weigthed graph* is a graph, together with a *weight function* $w : E \to \mathbb{R}$ that maps edges to real numbers. An example would be a network of streets, which are weighted by capacity or maximal allowed speed. A *directed graph* (or *digraph*) is a graph where the edges are *ordered pairs* $(i, j) \neq (j, i)$ instead of sets $\{i, j\}$.

A graph is *planar* if it can ben drawn in the two-dimensional plane without any edges crossing. A planar representation of a graph divides the plane into cells that are surrounded by a cycle of edges and vertices. The subgraph enclosing such a region is called a *face*. The cycle bounding the outside is the *outer face*.

The combinatorial data of a graph can be represented by an *adjacency matrix* A which has $A_{i,j} = 1$ if $\langle i, j\rangle \in E$. If the graph is weighted, one may also put the weight of the link into the adjancency matrix. These matrices form the basis of a poweful link between graph theory and linear algebra, which we will partly touch on in the following.

A graph is *bipartite* if one can color every vertex either white or black, such that there is no edge connects vertices of the same color. Below, we will prove some statements that hold for any planar graph only for *bipartite* planar graphs. The arguments are slightly more simple, without any essential detail missing. Also, from a physical perspective, the class of planar bi-partite graphs include the important case of the 2D lattice. The adjanceny matrix of a bipartite graph can be brought into block form, by first listing the "white" and then the "black" vertices.

A *perfect matching* of a graph G is a subset $M \subset E$ of edges such that any vertex is contain in one of these edges. One commonly invokes the analogy of the vertices representing people and edges specifying two can get along with each other. A perfect matching would thus divide up the population into working couples. In the case of bipartite graphs (and belaboring anachronistic gender stereotypes), every vertex can be thought of as having a sex (say black is male and white is female), so the matching analogy carries even further. Two obvious problems suggest themselves:

Existence/Counting of perfect matchings.

Given a graph G, does there exist a perfect matching?
Given a graph G, how many perfect matchings do exist?

Both questions turn out to be very difficult in general (we'll see this later), but also turn out to have a surprising algorithm that places them in P (we'll see that momentarily). There is a "weighed" version of both problems. If G is a weighted graph, the weight of a matching M is the product

$$\prod_{\{i,j\}\in M} w(\{i,j\})$$

of the pairs in the matching. The analogue of the questions above is then to determine the sum of all matchings or to decide whether there is a matching of maximimal or minimal given weight.

We aim to prove the following:

Theorem 20. *Let G be a planar bipartite graph, then the problem of counting all perfect matchings is in* P.

(As indicated before, "bipartite" isn't necessary, but makes the argument slightly simpler. Also, a version for weighted graphs can easily be established along the same lines as the proof presented.)

The first step is to map the problem of counting perfect matchings to the problem of counting yet another graph-theoretic structure: *cycle covers*.

Let G be a *directed* graph. A *cycle* is a closed sequences of edges that can be transversed along the orientation of G (see Fig. **??**). A *cycle cover* of G is a set of cycles such that every vertex lies on one of them. Also, if G is an undirected graph, then $\overleftrightarrow{G}$ is the graph obtained by replacing every undirect edge $\{i,j\}$ of G with the two directed edges $\langle i,j\rangle$ and $\langle j,i\rangle$.

Here's the first technical lemma:

Lemma 21. *Let G be an undirected palnar bipartite graph. There is a 1-1 map between ordered pairs of perfect matchings $\langle M_1, M_2\rangle$ of G and cycle coveres of $\overleftrightarrow{G}$.*

In other words

$$|\{\text{ perfect matchings of } G\ \}| = |\{\text{ cycle covers of } \overleftrightarrow{G}\ \}|^{1/2}.$$

As we'll find an efficient algorithm for counting the latter, we'll also be able to count the former.

Proof. The proof is constructive, see also Fig. **??**.

Pair of Matchings $\to$ cycle cover. Step 1: Replace the joint edges of M_1, M_2 by loops. I.e. if $\{i,j\} \in M_1 \cap M_2$, then put both $\langle i,j\rangle$ and $\langle j,i\rangle$ into the cycle. Step 2: Those edges in M_1 but not in M_2 are oriented from black to white. The edges in M_2 but not in M_1 are oriented from white to black. The result is a cycle cover, because every vertex has one incoming and one outgoing edge.

Cycle cover $\to$ pair of matchings. Put all edges that are oriented from black to white into M_1 and all edges oriented from white to blac into M_2.

It's now simple to verify that these two maps are inverses of each other. □

Now let's relate the number of cycle covers of a graph to a linear-algebraic quantity.

Definition 22. *Let A be an $n \times n$-matrix. It's* permanent *is defined to be*

$$\operatorname{perm} A = \sum_{\sigma} \prod_{i=1}^{n} A_{i,\sigma_i},$$

where the product is over all permutations $\sigma \in S_n$ of the n indices.

