

Week 10

Complexity Theory

- 1 Introduction
- 2 Logical puzzles and SAT
- 3 The complexity classes P and NP
- 4 Beyond NP
- 5 NP-completeness
- 6 Backtracking

In this last week we consider the following fundamental question:

- We have seen that some problems have “clever” algorithms, like the general change-making algorithm, which avoid the exponential running time caused by trying all combinations.
- While other problems (which we will see this week) don't seem to have such “clever” algorithms.

What is the reason for this?

And, as important as that — what can be done when actually we seem to need to try all possible combinations?!

The background reading for this week is

Chapter 34 — “NP-Completeness”

from CLRS.

- However we will only cover the intuitive notions.
- That is, for the fundamental notions **P**, **NP** and **NP-completeness** we will consider only the basic ideas.

Though, in one aspect we will go beyond the book:

In the last 10 years a revolution in “SAT solving” took place, making problems “feasible” which were considered “infeasible”.

So we will touch on this aspects.

“Separating Insolvable and Difficult”

In the NYT (July 13, 1999, by George Johnson):

*Anyone trying to cast a play or plan a social event has come face-to-face with what scientists call a **satisfiability problem**. Suppose that a theatrical director feels obligated to cast either his ingénue, Actress Alvarez, or his nephew, Actor Cohen, in a production. But Miss Alvarez won't be in a play with Cohen (her former lover), and she demands that the cast include her new flame, Actor Davenport. The producer, with her own favors to repay, insists that Actor Branislavsky have a part. But Branislavsky won't be in any play with Miss Alvarez or Davenport.*

A systematic solution

The actors become boolean variables (when assigned true they play, when false they don't):

- Actress Alvarez: a
- Actor Cohen: c
- Actor Davenport: d
- Actor Branislavsky: b

The conditions become logical formula:

- 1 $a \vee c$
- 2 $a \rightarrow \neg c$
- 3 d
- 4 b
- 5 $b \rightarrow (\neg a \wedge \neg d)$

Condition 4 implies b , condition 5 then implies $\neg a$, $\neg d$, contradicting condition 3. This problem instance is **unsatisfiable**.

Formulation corrected

If Actress Alvarez won't play, then she will be mad anyway, and so her demands don't need to be satisfied. So Condition 3 should be weakened, and we get:

$$① \quad a \vee c$$

$$② \quad a \rightarrow \neg c$$

$$③ \quad a \rightarrow d$$

$$④ \quad b$$

$$⑤ \quad b \rightarrow (\neg a \wedge \neg d)$$

Again condition 4 implies b , again condition 5 then implies $\neg a$, $\neg d$, now conditions 2, 3 are satisfied (recall “false implies everything”), while condition 1 implies c .

So the unique solution is that Branislavsky and Cohen play, while Alvarez and Davenport don't.

Lesson: Ingenuous young women shouldn't demand too much.

NYT continued (SAT and complexity)

*As more roles need to be filled, the problem becomes harder and harder to solve. Is it possible to satisfy the tangled web of conflicting demands? These satisfiability problems, called **SAT problems** for short, arise in thousands of situations, from staffing companies and scheduling airline flights to planning a wedding dinner that won't devolve into a food fight.*

And, reaching beyond such practical considerations, researchers embrace satisfiability problems as a tool for studying a phenomenon called computational complexity: some problems are

- *inherently easy to solve,*
- *some difficult*
- *and some impossible,*

but scientists are only beginning to understand why.

A new field of applications: verification

After that article has been written, in the last decade, SAT solving developed strong industrial tools:

- Large parts of hardware verification nowadays depend on SAT solving.
- This shouldn't be a surprise: Recall from your hardware lecture, that inside a computer we have (just) boolean logic!

Remark: Unfortunately, this is basically invisible, since in the existing societies the labour process is hidden — programming and (really) applied sciences only show up (sort of) in a James Bond movie, when we have a glimpse at the laboratory of the villain.

