NP-Completeness and Cook's Theorem

Lecture notes for COM3412 *Logic and Computation*

7th March 2008

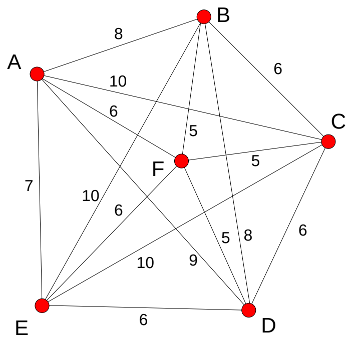
1 NP decision problems

The **decision problem** $D_{\mathcal{L}}$ for a formal language $\mathcal{L} \subseteq \Sigma^*$ is the computational task:

> Given an arbitrary string $I \in \Sigma^*$, to determine whether or not $I \in \mathcal{L}$.

The input string I is called an **instance** of the problem $D_{\mathcal{L}}$. It is a **positive** or "yes" instance if $I \in \mathcal{L}$, otherwise it is a **negative** or "no" instance. Any computational decision problem can be represented as the decision problem for some formal language $\mathcal{L}$.

Example The graph below gives an instance of the Travelling Salesman Problem:



The *optimisation problem* is 'What is the shortest round tour of the nodes in the graph?'. For a decision problem we need to specify a distance, e.g., 'Is there a round tour of length at most 37?'. This decision problem instance can be encoded by the string

$$a, b, 8; a, c, 10; a, d, 9; a, e, 7; a, f, 6; b, c, 6; b, d, 8; b, e, 10; b, f, 5; c, d, 6; c, e, 10; c, f, 5; d, e, 6; d, f, 5; e, f, 6; 37.$$

A decision problem is called **non-deterministically polynomial** (NP) if there is a language C and a relation $cert \in C \times \Sigma^*$ such that, for each $I \in \Sigma^*$,

1. $I \in L$ if and only if $cert(c, I)$ for some $c \in C$, and

2. for any $c \in C$, it can be determined whether or not $cert(c, I)$ in a time that is bounded by a polynomial function of the length of I.

For a positive instance I of $D_{\mathcal{L}}$, any element $c \in C$ such that $cert(c, I)$ is called a **certificate** for I. It certifies that I is indeed a positive instance (since only positive instances *have* certificates), and it can be checked to be a certificate in polynomial time.

For "it can be determined" in the definition above, we may read "there exists a Turing machine which can determine"; this is justified by the Church-Turing thesis. Later we shall be more specific about the format we would like our certificate-checking machine to have.

If we had access to unlimited parallelism, we could build a machine which, given instance $I \in \Sigma^*$, concurrently checks all $c \in C$ to determine whether $cert(c, I)$. This could be done in polynomial time. Alternatively, we could use a non-deterministic machine which *guesses* an element $c \in C$ and checks whether or not it is a certificate for I. This also can be done in polynomial time, but is not of any practical utility, since the chance of finding a certificate for I at random is remote; but it does explain the origin of the designation NP.

Very many computational problems encountered in practice are NP. You learnt about some of them in the second year.

Stephen Cook showed in 1971 that any NP problem can be reduced in polynomial time to the specific problem SAT, the Satisfiability Problem for Propositional Calculus clauses. We shall define this problem, explain Cook's proof, and then discuss its implications: in particular, it leads us to the very important notion of *NP-completeness*.

2 The satisfiability problem SAT

A **clause** in the Propositional Calculus is a formula of the form

$$\neg A_1 \vee \neg A_2 \vee \cdots \vee \neg A_m \vee B_1 \vee B_2 \vee \cdots \vee B_n,$$

where $m \geq 0, n \geq 0$, and each A_i and B_j is a single schematic letter. Schematic letters and their negations are collectively known as **literals**; so a clause is simply a *disjunction of literals*.

The clause above was written in **disjunctive form**. It can also be written in **conditional form**, without negation signs, as

$$A_1 \wedge A_2 \wedge \ldots A_m \rightarrow B_1 \vee B_2 \vee \cdots \vee B_n.$$

Note that if $m = 0$ this is simply

$$B_1 \vee B_2 \vee \cdots \vee B_n,$$

while if $n = 0$ it may be written as

$$\neg(A_1 \wedge A_2 \wedge \ldots A_m).$$

We shall use the conditional notation for clauses from now on.[1]

Note that *any* Propositional Calculus formula is equivalent to the conjunction of one or more clauses, for example:

$$\begin{aligned} A \leftrightarrow B &\cong (A \rightarrow B) \wedge (B \rightarrow A) \\ A \vee (B \wedge C) &\cong (A \vee B) \wedge (A \vee C) \\ A \wedge B \rightarrow \neg C &\cong \neg(A \wedge B \wedge C) \end{aligned}$$

The **satisfiability problem** (SAT) may be stated as follows:

> Given a finite set $\{C_1, C_2, \ldots, C_n\}$ of clauses, determine whether there is an assignment of truth-values to the schematic letters appearing in the clauses which makes all the clauses true.