Note that the definition bears striking resemblence to the one of the *determinant*

$$\det A = \sum_{\sigma} \operatorname{sign} \sigma \prod_{i=1}^{n} A_{i,\sigma_i},$$

differing only by the ± 1-factor of $\operatorname{sign} \sigma$. Now (perhaps surprisingly) the determinant can be computed in polynomial time (how?). The permanent, on the other hand does not have an efficient algorithm unless $\mathrm{P} = \mathrm{NP}$ (and evern worse things happen, see later). We will, however, use the connection below to construct an algorithm for computing the permanent of particular adjacency matrices. Why do we care? That's why:

Lemma 23. *Let G be a planar bipartite graph with adjacency matrix A. Then* $\operatorname{perm} A$ *equals the weighted sum of cycle covers of $\overleftrightarrow{G}$.*

Proof. As mentioned above, we can put the adjacency matrix into block-diagonal form

$$A = \begin{pmatrix} 0 & B \\ B^T & 0 \end{pmatrix},$$

with B an $n \times n$-matrix such that $B_{i,j} = 1$ if the ith white vertex connects to the jth black one. From the bloc-diagonal form, we see that

$$\prod_{i=1}^{2n} A_{i,\sigma_i} \tag{2}$$

is zero, unless σ maps the black vertices (listed second) to the white vertices (listed first). So we get non-zero contributions only for those permutations of the form $\mu\,\pi$, where

$$\mu = (1, n)(2, n+2) \ldots (n, 2n)$$

just maps the ith white to the ith black vertex and $\pi \in S_n$ permutes the white vertices amongst each other.

Recall from elementary group theory that every permutation π can be written as the product of disjoint (group-theoretic) *cycles*:

$$\pi = c^{(1)} \ldots c^{(k)}, \qquad c^{(r)} = (c_1^{(r)}, c_2^{(r)}, \ldots, c_{l_i}^{(r)}).$$

Now

$$\prod_{i=1}^{2n} A_{i,\sigma_i} = \prod_{i=1}^{n} B_{i,\pi_i}$$

is non-zero if every group-theoretic cycle corresponds to one graph-theoretic cycle. Because the product is over *all* vertices, we thus get 1 if the permutation corresponds to a cycle cover and zero else. Since both types of cycles do not depend on the vertex at which they start, but do depend on the orientation (unless the length is 2), there is a 1-1 correspondence between permutations that lead to a non-vanishing contribution and cycle covers. □

Now comes the crucial step. We'd like to weigh every matrix element $A_{i,j}$ by ± 1 in such a way that the resulting weighted matrix A' satisfies

$$\prod_{i=1}^{2n} A'(i,\sigma_i) = \operatorname{sign}\sigma \prod_{i=1}^{2n} A_{i,\sigma_i},$$

as in this case,

$$\operatorname{perm} A = \det A',$$

We could then compute the permanent using the well-known efficient algorithm for determinants! Of course, in general, there is no hope that such a weighting actually exists, much less that we can find it in polynomial time. For the planar case, Kasteleyn however did manage to find such a scheme in the 50s.

That's the rough strategy: We construct a directed version $\overrightarrow{G}$ of G with the property that *every even cycle of $\overleftrightarrow{G}$ that comes out of a cycle cover contains an odd number of edges co-oriented with $\overrightarrow{G}$*. We then define the *skew adjacency matrix*

$$(A')_{i,j} = \begin{cases} +1 & \langle i,j\rangle \in \overrightarrow{E} \\ -1 & \langle j,i\rangle \in \overrightarrow{E} \end{cases}$$

Then the product of A'_{i,σ_i} over any even cycle will be odd – as is the sign of any even (group theoretical) permutation. Thus, A' is the desired weighted matrix.

Now for the construction of the orientation. Let's consider the lattice as an example (Fig. **??**). Every face is in particular an even-length cycle. One can see explicitly in the figure that every face has an odd number of clockwise (cw) edges and thus an odd number of co-oriented edges, no matter in which direction we go around it. The central technical insight is that in the planar case, verifying the orientation on faces is sufficient.

Note that we only need to establish the property for cycles which enclose an *even* number of vertices. That's because in a cycle cover, any two vertices have to be paired up. But because the graph is planar, vertices in the interior of the cycle can't be matched up with those outside of it. But as every cycle in a bi-partite graph is even-length, this must also be true for each cycle in the interior.

Theorem 24. *Let G be a bipartite planar graph. Assume that every face has an odd number of cw edges. Then the number of cw edges in a cycle is odd if the number of enclosed vertices is even and vice versa.*

Proof. The proof is by induction on the number of faces enclosed by a cycle.

Let Γ_n by a cycle for which the statement is assumed to be true, let Γ_1 be an adjacent face and let Γ_{n+1} the cycle resulting from incorporating Γ_1 into Γ_n. Let $\mathrm{cw}(\Gamma)$ be the number of cw edges in a cycle. Then

$$\begin{aligned} \mathrm{cw}(\Gamma_{n+1}) &= \mathrm{cw}(\Gamma_n) + \mathrm{cw}(\Gamma_1) - \mathrm{cw}(\Gamma_n \cap \Gamma_1) - \mathrm{ccw}(\Gamma_n \cap \Gamma_1) \\ &= \mathrm{cw}(\Gamma_n) + \mathrm{cw}(\Gamma_1) - |\Gamma_n \cap \Gamma_1|. \end{aligned}$$

The number of enclosed vertices is

$$\mathrm{vert}(\Gamma_{n+1}) = \mathrm{vert}(\Gamma_n) + |\Gamma_n \cap \Gamma_1| - 1.$$

As Γ_1 is a face, it follows by assumption that $\mathrm{cw}(\Gamma_1)$ is odd. Thus

$$\mathrm{cw}(\Gamma_1) - |\Gamma_n \cap \Gamma_1| \quad \text{and} \quad |\Gamma_n \cap \Gamma_1| - 1.$$

have the same parity (the opposite parity of $|\Gamma_n \cap \Gamma_1|$). Therefore, parity of the number of cw edges and the parity of the number of enclosed vertices change in the same manner as $n \to n+1$. □

The final step now is to show that one can always have an odd cw-orientation on every face. But that's trivial: Start orienting any face you like. Then add face by face. Every new face comes with at least one new edge and you can orient it in such a way to ensure that the odd number constraint is met.