NYT continued (brute force)

In computer science, problems can be rated in difficulty by how much time it takes to search for solutions. If there are just a few actors to cast in a play, the answer can be found simply by trial and error.

For five actors, there are only 2 to the 5th possibilities, a mere 32 combinations: A is in, B is out, C is out, D is in, E is out.

The director can try every combination and see if any are consistent with everyone's demands.

The simplest brute-force method

In general, if we have a SAT problems consisting of

- m boolean formulas C_1, \dots, C_m
- with n boolean variables v_1, \dots, v_n ,

we can list all 2^n assignments of true, false to the variables, and check whether they fulfil all conditions C_1, \dots, C_m :

- if we find a solution, the problem (instance) is **satisfiable**,
- otherwise it is **unsatisfiable**.

Even for a large organisation, $n = 60$ would be very difficult, and definitely $n = 80$ is out of reach (within the next 10 years, say).

However, interesting problem instances typically have at least 1000 variables, sometimes even going into the millions, and $2^{1000} \approx 10^{301}$ will forever stay outside the reach of such a method.

Exploiting special structure

In practice, it is usually not necessary to “exhaustively search the problem space,” as mathematicians say. It is quickly evident that there is no way to cast Miss Alvarez, but that the play can go forward with Cohen. For enormous problems involving many variables, pursuing such shortcuts can often save valuable computer time.

Is this just because the problem is so small?

*Viewed more abstractly, the casting example belongs to a class called 2-SAT problems. If parsed into a logical formula (a kind of mathematical syntax called “conjunctive normal form”), they contain so-called clauses like “Alvarez or Cohen,” which each involve no more than two of the potential actors. If one tries to cast a play with 10 or 100 actors, the problem becomes harder. But as long as there are no more than two variables, or actors, in each of the many clauses, the solution time increases relatively slowly, in what is called **polynomial time**. Thus 2-SAT problems are said belong to a class of “solvable problems” called **P**.*

The example re-examined

The five boolean conditions were

$$① \quad a \vee c$$

$$② \quad a \rightarrow \neg c$$

$$③ \quad a \rightarrow d$$

$$④ \quad b$$

$$⑤ \quad b \rightarrow (\neg a \wedge \neg d)$$

Conjunctive normal form (CNF) means that each condition is a disjunction of “literals” (variables or negated variables). Using

$$x \rightarrow y \iff \neg x \vee y$$

indeed we get six disjunctions using each at most two variables:

$$① \quad a \vee c$$

$$② \quad a \rightarrow \neg c \iff \neg a \vee \neg c$$

$$③ \quad a \rightarrow d \iff \neg a \vee d$$

$$④ \quad b$$

$$⑤ \quad b \rightarrow (\neg a \wedge \neg d) \iff \neg b \vee (\neg a \wedge \neg d) \iff (\neg b \vee \neg a) \wedge (\neg b \vee \neg d)$$

3-SAT and NP

*But in more complex situations, with clauses that each have at least three variables (“Alvarez or Branislavsky or Cohen”), the space of possible solutions to explore can explode exponentially as the problem grows larger, with more actors added to the play. In the worst cases, these 3-SAT problems – even ones with only 50 actors – rapidly become unsolvable, even given eons of time. This places them in the domain of difficult problems called **NP-complete**. You can guess at the answer and quickly determine if it’s right or wrong (by coming up with a combination of actors and seeing if it works.) But systematically solving the problem can essentially take forever.*

Remark: Appropriate SAT-solver can actually solve each 3-SAT problem with, say, 700 variables, on a single work station within a month!

*The initials NP stand for the rather opaque term “nondeterministic polynomial.” The important point is that all NP-complete problems are intimately related. Other examples include the famous **traveling salesman problem**, in which you try to find the shortest route connecting many different cities. As the number of destinations increases, the difficulty can rise exponentially and even good approximations are a challenge. If a mathematician could find a general means of solving satisfiability problems this would also dispose of the traveling salesman problem and thousands of others.*

Remark: This has actually become true in a sense — good SAT solvers are available, and with them we can solve many other problems *in practice*.

