15-453 Lecture 34: More About Satisfiability

Mihai Budiu

April 21

This lecture is about the satisfiability problem: we discuss its theoretical and practical importance from several points of view.

1 Review

1.1 The classes P and NP

The class of languages which can be decided in polynomial time is denoted by P. The class of languages for which there is a polynomial-time *verifier* is called NP.

We have seen this **theorem**: NP = the class of languages which can be decided in polynomial time by a non-deterministic Turing Machine.

A polynomial-time reduction between two languages A and B is a function f which can be computed in polynomial time, which maps A to B in the following sense: $x \in A \Leftrightarrow f(x) \in B$. We write $A \leq_p B$ if such a function exists.

Definition: a problem is called NP-complete if:

- it is in NP;
- every other problem in NP reduces to it.

The NP-complete problems are the *hardest* problems in NP.

1.2 Satisfiability

A boolean formula is an expression built according to the following grammar:

$$
\begin{aligned}
\phi =& x && \text{(variables)} \\
& |(\phi_1 \vee \phi_2) && \text{(logical or)} \\
& |(\phi_1 \wedge \phi_2) && \text{(logical and)} \\
& |\neg\phi && \text{(logical negation)}
\end{aligned}
$$

The boolean variables can have one of the values {false, true}, sometimes denoted with 0 and 1.

Assigning a value to the variables gives a value to the formula.

A formula is called *satisfiable* if there exists an assignment to the variables which makes the formula true. The set of all satisfiable boolean formulas is called SAT.

In the last lecture we have seen the famous **Cook–Levin Theorem:** (proved independently by the two in 1971): "SAT is NP-complete".

Each boolean formula can be written in conjunctive normal form (CNF); then the formula is a conjunction of clauses, each clause is a disjunction of literals, and each literal is a variable or a variable negated. The form k-CNF has in each clause exactly k literals. Every formula can be written in k-CNF, for any k.

2 The Economic Importance of NP

The existence of NP-complete problems was known prior to the Cook–Levin theorem. The importance of this theorem was to show that *natural problems*, like SAT, are NP-complete.

After this theorem was proven, researchers started proving chains of completeness results, by reducing SAT or other NP-complete problems to new problems. In 1979 a whole book was published, containing hundreds of NP-complete problems: Garey M. and Johnson D: "Computers and Intractability — A Guide to the Theory of NP-completeness", W. H. Freeman, 1979.

Since 1971 more than a thousand of other important problems have been shown to be NP-complete. There are compendiums of such problems on the Internet.

It is very important to understand the significance of NP-completeness. NP-completeness has as important consequences as undecidability: not knowing that a problem is undecidable can make you waste time trying to find a solution. Not knowing that a problem is NP-complete can make you waste time trying to find an efficient solution.

2.1 Computing functions

Definition: a problem is called NP-hard if any problem in NP reduces $\leq_p$ to it.

An NP-hard problem is at least as hard as any NP-complete problem.

Definition: A function $f : \Sigma^* \to \Sigma^*$ is computable in time $t(n)$ if there is a Turing Machine which on input w halts in time $t(|w|)$ and on the tape it has written $f(w)$.

We can extend the notion of reduction to deal with computable functions (we will not do here so). Then we can extend the notion of hardness to computable functions too.

For example, the problem TSP-k: "$\langle G, k\rangle$, such that graph G has a Traveling Salesman tour of length less than k" is in NP, and we will prove later that it is NP-complete.

But the problem MIN-TSP: "Which is the minimum length of a Traveling Salesman tour for a graph G" is an optimization problem, and not a decision problem (the answer is a number, and not just "yes/no"). We will show that this problem is NP-hard in the following way: if we can solve MIN-TSP, we can solve TSP-k too.

Proof: assume we can solve MIN-TSP. There is a TM which given a graph G computes in polynomial time the minimum-length tour (or infinity if there is no tour). We are given an instance $\langle G, k\rangle$, and we want to see if this is an instance of TSP-k. Then we run MIN-TSP on G, we get an answer r. If $r \leq k$ we accept, else we reject.

2.2 Important NP-complete problems

Traveling-salesman: find the shortest circuit which visits all cities on a map.

Planning: plan the execution of some tasks on a number of machines in a factory to minimize completion time.

Scheduling: plan the execution of some dependent jobs on a multi-computer to minimize completion time.

Knapsack: having a set of valuable and heavy objects to carry, find what to put in a fixed-capacity backpack to maximize value and not break it (exceed the maximum weight).

Routing: having a circuit board and some devices, find the shortest way to make some indicated connections.