The size of an instance of SAT is the total number of schematic letters appearing in all the clauses; e.g., the instance

$$\{A \vee B, A \rightarrow C \vee D, \neg(B \wedge C \wedge E), C \rightarrow B \vee E, \neg(A \wedge E)\}$$

is of size 13.

[1] *Exercise*: Satisfy yourselves that these conditional forms are equivalent to the corresponding disjunctive forms.

SAT is easily seen to be NP. Given a candidate certificate, say

$$A = true, B = false, C = false, D = true, E = false$$

for the instance above, the time it takes to check whether it really is a certificate is in fact *linear* in the size of the instance (double the size of the instance, and you double the time).[2]

We shall show that *any* NP problem can be reduced to SAT in polynomial time. First, we must define what we mean by reducing one problem to another.

3 Problem reduction

A decision problem D_1 can be **reduced** to a decision problem D_2 if there is an algorithm which takes as input an arbitrary instance I_1 of D_1 and delivers as output an instance I_2 of D_2 such that I_2 is a positive instance of D_2 if and only if I_1 is a positive instance of D_1.

If D_1 can be reduced to D_2, and we have an algorithm which solves D_2, then we thereby have an algorithm which solves D_1. To solve an instance I of D_1 we first use the reduction algorithm to generate an instance I' of D_2, and then use the algorithm for solving D_2 to determine whether or not I' is a positive instance of D_2. If it is, then we know that I is a positive instance of D_1, and if it is not, then we know that I is a negative instance of D_1. Either way, we have solved D_1 for that instance.

Moreover, in this case, we can say that the computational complexity of D_1 is at most the sum of the computational complexities of D_2 and the reduction algorithm. If the reduction algorithm has polynomial complexity, we say that D_1 is **at most polynomially harder** than D_2. It means that the amount of computational work we have to do to solve D_1, over and above whatever is required to solve D_2, is polynomial in the size of the problem instance. In such a case the reduction algorithm provides us with a feasible way of solving D_1, given that we know how to solve D_2.

4 Cook's Theorem

Cook's Theorem states that

Any NP problem can be reduced to SAT in polynomial time.

In order to prove this, we require a uniform way of representing NP problems. Remember that what makes a problem NP is the existence of a polynomial-time algorithm—more specifically, a Turing machine—for checking candidate certificates. What Cook did was somewhat analogous to what Turing did when he showed that the *Entscheidungsproblem* was equivalent to the Halting Problem. He showed how to encode as Propositional Calculus clauses both the relevant facts about the problem instance and the Turing machine which does the certificate-checking, in such a way that the resulting set of clauses is satisfiable if and only if the original problem instance is positive. Thus the problem of determining the latter is reduced to the problem of determining the former.

Assume, then, that we are given an NP decision problem D. By the definition of NP, there is a polynomial function P and a Turing machine M which, when given any instance I of D, together with a candidate certificate c, will check in time no greater than $P(n)$, where n is the length of I, whether or not c is a certificate of I.

Let us assume that M has q states numbered $0, 1, 2, \ldots, q-1$, and a tape alphabet $a_1, a_2, \ldots, a_s$. We shall assume that the operation of the machine is governed by the functions T, U, and D as described in the chapter on the *Entscheidungsproblem*. We shall further assume that the initial tape is

[2] *Exercise*: Check whether the candidate certificate just given is or is not a certificate for the instance above. Does this tell you whether the instance is a positive instance of SAT?

inscribed with the problem instance on the squares $1, 2, 3, \ldots, n$, and the putative certificate on the squares $-m, \ldots, -2, -1$. Square zero can be assumed to contain a designated separator symbol. We shall also assume that the machine halts scanning square 0, and that the symbol in this square at that stage will be a_1 if and only if the candidate certificate is a true certificate. Note that we must have $m \leq P(n)$. This is because with a problem instance of length n the computation is completed in at most $P(n)$ steps; during this process, the Turing machine head cannot move more than $P(n)$ steps to the left of its starting point.