NYT completed

But mathematicians strongly suspect that unless they have been missing something, there is no general way to solve large NP-complete problems precisely during the lifetime of the human race or even the universe. Instead, researchers look for ways to understand which problems are merely difficult and which are impossibly complex.

Remark: No problem is “impossible complex” —

“In mathematics there is no ignorabimus.”

1930 David Hilbert said:

We must not believe those, who today, with philosophical bearing and deliberative tone, prophesy the fall of culture and accept the ignorabimus. For us there is no ignorabimus, and in my opinion none whatever in natural science. In opposition to the foolish ignorabimus our slogan shall be: We must know — we will know!

P – polynomial time

We need a good notion of “easy problems”.

- Here we speak of **general problems**, like shortest-path problems or change-making problems, not just of **problem instances** (like a concrete change-making problem).
- To make our life easier, we concentrate on **decision problems**, where just “yes” or “no” is to be computed.
- An example is the SAT problem in the form: “Is the input CNF satisfiable or not?”
- Another example is the change-making problem in the form: “Can we precisely return change or not?”

Now **P** stands for all general decision problems which can be solved in polynomial time, that is, there exists an algorithm A which decides correctly all inputs, and there is some fixed k such that the running time of A is $O(n^k)$, where n is the input size.

NP – verification in polynomial time

NP is the class of decision problems where a solution is “easy” to verify:

- For the casting problem (in the general form), a potential solution is the list of actors which play and which don't play, and the verification consists in checking whether all conditions are fulfilled for this assignment.
- In general, for a SAT problem a potential solution is an assignment of boolean truth values to the variables, and the verification process consists in checking whether under this assignments all boolean condition (clauses) become true.

The precise definition of NP is, that this is the class of all decision problems where for the “yes”-instances certificates exist which can be verified in polynomial time (this implies that these certificates must be “short”, i.e., their length is polynomially bounded).

NP in other words

For a problem in NP, like SAT, we not only have a decision problem, but there is a notion of “solution” underlying the decision, where “yes” means there is a solution.

It may be hard to *find* a solution,
but a possible solution is relatively small,
and once you have guessed one
it is easy to *verify* whether it is actually a solution.

This is the basis for the trivial algorithms (by “simplest brute force”): We run through all possible solutions, check in polynomial time whether it is a solution or not, and if we don’t find a solution then the input is a “no”-instance.

So the class NP is defined as the class of all (general) problems, such that we have an efficient notion of a “witness” for the fact that a concrete problem is solvable. Thus a witness (which is not too big) exists for a concrete problem iff it is solvable.

“In P” and “in NP”

We know, that the graph-connectedness problem, deciding whether a graph is connected, is in P. (Recall that we can use depth-first or breadth-first search for that.)

- Now we can also say that graph-connectedness is in NP.
- Since $P \subseteq NP$, this is correct.
- Saying that some problem is in NP establishes an **upper bound** on the algorithmic complexity — but it might still be just in P.

Though if asked about the most appropriate complexity class for a problem, then obviously we want to give an answer as sharp as possible.

The $P=NP$ -problem

We have $P \subseteq NP$, since if a problem is polytime-decidable, then it is also polytime-verifiable (just take the polytime-decision as the verification).

On the other hand, if an arbitrary problem is polytime-verifiable, then we should not be able to show (in general!), that it is also polytime-decidable:

Just verifying a given solution should in general be easier
than to actually providing a solution.

So the conjecture is

$$P \subset NP.$$

This is the famous **P vs. NP Problem**, the central open problem of computer science.

Example: graph colouring

We have already seen that SAT is in NP. Another problem is the **graph k -colouring problem**:

Given a graph G and a natural number k ,
decide whether G is k -colourable.