Thus we have shown the (unweighted, bipartite version of)

Theorem 25. *Let G be a planar weighted graph. Then there is a polynomial-time algorithm that counts the weighted perfect matchings in G.*

We now show how computing partition functions of the Ising model on a regular 2D lattice reduces to suming weighted perfect matchings in a planar graph.

Let J be an Ising interaction matrix on a regular 2D lattice G. We will set up a 1-1 correspondence between pairs of equivalent spin configurations and perfect matchings in a certain planar graph $\tilde{G}$. This other graph is built in two steps: First by passing from G to its *dual graph* G^*, and then by substituting the vertices of G^* by certain *gadgets*, as explained below.

1. *Pairs of configurations → sets of unsatisfied edges*. Remember that in order to minimize the energy

$$H(\sigma) = -\sum_{\{i,j\}\in E} J_{i,j}\sigma_i\sigma_j,$$

we'd ideally want that every summand $J_{i,j}\sigma_i\sigma_j$ is positive. Of course, in the presence of frustration, this is impossible to achieve for all summands simultaneously. Call an edge $\{i,j\}$ *unsatisfied* if

$$J_{i,j}\sigma_i\sigma_j < 0.$$

We can write the energy of a configuration σ as a function of the set of unsatisfied edges $U(\sigma)$:

$$H(\sigma) = -\left(\sum_{i,j}|J_{i,j}|\right) + 2\sum_{\{i,j\}\in U}|J_{i,j}|.$$

2. *Unsatisfied edges → pairs of configurations*. It is clear that we can recover the spin configuration up to a simultaneous reversal of all spins $\{\sigma, \bar{\sigma}\}$ from the set of unsatisfied edges $U(\sigma)$ (why?). However, we can do more: namely we can describe all sets U that can appear as the set of unsatisfied edges of a spin configuration. With this characterization at hand, we can describe the entire problem in terms of U, which turns out to be advantageous. The constraints on valid U's are this:

Lemma 26. *Let $U \subset V$ a set of edges of a regular 2D lattice. There is a spin configuration σ such that U are the unsatisfied edges if and only if for every face, the parity of the number of unsatisfied edges matches the parity of the number of edges with negative couplings $J_{i,j} < 0$.*

Proof sketch. 1. $\sigma \to U$: Simple. List cases and remember that both the number of edges around a face is even and that also the number of anti-parallel spins along a cycle must be even.

2. $U \to \sigma$: Constructive. Start assigning spins from top-left to bottom-right. Notice that there are at most two ways of connecting each new spin to the already assigned ones. That requires one consistency condition, which is exactly the one given. □

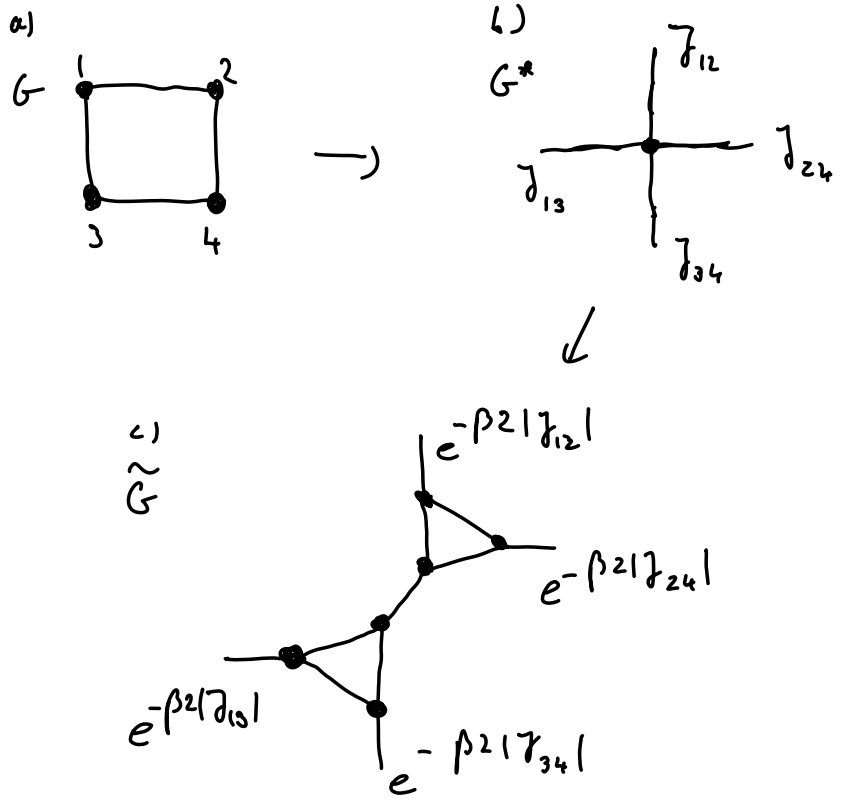


Figure 5: Construction of $\tilde{G}$ via suitable *gadgets*. a) An even face from the entirior of the lattice. b) The corresponding subgraph of the dual graph G^*. The face has become a vertex and the edges drawn extend to the adjacent dual vertices. They are weighted the same way as the vertices they cross. c) Now pass to $\tilde{G}$. The dual vertex is replaced by nine vertices as shown. The weights are chosen as indicated, where $\beta \in \mathbb{R}$ is an arbitrary number (interpreted later as inverse temperature). Edges without an indicated weight are assigned the weight 1.