We define some atomic propositions with their intended interpretations as follows:

1. For $i = 0, 1, \ldots, P(n)$ and $j = 0, 1, \ldots, q-1$, the proposition Q_{ij} says that after i computation steps, M is in state j.
2. For $i = 0, 1, \ldots, P(n)$, $j = -P(n), \ldots, P(n)$, and $k = 1, 2, \ldots, s$, the proposition S_{ijk} says that after i computation steps, square j of the tape contains the symbol a_k.
3. $i = 0, 1, \ldots, P(n)$ and $j = -P(n), \ldots, P(n)$, the proposition T_{ij} says that after i computation steps, the machine M is scanning square j of the tape.

Next, we define some clauses to describe the computation executed by M:

1. *At each computation step, M is in at least one state.* For each $i = 0, \ldots, P(n)$ we have the clause

   $$Q_{i0} \vee Q_{i1} \vee \cdots \vee Q_{i(q-1)},$$

   giving $(P(n)+1)q = O(P(n))$ literals altogether.

2. *At each computation step, M is in at most one state.* For each $i = 0, \ldots, P(n)$ and for each pair j, k of distinct states, we have the clause

   $$\neg(Q_{ij} \wedge Q_{ik}),$$

   giving a total of $q(q-1)(P(n)+1) = O(P(n))$ literals altogether.

3. *At each step, each tape square contains at least one alphabet symbol.* For each $i = 0, \ldots, P(n)$ and $-P(n) \leq j \leq P(n)$ we have the clause

   $$S_{ij1} \vee S_{ij2} \vee \cdots \vee S_{ijs},$$

   giving $(P(n)+1)(2P(n)+1)s = O(P(n)^2)$ literals altogether.

4. *At each step, each tape square contains at most one alphabet symbol.* For each $i = 0, \ldots, P(n)$ and $-P(n) \leq j \leq P(n)$, and each distinct pair a_k, a_l of symbols we have the clause

   $$\neg(S_{ijk} \wedge S_{ijl}),$$

   giving a total of $(P(n)+1)(2P(n)+1)s(s-1) = O(P(n)^2)$ literals altogether

5. *At each step, the tape is scanning at least one square.* For each $i = 0, \ldots, P(n)$, we have the clause

   $$T_{i(-P(n))} \vee T_{i(1-P(n))} \vee \cdots \vee T_{i(P(n)-1)} \vee T_{iP(n)},$$

   giving $(P(n)+1)(2P(n)+1) = O(P(n)^2)$ literals altogether.

6. *At each step, the tape is scanning at most one square.* For each $i = 0, \ldots, P(n)$, and each distinct pair j, k of tape squares from $-P(n)$ to $P(n)$, we have the clause

   $$\neg(T_{ij} \wedge T_{ik}),$$

   giving a total of $2P(n)(2P(n)+1)(P(n)+1) = O(P(n)^3)$ literals.

7. *Initially, the machine is in state 1 scanning square 1.* This is expressed by the two clauses

   $$Q_{01}, T_{01},$$

   giving just two literals.

8. *The configuration at each step after the first is determined from the configuration at the previous step by the functions T, U, and D defining the machine M.* For each $i = 0, \ldots, P(n)$, $-P(n) \leq j \leq P(n)$, $k = 0, \ldots, q-1$, and $l = 1, \ldots, s$, we have the clauses

   $$\begin{array}{rcl} T_{ij} \wedge Q_{ik} \wedge S_{ijl} & \rightarrow & Q_{(i+1)T(k,l)} \\ T_{ij} \wedge Q_{ik} \wedge S_{ijl} & \rightarrow & S_{(i+1)jU(k,l)} \\ T_{ij} \wedge Q_{ik} \wedge S_{ijl} & \rightarrow & T_{(i+1)(j+D(k,l))} \\ S_{ijk} & \rightarrow & T_{ij} \vee S_{(i+1)jk} \end{array}$$

   The fourth of these clauses ensures that the contents of any tape square other than the currently scanned square remains the same (to see this, note that the given clause is equivalent to the formula $S_{ijk} \wedge \neg T_{ij} \rightarrow S_{(i+1)jk}$). These clauses contribute a total of $(12s+3)(P(n)+1)(2P(n)+1)q = O(P(n)^2)$ literals.

9. *Initially, the string $a_{i_1}a_{i_2} \ldots a_{i_n}$ defining the problem instance I is inscribed on squares $1, 2, \ldots, n$ of the tape.* This is expressed by the n clauses

   $$S_{01i_1}, S_{02i_2}, \ldots, S_{0ni_n},$$

   a total of n literals.

10. *By the $P(n)$th step, the machine has reached the halt state, and is then scanning square 0, which contains the symbol a_1.* This is expressed by the three clauses

    $$Q_{P(n)0}, S_{P(n)01}, T_{P(n)0},$$

    giving another 3 literals.