Here a k -colouring of a graph is map $f : V(G) \rightarrow \{1, \dots, k\}$,
such that adjacent vertices obtain different colours.

Obviously graph k -colourability (for fixed k) is in NP. Similarly
to SAT we have

- graph 2-colourability is polytime-decidable,
- while graph 3-colourability is “NP-complete” (and thus likely not polytime-decidable).

Remarks on graph colouring

- A graph G is **bipartite** if(f) it is 2-colourable, which means that we can bipartition the vertex set into two (disjoint) sets A and B such that no edge in G is inside A or inside B (but must connect A and B).
- As an example, decide whether the grid graphs with m lines and n columns are bipartite:
 - 1 The vertices are the pairs (a, b) of natural numbers with $1 \leq a \leq m$ and $1 \leq b \leq n$.
 - 2 The edges are between vertices which coincide in one component and differ in the other component exactly by 1.
- The simplest example of a graph which is not bipartite, but 3-colourable, is the triangle K_3 .
- And in general the complete graph K_n with n vertices is n -colourable, but not $(n - 1)$ -colourable. (K_n has vertex set $\{1, \dots, n\}$ and all possible $\frac{1}{2}n(n - 1)$ edges.)

Chess as a decision problem

Let's try to determine the “natural” complexity class of chess.

- The first problem is that we need to frame it as a decision problem.
- So let's consider the problem whether a given chess position is a winning position (or not) for the player (white or black) who is next to move.

Funnily now, chess is not just in P, but solvable in **constant time!**

- This is because chess is a finite game, since there are only finitely many positions, and repeating a position is senseless.
- And the big-Oh notation absorbs everything doable in finite time (complexity theory is about **asymptotic complexity**).

So we need to invent “generalised chess”:

- 1 The input board can now be an arbitrary $n \times n$ board.
- 2 We use the same figures and rules for their moves, however we allow as many of them to appear in a position as the board allows.
- 3 We might care about the proper generalisation of special rules, but this is not essential.

Now chess (in this generalised form) should be “really hard”.

Really hard

How many positions are there?

- 1 We have n^2 fields.
- 2 White and black each have 6 different types of figures.
- 3 So there are $(6 \cdot 2 + 1)^{n^2} = 13^{n^2}$ many positions (that is, exponentially many).
- 4 If we also include whether white or black is to move next, then we get $2 \cdot 13^{n^2}$ many positions.
- 5 The special rules introduce a slight dependency on the history of the game, but this is not essentially.

Making the reasonable requirement that no position is repeated, the maximal length of a game is thus $2 \cdot 13^{n^2}$. Now how many moves are there per position?

- 1 Each figure has at most $4n$ many fields to move to.
- 2 So we have at most $n^2 \cdot 4n = 4n^3$ many moves.

Thus we can decide (generalised) chess in time
 $O((4n^3)^{2 \cdot 13^{n^2}}) = O((8n^6)^{13^{n^2}})$ (doubly exponential!).

Really really hard?

Recall, what we have just shown is just an *upper bound* — perhaps we can do (much) better?

- If everything “collapses”, that is, we would have $P = NP$, then (roughly!) “automatically” one level of exponentiation would collapse.
- However then (generalised) chess would still be single exponential.
- And since it is a natural problem, one can actually **prove** that (generalised) chess is not in NP.

Bounding the number of moves

Perhaps we shouldn't ask for "just wins", but for winning in k moves, where k is part of the input (in unary notation!).

- This is like "chess mate in 2".
- Now we can decide the problem in time $O((4n^3)^k) = O(4^k n^{3k})$ (which is singly exponential).
- Moreover, this running through all possibilities (in the game tree) can be done in *space* $O(k \cdot 4n^3)$ by the *backtracking algorithm* (one only needs to store the positions along the currently investigated path).
- So the problem can be solved in **polynomial space**, which as complexity class is denoted by PSPACE.