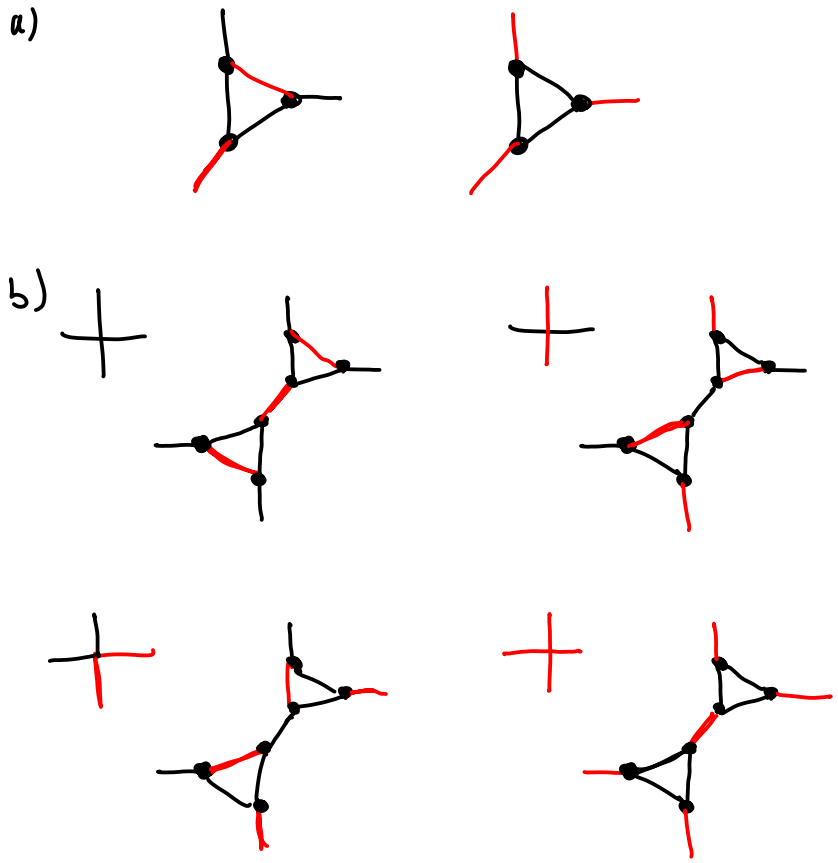


Figure 6: The basic *gadget* is the triangle shown in Figure (a). There is exactly one perfect matching for each odd configuration of outgoing edges. Indeed: If the matching connects two internal vertices, then the third one has to be paired up with an external vertex (left, one outgoing edge). If none of the edges are paired up, each one has an external partner (right, three outgoing edges). But there is no way of pairing up the three internal vertices, so these exhausts all the possibilities. (b) Combining these basic gadgets for internal even faces.

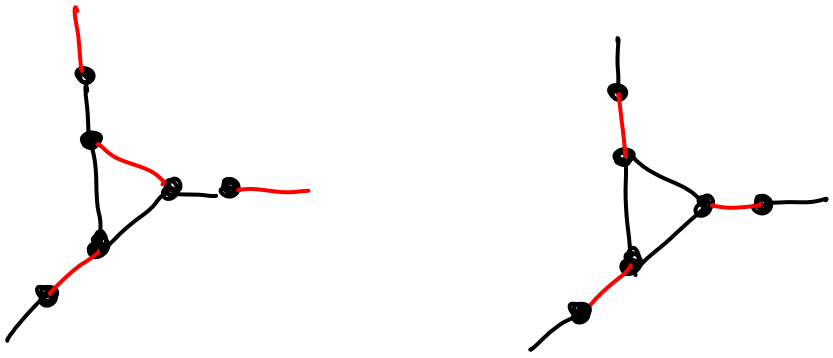


Figure 7: Compared to Figure 6(a), additional vertices on the outgoing lines act as "inverters". We thus obtain a gadget that enforces an *odd* number of outside pairings. Use these for odd faces.

3. Now we construct a new graph $\tilde{G}$ in two stages. First, the *dual graph* G^* (Fig. **??**) of a plane graph is a graph whose vertices are the faces of G. Faces are connected by a dual edge if they share a primal edge. Finally, we modify G^* to get a new graph $\tilde{G}$ in such a way that every subset U as above corresponds to a perfect matching in G^*. To this end, we employ so-called *gadgets*. The word gadget is a loosely defined term that stands for a construction that is "native" to the problem we want to prove hard (a subgraph, a Hamiltonian term, a clause...), but that encodes a concept of the problem we are reducing from (a match, an increment of the state of a Turing machine...). In our case, we will replace even/odd degree vertices of the dual graph by subgraph gadgets as detailed in Figs. 5, 6, 7. The resulting graph is $\tilde{G}$.