Altogether the number of literals involved in these clauses is $O(P(n)^3)$ (in working this out, note that q and s are constants, that is, they depend only on the machine and do not vary with the problem instance; thus they do not contribute to the growth of the the number of literals with increasing problem size, which is what the O notation captures for us). It is thus clear that the procedure for setting up these clauses, given the original machine M and the instance I of problem D, can be accomplished in polynomial time.

We must now show that we have succeeded in reducing D to SAT. Suppose first that I is a positive instance of D. This means that there is a certificate c such that when M is run with inputs c, I, it will halt scanning symbol a_1 on square 0. This means that there is some sequence of symbols that can be placed initially on squares $-P(n), \ldots, -1$ of the tape so that all the clauses above are satisfied. Hence those clauses constitute a positive instance of SAT.

Conversely, suppose I is a negative instance of D. In that case there is *no* certificate for I, which means that *whatever* symbols are placed on squares $-P(n), \ldots, -1$ of the tape, when the computation halts the machine will not be scanning a_1 on square 0. This means that the set of clauses above is not satisfiable, and hence constitutes a negative instance of SAT.

Thus from the instance I of problem D we have constructed, in polynomial time, a set of clauses which constitute a positive instance of SAT if and only I is a positive instance of D. In other words, we have reduced D to SAT in polynomial time. And since D was an arbitrary NP problem it follows that *any* NP problem can be reduced to SAT in polynomial time.

5 NP-completeness

Cook's Theorem implies that any NP problem is at most polynomially harder than SAT. This means that if we find a way of solving SAT in polynomial time, we will then be in a position to solve any NP problem in polynomial time. This would have huge practical repercussions, since many frequently encountered problems which are so far believed to be intractable are NP.

This special property of SAT is called **NP-completeness**. A decision problem is NP-complete if it has the property that any NP problem can be reduced to it in polynomial time. SAT was the first NP-complete problem to be recognised as such (the theory of NP-completeness having come into existence with the proof of Cook's Theorem), but it is by no means the only one. There are now literally thousands of problems, cropping up in many different areas of computing, which have been proved to be NP-complete.

In order to prove that an NP problem is NP-complete, all that is needed is to show that SAT can be reduced to it in polynomial time. The reason for this is that the sequential composition of two polynomial-time algorithms is itself a polynomial-time algorithm, since the sum of two polynomials is itself a polynomial.[3]. Suppose SAT can be reduced to problem D in polynomial time. Now take any NP problem D'. We know we can reduce it to SAT in polynomial time, and we know we can reduce SAT to D in polynomial time. The result of these two conversions is a polynomial-time conversion of D' to D. Since D' was an arbitrary NP problem, it follows that D is NP-complete.

We illustrate this by showing that the problem 3SAT is NP-complete. This problem is similar to SAT, but restricts the clauses to at most three schematic letters each:

> Given a finite set $\{C_1, C_2, \ldots, C_n\}$ of clauses, each of which contains at most three schematic letters, determine whether there is an assignment of truth-values to the schematic letters appearing in the clauses which makes all the clauses true.

3SAT is obviously NP (since it is a special case of SAT, which is NP). It turns out to be straightforward to reduce an arbitrary instance of SAT to an instance of 3SAT with the same satisfiability property.

Take any clause written in disjunctive form as $C \equiv L_1 \vee L_2 \vee \ldots \vee L_n$, where $n > 3$ and each L_i is a literal. We replace this by $n-2$ new clauses, using $n-3$ new schematic letters $X_1, \ldots, X_{n-3}$, as follows:

$$\begin{array}{l}
L_1 \vee L_2 \vee X_1 \\
X_1 \rightarrow L_3 \vee X_2 \\
X_2 \rightarrow L_4 \vee X_3 \\
\qquad\quad \vdots \\
X_{n-4} \rightarrow L_{n-2} \vee X_{n-3} \\
X_{n-3} \rightarrow L_{n-1} \vee L_n
\end{array}$$

[3] For example, the sum of $x^5 - 5x^3 + 3$ and $4x^4 - x^2 - 8$ is $x^5 + 4x^4 - 5x^3 - x^2 - 5$

Call the new set of clauses $\mathcal{C}$. Any truth-assignment to the schematic letters appearing in the L_i which satisfies C can be extended to the X_i so that $\mathcal{C}$ is satisfied, and conversely any truth-assignment which satisfies $\mathcal{C}$ also satisfies C.