Since (generalised) chess is a "natural" problem, we can show that "winning in k moves" is in NP

if and only if $\text{NP} = \text{PSPACE}$ holds.

What we cannot prove (but strongly believe)

By the backtracking algorithm we can show $NP \subseteq PSPACE$. So we have

$$P \subseteq NP \subseteq PSPACE.$$

According to current knowledge, we can not exclude $P = PSPACE$, which would mean we would live in a completely different world (similarly to the possibility of creating whole new universes):

- 1 Although we can do a lot of interesting things about SAT, it is strongly believed that $SAT \notin P$.
- 2 And problem like “white wins in k moves” are strongly believe to be not even in NP (the alternation of white and black makes it much harder to prune the search tree).

So it is strongly believed that

$$P \subset NP \subset PSPACE.$$

SAT is not just “any member of NP”:

- SAT in fact represents the whole class.
- That is, any other problem in NP can be expressed as a special case of the SAT problem.

Now how to make this “universality of SAT” precise?

- We need to make precise what it means to express some other problem in NP as a special case of SAT.
- This will be expressed by saying that the other problem is “reduced to SAT”.

The notion of a “reduction”

Consider some other (general) problem $\mathcal{P} \in \text{NP}$.

- 1 Instead of designing a special algorithm for solving \mathcal{P} (that is, designing a decision algorithm, outputting correctly “yes” or “no” according to the nature of \mathcal{P}), we just want to design a “cheap” translation of \mathcal{P} into SAT.
- 2 That is, we want to have some function t , which maps instances I of \mathcal{P} into instances $t(I)$ of SAT, such that $t(I)$ is satisfiable iff I is a “yes”-instance of \mathcal{P} .
- 3 In this way we can use some “off-the-shelves” SAT solver, which hopefully is much more efficient than a hand-crafted algorithm for \mathcal{P} .
- 4 In order for this to be efficient, we demand t to be efficiently computable, i.e., computable in polynomial time.

The definition of NP-completeness

We say that a problem $\mathcal{P}_1 \in \text{NP}$ is **reducible** to $\mathcal{P}_2 \in \text{NP}$ if there exists a polytime-function t such that the answer for the decision problem for I w.r.t. \mathcal{P}_1 is the same as the answer for the decision problem for $t(I)$ w.r.t. \mathcal{P}_2 .

Now a problem $\mathcal{P} \in \text{NP}$ is called **NP-complete** if every problem $\mathcal{P}' \in \text{NP}$ is reducible to \mathcal{P} .

So NP-completeness for a decision problem \mathcal{P} means precisely two things:

- 1 We have the upper bound $\mathcal{P} \in \text{NP}$.
- 2 We have “NP-hardness” (some sort of “lower bound”), that is, every (other) problem $\mathcal{P}' \in \text{NP}$ is reducible to \mathcal{P} .

Solving one NP-complete problem is enough

If we have one NP-complete problem \mathcal{P} , then in principle we can solve any other problem in NP as well.

- That is, modulo a polytime reduction.
- From our broad theoretical perspective, this polytime computation doesn't matter.
- In practice however it matters, and really interesting application of \mathcal{P} can only to be expected for “neighbourhood problems” in NP.

Back to the theoretical perspective, we have obviously:

Lemma *Given some NP-complete problem \mathcal{P} , we have $P = NP$ if and only if this special problem \mathcal{P} can be decided in polynomial time.*

False statements I

In CLRS in the introduction to Chapter 34 we find

If you can establish a problem as NP-complete, you provide good evidence for its intractability. As an engineer, you would then do better to spend your time developing an approximation algorithm or solving a tractable special case, rather than searching for a fast algorithm that solves the problem exactly.

We have a typical example of ideological use of language:

- 1 First one makes the purely technical definition “intractable means NP-complete”, where one could also have used the notions “interesting”, “lovely”, or “ghrsuhf”.
- 2 Then you take the intuitive meaning of the word, and claim this as a “mathematical fact”.