Now every set of unsatisfied edges U corresponds to a perfect matching M of $\tilde{G}$ with weight

$$\begin{aligned} w(M) &= e^{-\beta \sum_{\{i,j\}\in U} 2|J_{i,j}|} \\ &= e^{-\beta\left(H(U)+\sum_{i,j}|J_{i,j}|\right)} \\ &= e^{-\beta H(U)} e^{-\beta \sum_{i,j}|J_{i,j}|}. \end{aligned}$$

And thus

$$\begin{aligned} \sum_M w(M) &= e^{-\beta \sum_{i,j}|J_{i,j}|} \sum_U e^{-\beta H(U)} \\ &= e^{-\beta \sum_{i,j}|J_{i,j}|} \frac{1}{2} Z(J), \end{aligned}$$

where $Z(J)$ is the partition function of J and we encouter a factor of two because every set of unsatisfied edges U corresponds to *two* equivalent spin configurations.

We conclude

Theorem 27. *The partition function of the ising model on a 2D regular lattice can be efficiently computed from the sum of weighted matchings of a plane graph. The latter, in turn, can be efficiently transformed into the determinant of a weighted adjacency matrix. In total, the partition function can be evaluated in polynomial time.*

Exercise: Show that the ground state energy can be efficicently extracted from the partition function.

4.4 Hard instances of the Ising model

Now for the converse. We will show that in general, the Ising ground state cannot be efficiently calculcated. A chain of reductions that works would be

$$\begin{array}{c} \mathrm{NP} \\ \Big\downarrow_p \\ \textsc{Sat} \\ \Big\downarrow_p \\ \textsc{3Sat} \\ \Big\downarrow_p \\ \textsc{Subset Sum} \\ \Big\downarrow_p \\ \textsc{Partition} \\ \Big\downarrow_p \\ \textsc{MaxCut} \\ \Big\downarrow_p \\ \textsc{IsingGroundState}. \end{array}$$

It may not be the most economic way of arriving at the result, but we'll visit plenty of famous problems along the way and none of the reductions is really hard (except the first one, but this was already a homework).

Let's introduce the problems encounted along that path.

3SAT.—The *3-literal conjunctive normal form Boolean satisfiability problem*. Definitions: let x_i be Boolean variables. The expressions $x_i, \bar{x}_i = \neg x_i$ are called *literals*. A *disjunction* is a logical or, a *conjunction* a logical ane. A *clause* is a disjunction of literals (e.g. $(x_1 \vee x_3 \vee \bar{x}_7)$). A formula ϕ is in *conjunctive normal form* if it is expressed as a conjunction over clauses. A formula is in *3-CNF* if it is in conjunctive normal form and every clause includes exactly three literals. With these notions, 3SAT is the language of satisfiable formulas in 3CNF. It's NP-complete (unlike 2SAT, which is in P).

SUBSET SUM.—A tuple $\langle a_1, \ldots, a_n \rangle$ of integers and a number T such that there is a subset $S \subset \{1, \ldots, n\}$ with

$$\sum_{k \in S} a_k = T.$$

PARTITION.—A tuple $\langle a_1, \ldots, a_n \rangle$ of integers such that there is a subset $S \subset \{1, \ldots, n\}$ with

$$\sum_{k \in S} a_k = \sum_{k \in S^C} a_k.$$

MAX CUT.—Let G be a weighted graph. A *cut* is just a subset $S \subset V$ of vertices of G. It's *weight* is the sum of the weights of the edges connecting S with the rest

$$w(S) = \sum_{\{i,j\} \in E, i \in S, j \in S^C} w(\{i,j\}).$$

The language MAC CUT is the set of weighted graphs and numbers T such that there exists a cut with weight at least T.

We'll demonstrate one of the (less-trivial) reductions:

Lemma 28. PARTITION *reduces to* 3SAT.

Proof. TBD: type up. □

5 Classes beyond P/NP: Polynomial hierarchy, probabilistic computation

5.1 Polynomial hierarchy

Here, we'll encounter an entire hierarchy of complexity classes that generalize P and NP. It will allow us to resolve differences in (conjectured) hardness of problems that P/NP are not fine enough to resolve.

Recall that the class NP consists of languages with succint proof of membership. I.e. $L \in \mathrm{NP}$ if there is a polynomial p and a poly-time Turing machine M such that

$$x \in L \Leftrightarrow \exists_p y \text{ s.t. } M(x,y) = 1.$$

From now on, we use the notation "$\exists_p y$" to denote the statement there there is a y of length $p(|x|)$. Likewise, coNP is the complement of NP, i.e. the set of languages with succinct proof of non-membership. Concretely, $L \in \mathrm{coNP}$ if

$$x \notin L \Leftrightarrow \exists_p y \text{ s.t. } M(x,y) = 1$$

which is equivalent to

$$x \in L \Leftrightarrow \forall_p y \text{ s.t. } M'(x,y) = 1,$$

for $M' = \neg M$. Let's also recall an example. Start with ISINGGS

$$\text{ISINGGS } = \{\langle J, E\rangle \,|\, \exists \sigma \text{ s.t. } H_J(\sigma) \leq E\} \in \mathrm{NP}\,.$$

In coNP, we find its complmement

$$\text{ISINGGS}^{\,C} = \{\langle J, E\rangle \,|\, \forall \sigma,\ H_J(\sigma) > E\} \in \mathrm{coNP}\,.$$

Neither version formalizes the most natural question, however. That would be: What is the exact ground state energy of the given model? Formally:

EXACTGS	
INPUT:	coupling matrix J
OUTPUT:	ground state energy $E_0(J) = \min_\sigma H_J(\sigma)$.