Register allocation: how should a compiler assign registers to variables to minimize the running time.

Integer programming: does a system of linear equations with integer coefficients have integer solutions?

There are effectively billions of dollars hanging on good solutions to such problems.

Note: the title of this section is an abuse: these problems are **not** in NP (so they cannot be called "NP-complete"), but they aren't more difficult than NP-complete problems. We can reduce each of them to a polynomial number of NP-complete problems. You will make two such proofs in homework 10.

2.3 Cryptography and NP

In case you still aren't convinced that the class NP is important, ponder this: for any encryption algorithm, decryption has to be in NP! The decryption program can be seen as a verifier, the certificate being the encryption key.

So if P=NP, there is no cryptography! (Assuming all of P is easy.) I.e. we can break any encryption algorithm in polynomial time. A proof that P=NP would make CIA, KGB and a lot of other organizations very very unhappy!

Also, if P=NP, there are no pseudo-random number generators! (A very interesting theorem shows that there is cryptography if and only if there exist pseudo-random number generators.)

3 Solution Density and Problem Hardness

NP-complete problems show up a lot in practice. Should we just give up solving them?

An unrelated question: NP-complete problems are "hard". Are they suitable for building unbreakable encryption schemes?

The answer to both these questions is essentially **no**!

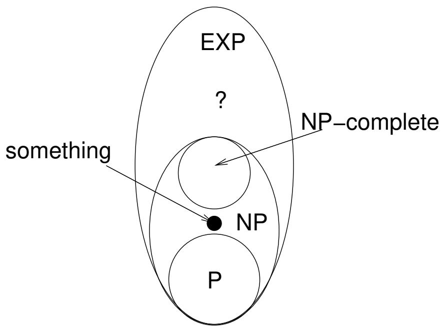
NP-complete problems are hard in the *worst-case* sense. Remember: we are talking about worst-case complexity. If out of all the inputs of size n only one requires a lot of time, and all the rest can be solved quickly, we say the time for size n is the biggest of all these.

But if most problems of size n are easy, we have hope to solve them. For example, there are a lot of Traveling-Salesman tours where we can immediately say there is no tour: when the graph is disconnected.

Cryptography problems must be hard in the average-case sense: if we generate a random key, it should be hard to break. Many NP-complete problems tend to be easy on the average-case (this is an imprecise statement). Cryptography today is using problems not known to be NP-complete, but not known to be in P either. For instance very often the *factoring* problem is used: find the factors of an integer number. This problem is *not* known to be NP-complete. It is not known to be in P either.

If you think factoring is easy, because you have learned in school how to factor numbers, check that algorithm again: its complexity is $O(2^n/n)$. This is totally infeasible for numbers with more than 20 digits, even using the fastest super-computers.

One can prove by diagonalization an interesting theorem: if P $\neq$ NP, then there is a problem which is *not* NP-complete but which is not in P. So, if P$\neq$NP, the world picture looks like this:



It may be that factoring is such a "something".

3.1 SAT Solution Density

How hard is an average SAT problem? A group of researchers from mixed backgrounds have tried to address this question empirically. Their article drew a lot of attention (it was featured in Science and New York Times, see
`http://www.nytimes.com/library/national/science/071399sci-satisfiability-problems.html`).

They study the difficulty of a problem with respect to how *constrained* the problem is. Let us look at k-CNFSAT. For a given formula in k-CNF form we define α, the constraint, as the ratio between the number of clauses and the total number of variables.

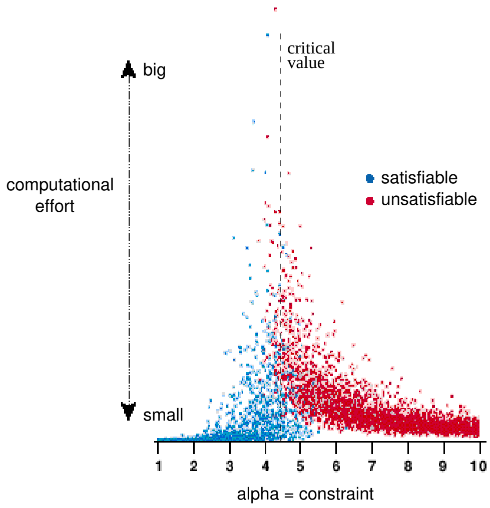
What is the intuition for α? Let us consider a clause which contains the literal x. For this clause to be true, we need at least one literal to be true. If we pick x to be true, then $\neg x$ will be false. So all the clauses which contain $\neg x$ now have one less possibility of becoming "true". They are "constrained" by the first clause.