Furthermore, for example for SAT, approximation algorithms are completely irrelevant, and also “tractable special cases” (like 2-SAT) are of no practical relevance, while the exact solution of (arbitrary) SAT problems is of high practical relevance.

The NP-completeness of SAT

Stephen Cook introduced in 1971 the notion of NP-completeness, and proved the NP-completeness of SAT.

SAT is the central NP-complete problem. How to prove it?

- 1 We have already argued that SAT is in NP.
- 2 Now consider some arbitrary problem $\mathcal{P} \in \text{NP}$.
- 3 We have to find a polytime reduction t of \mathcal{P} to SAT.
- 4 We know that there is a polytime algorithm A such that x is a “yes”-instance of \mathcal{P} iff there exists a short witness w with $A(x, w)$ outputting “yes”.
- 5 There is a computer made of boolean circuits computing A .
- 6 This computer can be translated into a boolean formula $F(x, w)$, true iff w witnesses that x is a “yes”-instance.
- 7 So $t(I)$ (where I is a 0, 1-string) substitutes this concrete I for x into the general formula $F(x, w)$, and we obtain $t(I)$ which is satisfiable iff x is a “yes”-instance of \mathcal{P} .

False statements II

In CLRS we further find:

When we demonstrate that a problem is NP-complete, we are making a statement about how hard it is (or at least how hard we think it is), rather than about how easy it is. We are not trying to prove the existence of an efficient algorithm, but instead that no efficient algorithm is likely to exist. In this way, NP-completeness proofs bear some similarity to the proof of an $\Omega(n \lg n)$ time lower bound for any comparison sort algorithm; the specific techniques used for showing NP-completeness differ from the decision-tree method, however.

This is quite strange:

- A proof of NP-completeness doesn't show something is hard, but provides efficient algorithms for verification and reduction.
- The sorting lower-bound proof is of a different nature.

Translating colouring problems into SAT

Consider the cycle $C^5 = 1 \text{ --- } 2 \text{ --- } 3 \text{ --- } 4 \text{ --- } 5$.

We want to determine whether C^5 is 2-colourable (or, in other words, whether C^5 is bipartite). Let us formulate the 2-colouring problem for C^5 as a SAT problem:

Variables are $p_{i,j}$ for $i \in \{1, \dots, 5\}$ and $j \in \{1, 2\}$, where $p_{i,j}$ expresses, that vertex i gets colour j .

First the clauses to ensure that every vertex gets a colour:

$$C_1 := p_{1,1} \vee p_{1,2}$$

$$C_2 := p_{2,1} \vee p_{2,2}$$

$$C_3 := p_{3,1} \vee p_{3,2}$$

$$C_4 := p_{4,1} \vee p_{4,2}$$

$$C_5 := p_{5,1} \vee p_{5,2}$$

Next come the clause-sets expressing that adjacent vertices get different colours:

$$F_{1,2} := (\neg p_{1,1} \vee \neg p_{2,1}) \wedge (\neg p_{1,2} \vee \neg p_{2,2})$$

$$F_{2,3} := (\neg p_{2,1} \vee \neg p_{3,1}) \wedge (\neg p_{2,2} \vee \neg p_{3,2})$$

$$F_{3,4} := (\neg p_{3,1} \vee \neg p_{4,1}) \wedge (\neg p_{3,2} \vee \neg p_{4,2})$$

$$F_{4,5} := (\neg p_{4,1} \vee \neg p_{5,1}) \wedge (\neg p_{4,2} \vee \neg p_{5,2})$$

$$F_{5,1} := (\neg p_{5,1} \vee \neg p_{1,1}) \wedge (\neg p_{5,2} \vee \neg p_{1,2}).$$

Finally we obtain $F_2(C^5)$ which is satisfiable iff C^5 is 2-colourable:

$$F_2(C^5) = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge$$

$$F_{1,2} \wedge F_{2,3} \wedge F_{3,4} \wedge F_{4,5} \wedge F_{5,1}.$$

$F_2(C^5)$ has $5 \cdot 2 = 10$ variables and $5 + 5 \cdot 2 = 15$ clauses.