(As usual, we could turn it into a set of decision problems, one for each bit of the ground state energy. We will not be overly pedantic concerning this distinction.) It is clear that EXACTGS is at least as hard to compute as ISINGGS and its complement. In fact, it makes a statement both about the "existential" and the "for all" quantifier parts:

$$E = \text{EXACTGS}(J) \Leftrightarrow \exists_\sigma \forall_{\sigma'} \big(H_J(\sigma) = E\big) \wedge \big(H_J(\sigma') \geq E\big).$$

Similar nested quantifiers also appear in in other natural languages. For example, set

$$\text{MIN-EQU} = \{\langle \psi \text{ Boolean formula }, k \in \mathbb{N}\rangle \mid \exists \phi \text{ Boolean formula of length } k \wedge \phi = \psi\}\,.$$

Generalizing the notions of NP and coNP, the

5.2 Randomized Turing Machines

It is a plausible assumption that there exists a physical mechanism that produces a *random number* $r \in \{0, 1\}$. A toss of a fair coin, for example, or maybe a polarization measurement on a photon that has passed a first polarizer enclosing a 45 angle with the second one (Fig. **??**). It's a deep philosophical question what exactly we *mean* by *random*. We won't go into that problem here, but we'll see that quantum mechanics has a lot to say about it.

In any case, the realization above seems to put the Church-Turing thesis in jeopardy: A TM as introduced before is completely deterministic, so it can't emulate a coin toss. That would be a problem if tossing coins helps in solving computational problems.

Does it? An example that suggests the answer might be "yes" is given by *probabilistic primality tests*.

Let PRIMES be the language of prime numbers. Lacking access to a quantum computer, we don't know how to factor integers efficiently, so the most naive algorithm for checking primality (i.e. just compute the prime decomposition and count terms) won't place PRIMES in P. The language is important for practical purposes, though. As we will see later, the popular public key cryptography protocol RSA has publick keys $n = pq$, where p and q are large (and secrete) primes. The presumed inability to factor n quickly, prevents one to turn the public key n into the private key p, q. But in order to generate the key pair in the first place, we have to find two large primes p, q. This is done by choosing random big numbers and checking whether they are, in fact, prime. (Homework: Check the crypto library of OpenSSL on github to see how this is done).

Thus, the RSA protocol relies as much on our ability to identify prime numbers, as it does on our inability to factorize. To this end, one relies that there are certain simple formulas that are satisfied by prime numbers, but not necessarily by composite ones. One simple such property is given by *Fermat's Little Theorem*.

Lemma 29 (Fermat's Little Theorem)**.** *Let n be prime and a any integer. Then $a^{n-1} = 1 \mod n$.*

There are many proofs (none of which you need to understand going forward). If you know some elementary group theory, then this is a simple way of understanding the result: If n is prime, then every number $a \in \mathbb{Z}_n^* = \{1, \ldots, n-1\}$ has a multiplicative inverse mod n. Thus $\mathbb{Z}_n^*$ forms a group under the multiply-mod-n operation. But the group has size $n-1$. By Lagrange's theorem, the order of any element must divide the order of the group.

Anyway, you can use Mathematica to choose a few random a's and n's and test whether the above relation always holds. You'll find: it does not. So this gives a way to *witness* compositness of a number n.

Definition 30 (Compositeness Witness)**.** *Let n be a positive integer and a such that $a^{n-1} \neq 1 \mod n$. Then a is a* (Fermat) witness *for the composite nature of n.*

It's now natural to ask whether any composite n has such a witness. Unfortunately, the answer is negative. A infinite number of composites that can't be detected this way exists. They are known as *Charmichael numbers*. However, it is true that there are a refined tests $T(a, n)$ one can perform (using the fact that if n is prime there is at most one square root of any a modulo n) such that i) If n is prime, $T(a, n)$ is true for every integer a and ii) for at least three quarters of all integers $a < n$, the test $T(a, n)$ fails. This is the *Miller-Rabin primality test*. (It's actually very elementary – consult Wikipedia (or OpenSSL's source code) if you want to know the details).

In any case, while a majority of a's witness the compositeness of n, there seems to be no algorithmic way of determining *which ones*. If we have access to a random number generator, this does not need to worry us. Just repeat the test k times with random a's. If n is composite, the probability of it not being witnessed by any given a is $p_f = \frac{1}{4}$, thus the probability that it won't be witnessed at all is

$$p_f^k = \left(\frac{1}{4}\right)^k = 2^{-k \log 4},$$

which is soon smaller than the probability of a meteor hitting the comptuer's operator and rendering the calculation the least of their problems. At the same time, there is *no* obvious poly-time deterministic primality test. (Afer many decades of being open, primality testing was placed in P in 2004 (alas, I can remember....). However, the deterministic algorithm is sufficiently complicated to be useless in practice).

So randomness might help. Well, in fact, there is also evidence suggesting that the benefits of randomness are limited. We won't discuss this here in great detail, and in any case, the problem is not settled. In the meantime, we should define a model of a probablistic Turing machine and define related complexity classes so that we can at least *talk about* the potential power of randomness.

Definition 31. *A* probabilistic Turing machine *is a Turing machine with two transition functions δ_0, δ_1. At every step in the computation, a fair coin is tossed and according to the outcome, one or the other function is used.*

While we don't know what the power of probabilistic Turing machines relative to deterministic ones are, we give a name to the set of problems solvable in probabilistic polynomial time.

Definition 32. *A lanaguage L is in* BPP*, or* bounded probabilistic polynomial time*, if there is a Turing machine M such that*

$$\text{Prob}\left[M(x) = L(x)\right] \geq \frac{2}{3}.$$

and a polynomial p such that $M(x)$ halts with probability 1 after $p(|x|)$ steps.