To prove this, suppose that a certain truth-assignment satisfies C. Then it satisfies at least one of the literals appearing in C, say L_k. Now assign $true$ to $X_1, X_2, \ldots, X_{k-2}$ and $false$ to $X_{k-1}, \ldots, X_{n-3}$. Then all the clauses in $\mathcal{C}$ are satisfied: for $i = 1, 2, \ldots, k-2$, the ith clause is satisfied because X_i is true; the $(k-1)$th clause is satisfied because L_k is true; and for $j = k, k+1, \ldots, n-2$ the jth clause is satisfied because X_{j-1} (appearing in the antecedent) is false.

Conversely, suppose we have a truth-assignment satisfying $\mathcal{C}$: each clause in $\mathcal{C}$ is satisfied. Suppose $L_1, \ldots, L_{n-2}$ are all false. Then it is easy to see that all the X_i must be true; in particular X_{n-3} is true, so either L_{n-1} or L_n is true. Thus in any event at least one of the L_i is true, and hence C is true.

If we take an instance of SAT and replace *all* the clauses containing more than three literals by clauses containing exactly three in the way described above, we end up with an instance of 3SAT which is satisfiable if and only if the original instance is satisfiable. Moreover, the conversion can be accomplished in polynomial time. It follows that 3SAT, like SAT, is NP-complete.

6 P=NP?

We have seen that a problem is NP-complete if and only if it is NP and any NP problem can be reduced to it in polynomial time. (A problem satisfying the second condition is called **NP-hard**; so NP-complete means NP *and* NP-hard.) It follows from this that all NP-complete problems are mutually interreducible in polynomial time. For if D_1 and D_2 are both NP-complete, then D_1 can be reduced to D_2 by virtue of the fact that D_1 is NP and D_2 is NP-hard, and D_2 can be reduced to D_1 by virtue of the fact that D_2 is NP and D_1 is NP-hard. Thus as far as computational complexity is concerned, all NP-complete problems agree to within some polynomial amount of difference.

But what is the computational complexity of these problems? If any one NP-complete problem can be shown to be of polynomial complexity, then by the above they all are. If on the other hand any one NP-complete problem can be shown not to be solvable in polynomial time, then by the above, none of them are so solvable. All NP-complete problems stand or fall together.

The current state of our knowledge is this: we know how to solve NP-complete problems in *exponential* time, but there is no NP-complete problem for which any algorithm is known that runs in less than exponential time. On the other hand, no-one has ever succeeded in proving that it is not possible to solve an NP-complete problem faster than that. This implies, of course, that no-one has proved that NP-complete problems can't be solved in *polynomial* time.

If we could solve NP-complete problems in polynomial time, then the whole class NP would collapse down into the class P of problems solvable in polynomial time. This is because the NP-complete problems are the *hardest* of all NP problems, and if they are polynomial then all NP problems are polynomial. Thus the question of whether or not there are polynomial algorithms for NP-complete problems has become known as the "P=NP?" problem. Most people who have an opinion on this believe that the answer is no, that is, NP-complete problems are strictly harder than polynomial problems.

All this assumes, of course, the Turing model of computation, which applies to all existing computers. However, as suggested above, if we had access to unlimited parallelism, we could solve any NP problem (including therefore the NP-complete problems) in polynomial time. Existing computers do not provide us with such parallelism; but if the current work on Quantum Computation comes to fruition, then it may be that a new generation of computers will have exactly such abilities, and if that happens, then everything changes. It would not solve the classic P=NP question, of course, because that question concerns the properties of Turing-equivalent computation; what would happen is that the theory of Turing-equivalent computation would suddenly become much less relevant to practical computing concerns, making the question only of theoretical interest.

7 Bibliographic notes

The treatment of Cook's Theorem and the conversion of SAT to 3SAT above are based on the account given by Herbert S. Wilf, *Algorithms and Complexity* (Prentice-Hall International, 1986, now out of print, but there are two copies in the library).

A classic exposition of the theory of NP-completeness, with many examples of NP-complete problems, is M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness* (W. H. Freeman, 1979).

Other useful sources for this material are:

- Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall International, 1998 (Chapter 7: NP-completeness)
- John C. Martin, *Introduction to Languages and the Theory of Computation*, McGraw-Hill, 1991 (Chapter 23: Tractable and Intractable Problems)
- Thomas A. Sudkamp, *Languages and Machines: An Introduction to the Theory of Computer Science*, Addison-Wesley, 1988 (Chapter 14: Computational Complexity)
- V. J. Rayward-Smith, *A First Course in Computability*, Blackwell Scientific Publications, 1986 (Chapter 6: Complexity Theory)

All these books are in the university library. For a less technical overview, consult the two books by David Harel: *Algorithmics: The Spirit of Computing* (Addison-Wesley, 1987) and *Computers Ltd: What They* Really *Can't Do* (OUP, 2000).