A 3-SAT instance

Let's consider a concrete SAT problem:

$$F = \{ \{a, b, c\}, \{c, d, e\}, \{a, e, f\}, \\ \{a, d, g\}, \{b, e, g\}, \{c, f, g\}, \{b, d, f\}, \\ \{\bar{a}, \bar{b}, \bar{c}\}, \{\bar{c}, \bar{d}, \bar{e}\}, \{\bar{a}, \bar{e}, \bar{f}\}, \\ \{\bar{a}, \bar{d}, \bar{g}\}, \{\bar{b}, \bar{e}, \bar{g}\}, \{\bar{c}, \bar{f}, \bar{g}\}, \{\bar{b}, \bar{d}, \bar{f}\} \}.$$

We are using here **clause-sets**, which are a more mathematical presentation of CNFs: We are just using sets of sets, where the outer composition is understood as conjunction, and the inner composition as disjunction.

We have seven variables (namely a, b, c, d, e, f, g), and thus by the simplest brute-force approach we would need to consider $2^7 = 128$ assignments in order to find out whether F is satisfiable or not.

The basic idea of backtracking

- We need something more clever.
- The basic idea is to make case distinctions on variables, and to exploit simplifications enabled by these case distinctions.

For the first case distinction, we consider $\langle a \rightarrow 0 \rangle$ and $\langle a \rightarrow 1 \rangle$.

Here $\langle a \rightarrow 0 \rangle$ and $\langle a \rightarrow 1 \rangle$ denote the **partial assignments** which (just) set the variable a to 0 resp. 1 (where 0 means false and 1 means true).

This backtracking approach in fact works for every problem in NP. However, especially with SAT there are many techniques to gain dramatic improvements.

The first branch

Let's first investigate $\langle a \rightarrow 0 \rangle$.

$$F_{a=0} = \{ \{b, c\}, \{c, d, e\}, \{e, f\}, \\ \{d, g\}, \{b, e, g\}, \{c, f, g\}, \{b, d, f\}, \\ \{\bar{c}, \bar{d}, \bar{e}\}, \{\bar{b}, \bar{e}, \bar{g}\}, \{\bar{c}, \bar{f}, \bar{g}\}, \{\bar{b}, \bar{d}, \bar{f}\} \}.$$

Now let's split on b , that is, we consider the cases $\langle b \rightarrow 0 \rangle$ and $\langle b \rightarrow 1 \rangle$.

$$F_{a=0, b=0} = \{ \{c\}, \{c, d, e\}, \{e, f\}, \\ \{d, g\}, \{e, g\}, \{c, f, g\}, \{d, f\}, \\ \{\bar{c}, \bar{d}, \bar{e}\}, \{\bar{c}, \bar{f}, \bar{g}\} \}.$$

Here something nice happened, namely $\{c\}$ appears now in a **unit clause**: We conclude that variable c **must** be set to 1:

$$F_{a=0, b=0, c=1} = \{ \{e, f\}, \{d, g\}, \{e, g\}, \{d, f\}, \\ \{\bar{d}, \bar{e}\}, \{\bar{f}, \bar{g}\} \}.$$

First backtracking

Now let's split on e :

$$F_{a=0,b=0,c=1,e=0} = \{ \{f\}, \{d, g\}, \{g\}, \{d, f\}, \{\bar{f}, \bar{g}\} \}.$$

Here we get two unit clauses, and we see, that $F_{a=0,b=0,c=0,e=0}$ is unsatisfiable. So we have to **backtrack**. The last case distinction was on e , so let's consider the other case:

$$F_{a=0,b=0,c=1,e=1} = \{ \{d, g\}, \{d, f\}, \{\bar{d}\}, \{\bar{f}, \bar{g}\} \}.$$