If we assume that variables are uniformly distributed, then the number of constraints grows exactly like the number of clauses: the more clauses you have, the more constraints among themselves. Because *all* the clauses have to evaluate to true for the formula to be satisfiable, it follows that the more clauses we have, the harder it will be to satisfy the formula.

This is just an intuitive picture: we would expect unconstrained formulas to have a lot of solutions, and very constrained formulas to have few or no solution. We also expect very unconstrained and very constrained formulas to be easy to solve:

- For the very unconstrained ones, almost every assignment should be a solution;
- For the very constrained, we should be able to exhibit quickly a contradiction (a chain of constraints which leaves no solution).

How can we test this hypothesis? The cited paper tackles this in a very unconventional way (for theoretical computer science): experimentally. The researchers generate a lot of *random* 3-CNFSAT problems of rather modest size, which then they try to solve by brute force (e.g. try all possibilities).



This figure (stolen from the web page of Bart Selman from Cornell) shows the qualitative results. Around a critical value (which depends on k) the problems are much harder. For such instances the known algorithms spend a long time.

The original paper in Science made parallels between this phenomenon and the solidification of liquids: in a liquid there are few constraints between the molecules, which can move freely. Just when the transition to solid state is made, the degrees of freedom of the molecules decrease dramatically. That is why the original article calls the transition phenomenon for SAT a "phase transition".

4 Solving NP-complete problems

So some instances of NP-complete problems are easy, and some are hard. But in real-life we are often faced with NP-complete problems, which we need to solve. How do we proceed?

A part of Artificial Intelligence tries to deal with this problem, through the study of *heuristics*: these are algorithms which have still an huge worst-case bound, but attempt to solve efficiently most of the problem instances.

Note: even if P $\neq$ NP, it is interesting to know how far apart are NP and EXP. To date the best deterministic algorithms for SAT run in exponential time!

4.1 The Davis-Putnam Algorithm

The most-effective algorithm remains basically one proposed in 1960 by M. Davis and H. Putnam. Some embellishments have been proposed during time, but the substance is the same:

1. Choose a literal to satisfy.
2. Delete all clauses containing the literal (they are satisfied now).
3. Delete the negated literal from all other clauses.
4. Continue until no clauses are left or until formula is not satisfiable (i.e. you obtain a clause without any literal).

The crucial part is how to choose the literal in step 1:

- If a clause contains only one literal, it is the one chosen.
- Otherwise a literal is chosen arbitrarily (here there are some variations and embellishments) and assigned successively 0 and 1.
- If the formula fails to be satisfied in either case, backtrack.

4.2 The Walk-SAT Randomized Algorithm

This algorithm is "incomplete": it can only find satisfying assignments if they exist, but cannot prove that a formula is unsatisfiable. It is especially effective when the formula is underconstrained. The algorithm was proposed by Bart Selman. It essentially does a "random walk on a hypercube". Here it is:

1. Assign randomly each variable.
2. If the formula is satisfied, we are done.
3. Choose one unsatisfied clause randomly and uniformly.
4. Flip a coin:
   (a) If "head", toggle the value of a variable selected at random in the clause.
   (b) If "tails", choose the variable whose toggling would maximize the number of other satisfied clauses.
5. Unless a threshold for the number of steps has been exceeded, goto 2.

4.3 The Fastest Solvers

Note: The fastest way to solve SAT to date uses reconfigurable hardware (my own research area!). A SAT formula is compiled into a circuit and the variables into small finite-state machines. Next the hardware is let run (a variation of the Davis-Putnam algorithm).

4.4 Other Approaches

Sometimes when we have optimization problems we can settle for approximate solutions. Some things can be easily approximated, some other remain NP-hard. There is a whole area of research on approximation algorithms.

On the other hand, very often randomization helps. There are very hard problems for which we only have randomized algorithms. Randomized computation is a whole new model of computation, which has surprisingly interesting results. For instance, all known effective algorithms to generate prime numbers are Monte-Carlo; prime numbers are the quintesential ingredient in most modern cryptographic algorithms.

5 Conclusions

There are a few lessons worth remembering:

- SAT is a very important problem not only from a theoretical p.o.v, but also from a practical one.
- Many real-life optimization problems are NP-hard. Having a quick solution for SAT would solve them all.
- NP-complete problems are worst-case hard.
- If NP=P, there is no cryptography.
- Cryptography uses problems which are not known to be NP-complete; they must be hard on average.
- We know very simple algorithms which can solve easy instances of SAT quickly.