Note that in the definition, we require the machine to take polynomial time with probability one, rather than *expected polynomial time*. The latter would also make sense. Also, do not confuse *probabilistic* Turing machines (which exist) with *non-deterministic* Turing machines (which are just conceptual devices). The danger of confusion is all the more pronounced as a non-probabilistic TM would usually be called determinsitc. Thus non-deterministic is *not* the opposite of deterministic...

In this language, we have argued before that PRIMES $\in$ BPP.

While the primes example suggested that randomness might aid computations, there are also many hints that ultimately, the utility of randomness is limited. In fact,

many researchers now believe that $\mathrm{BPP} \stackrel{?}{=} \mathrm{P}$. One of the intuitions behind that conjecture is that there exists (cryptographically motivated) pseudo-random number generators of sufficient qualitty that one can just use their output rather than true random numbers. We won't go into this discussion in any detail, but will limit the power of BPPbelow.

There is an alternative characterization of BPP, which makes the randomness more explicit.

Lemma 33. *A language L is in* BPP*if and only if there exists a polynomial p and a* deterministic *TM M such that*

$$\mathrm{Prob}_u\left[M(x,u) = L(x)\right] \geq \frac{2}{3},$$

where u is drawn uniformly from $\{0,1\}^{p(|x|)}$.

Above M is a deterministic TM, which gets endowed with a supply of $p(|x|)$ random bits u at the start and can – quite obviously – simulate a probabilistic TM.

The value $\frac{2}{3}$ in the definition is rather arbitrary. In fact, one can use a simple trick called *probability amplification* to achieve an exponentially small probability of error. The protocol is simple enough. For a given x, just run $M(x)$ several times – say k times – and output whichever conclusion was reached by the majority of runs. One can use a simple large-deviation bound to show that the probability that the majority gets it wrong is exponentially small in k (see c.f. TBD). A more qualitative argument is sketched in Fig. **??**.

With these tools, we can prove a surprising and non-trivial result:

Theorem 34. $\mathrm{BPP} \subset \Sigma_2^p \cap \Pi_2^p$.

In other words: While some people conjecture that BPPis equal to Pand thus equal to the zeroth level of the polynomial hierarchy, we can at least show that it is contained int he 2nd level.

The proof uses the *probabilistic proof method*. That means that we'll establish the existence of a certain construction by showing that we can design a random process that outputs it with positive probability. Since the probability of obtaining a result is larger than zero, in particular, such a structure has to exist. The proof gives us no indicication of how to actually obtain a deterministic solution. Such probabilistic proofs play an increasingly important role in mathematics. In some areas – e.g. *coding theory* – probabilistic constructions are extremely central and far outperform what can actually be explicitly constructed. It pays to understand the nature of the proof below. Please note that we are using randomness to reason about the power of randomized Turing machines. While the two uses of randomness may be related on a conceptual level, they should not be confused.

Proof. Let $L \in \mathrm{BPP}$. Assume we have used randomness amplification such so that

$$\mathrm{Prob}_u[M(x,u) = L(x)] \geq 1 - 2^{-n}.$$

This is possible using polynomially many bits of randomness, say $|u| = p(|x|)$ many.

Fix a particular input x, let $n = |x|, m = |u|$. Let $A_x \subset \{0,1\}^m$ be those bit strings that cause M to accept

$$A_x = \{u \,|\, M(x,u) = 1\}.$$

We will prove the following lemma:

Lemma 35. *Set* $k = \lfloor \frac{m}{n} \rfloor + 1$.

Assume $x \in L$. *Then there exists* $u_1, \ldots, u_k \in \{0,1\}^m$ *such that*

$$U := \bigcup_{i=1}^{k} A + u_i = \{0,1\}^m.$$

Conversely, if $x \notin L$, *no such* $u_1, \ldots, u_k$ *exist.*

□

5.3 Interactive Proofs

An *interactive proof protocol with* k rounds.

. . . completeness . . . . . soundness . . .

. . . TBD . . .

Note that the previous protocols cruicially relied on the fact that the random bits of the verifier were *secret*. More technically speaking, the function g computer by the prover had to depend only on the input received from the verifier, not on the random bits.

It's a highly non-trivial realization that there is a way of certifying graph non-isomorphism even if the random bits known to the verifier. We will establish this result here. It will by far by the most advanced classical complexity result we'll treat in this course. So pay attention (and don't worry too much if not all details stick).

Definition 36 (AM)**.** *A language* L *is in the complexity class* AM*(Arthur-Merlin) if membership can be certified using the following protocol: The verifier (Arthur) sends a polynomial number of random bits* u *to the prover (Merlin). Merlin responds with a polynomial-sized answer* a. *Arthur then computes the poly-time verificiation function* $V(x, u, a)$ *which is required to fulfill*

$$\text{Prob}[V(x, u, a) = L(x)] \geq \frac{2}{3}$$

6 Convex optimization, marginals, Bell tests

. . . a lot to be typed up. . .

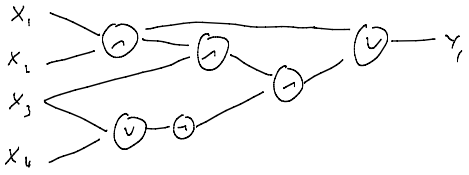


Figure 8: A Boolean circuit representing the formula $y_1 = ((x_1 \wedge x_2) \wedge x_3) \wedge (\neg(x_2 \vee x_4)) \vee (x_1 \wedge x_2)$. The graphical representation is clearly easier to parse.