Here we have to set $d = 0$:

$$F_{a=0,b=0,c=1,e=1,d=0} = \{ \{g\}, \{f\}, \{\bar{f}, \bar{g}\} \},$$

and again we found an unsatisfiable case. So we must backtrack to the next-higher level, that is, we have to consider $b = 1$:

$$F_{a=0,b=1} = \{ \{c, d, e\}, \{e, f\}, \{d, g\}, \{c, f, g\}, \\ \{\bar{c}, \bar{d}, \bar{e}\}, \{\bar{e}, \bar{g}\}, \{\bar{c}, \bar{f}, \bar{g}\}, \{\bar{d}, \bar{f}\} \}.$$

Let's split on e again (since it occurs in a binary clause(!)):

$$F_{a=0,b=1,e=0} = \{ \{c, d\}, \{f\}, \{d, g\}, \{c, f, g\}, \\ \{\bar{c}, \bar{f}, \bar{g}\}, \{\bar{d}, \bar{f}\} \}.$$

By “unit-clause elimination” we must set $f = 1$:

$$F_{a=0,b=1,e=0,f=1} = \{ \{c, d\}, \{d, g\}, \{\bar{c}, \bar{g}\}, \{\bar{d}\} \}.$$

Again, we must set $d = 0$:

$$F_{a=0,b=1,e=0,f=1,d=0} = \{ \{c\}, \{g\}, \{\bar{c}, \bar{g}\} \}.$$

We see that this case is unsatisfiable. We backtrack to $e = 1$:

$$F_{a=0,b=1,e=1} = \{ \{d, g\}, \{c, f, g\}, \{\bar{c}, \bar{d}\}, \{\bar{g}\}, \{\bar{c}, \bar{f}, \bar{g}\}, \{\bar{d}, \bar{f}\} \}.$$

We must set $g = 0$:

$$F_{a=0,b=1,e=1,g=0} = \{ \{d\}, \{c, f\}, \{\bar{c}, \bar{d}\}, \{\bar{d}, \bar{f}\} \}.$$

Now we must set $d = 1$

$$F_{a=0,b=1,e=1,g=0,d=1} = \{ \{c, f\}, \{\bar{c}\}, \{\bar{f}\} \},$$

and the result is unsatisfiable.

Completion

Thus we must backtrack to the highest level, namely we must consider

$$F_{a=1} = \{ \{c, d, e\}, \{b, e, g\}, \{c, f, g\}, \{b, d, f\}, \\ \{\bar{b}, \bar{c}\}, \{\bar{c}, \bar{d}, \bar{e}\}, \{\bar{e}, \bar{f}\}, \\ \{\bar{d}, \bar{g}\}, \{\bar{b}, \bar{e}, \bar{g}\}, \{\bar{c}, \bar{f}, \bar{g}\}, \{\bar{b}, \bar{d}, \bar{f}\} \}.$$

I leave it to you to complete the backtracking process, and to show that also $F_{a=1}$ is unsatisfiable, whence F is unsatisfiable (you might also think about possibilities to shorten this process).

The whole process is still quite laborious, but better than to consider 128 assignments. The savings we gained arose from

- the unit clause eliminations
- and from finding a contradiction already when not all variables have been assigned a value.

Conclusion

- P is the class of problems which are relatively “easy” to solve. This means basically scalability — doubling the size of the problem will only increase the running time by a constant factor.
- NP is the (larger) class of (decision) problems where for yes-answers there exist short certificates which are also easy to verify.
- Each problem in NP is solvable by backtracking algorithms, using exponential time (but only polynomial space).
- SAT is an important NP-complete problem, and its NP-completeness can be utilised by translating other problems into SAT (“reducing” them to SAT), and using SAT solvers.
- That a problem is NP-complete doesn’t mean that it’s “intractable”, but that we have to use stronger methods like backtracking.