Part II
Physics for computer science

7 Quantum Computing

Recall that the strong Church-Turing thesis states that a problem that can be solved by any physical process, can be solved on a Turing machine with at most a polynomial slowdown. Physical processs relying on quantum mechanics are the first candidates to challenge this intuition. There is now strong theoretical evidence that highly controlled quantum systems can exponentially outperform classical computers for some some computational problems. There is no unconditional *proof* of this, partly because finding one would presumably imply proving that a given problem not only has a poly-time quantum algorithm (that's doable), but that there is no poly-time classical one. As we have emphasized time and again, we are lacking the theoretical tools to unconditionally prove problems hard.

The purpose of the following sections is to introduce some important algorithms, theoretical results, and physical implementations.

7.1 Gate Model

The paradigmatic model of classical computer is the Turing machine. Since all known classical computational models are equivalent, the theory could take a different model as the defining one. The honor was extended to the Turing machine partly for historical reasons and partly because it uses very little structure which makes reductions – as encountered in the Cook-Levin-Theorem – easier.

While the concept of a *quantum Turing machine* has been defined, it is not the model most commonly used. Rather, again partly for historical reasons, the *circuit model* or *gate model* is usually employed. It is a mathematical model for the electronic logic gates that comprie contemporary computers.

We start by introducing the classical version. The concept is much simpler than the formal definition might suggest – see Fig. 8.

Definition 37. *A* Boolean circuit *is a directed acyclic graph. Vertices with no incoming edges are* sources *and represent input Boolean variables. Vertices with no outgoing edges represent* sinks *and are associated with output Boolean variables. All inner vertices (i.e. neither sources nor sinks) are labeled by either $\wedge$ (logical and), $\vee$ (logical or), or $\neg$ (logical not). The "and" and "or" vertices have* fan in 2*, i.e. two incoming vertices. The "not" vertices have fan in 1.*

Each Boolean circuit defines a Boolean formula in the obvious way (Fig. 8).

What is the computational power of circuits? In the abscence of an answer, we do what theoretical c.s. does: define a new complexity class that captures the set of languages that can be decided by circuits whose complexity is bounded as a function of the size of the input bit string.

Definition 38. *Let L be a language, $f : \mathbb{N} \to \mathbb{N}$ a function. We say that L is the the complexity class* $\mathrm{SIZE}(f)$ *if for every $n \in \mathbb{N}$, there exists a circuit C_n with one of size $f(n)$ and one outputbit such that for every $x \in \{0,1\}^{\times n}$ it holds that $x \in L \Leftrightarrow C_n(x) = 1$.*

The class $\mathrm{P}_{/\,\mathrm{poly}}$ is the set of languages $\mathrm{P}_{/\,\mathrm{poly}} = \bigcup_k (n^k)$ of languages decidable by poly-size circuits.

(The origin of the name $\mathrm{P}_{/\,\mathrm{poly}}$ will become clear momentarily).

What is the relation between languages with a poly-sized circuits and languages decidable by a poly-time Turing machine. It turns out that, due to a technical reason, the former is much more powerful than the latter.

First, observe that $\mathrm{P} \subset \mathrm{P}_{/\,\mathrm{poly}}$ – i.e. every language that is decidable by a poly-time TM has a poly-sized circuit. This fact follows directly from the Cook-Levin construction, where the fact that a TM exists which accepts a given input string is cast into the form of a Boolean formula. These, in turn, can be expressed as a Boolean circuit.

The converse is not true. The reason for the gap between $\mathrm{P}_{/\,\mathrm{poly}}$ and P is that in the Turing machine case, we required there be *one* Turing machine that decides all instances, whereas in the circuit case, we allowed for a *different circuit* for every input size n. A single finite Turing machine need not be capable of producing the optimal circuit for every n. Case in point: the language

$$\textsc{HaltingOnes} = \{1^{\times k} \mid k \text{ is Turing number of Halting Turing machine } \}$$

of bit strings that consists of a number k of ones such that k is the Turing number of a Halting Turing machine is clearly not decidable by a TM. However, $\textsc{HaltingOnes} \in \mathrm{P}_{/\,\mathrm{poly}}$: for every k that corresponds to a halting TM, there clearly is a poly-sized circuits which checks whether the input is all 1's. Also, for all other k's, there exists a (trivial) circuit, which outputs 0 for all inputs.

So while $\mathrm{P}_{/\,\mathrm{poly}}$ does contain undecidable languages, it should be clear from the example that this is a technical artifact of the definition. It goes away if one requires that the circuit be efficiently constructable by a TM on input of n (clearly, that is not the case for the HALTINGONES-circuit described above). Circuit families with that property are called *uniform*. Now it should be clear that the set of languages decidable by uniform poly-sized circuits is equal to P. The above discussion mainly serves the purpose to warn us against making naive definitions in the context of computational models.

The name $\mathrm{P}_{/\,\mathrm{poly}}$ stands for "poly-time, with *polynomial advice*". The "advice" being the circuit that helsp solve the problem.

With a theroetical understanding of circuits at hand, let's proceed to introduce *quantum circuits*.

7.2 Simple circuits: Teleportation and Deutsch-Josza Algorithm

7.3 Shor's Algorithm & Cryptogrpahic Key Exchange

References

[1] Scott Aaronson. Guest Column: NP-complete Problems and Physical Reality. *SIGACT News*, 36(1):30–52, 2005.