13. POLYNOMIAL-TIME REDUCTIONS

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

November 2024

During the chapter on Computability Theory, we have studied *many-one reductions*: a way of transforming the input of a problem into the input of another, maintaining the answer (`TRUE` or `FALSE`).

Given a new problem, we employed reductions to:

- prove that the new problem is undecidable by reducing a known undecidable problem to it
- prove that the new problem is decidable by reducing it to a known decidable problem

We will extend this technique now to allows classifying decidable problems further, into those that admit a polynomial-time solution and those that don't. In the next lecture, we will see that this technique allows for a more subtler classification, involving the class of **NP**-**C**omplete problems. These are, under a reasonable rigorous definition, the "hardest" problems in **NP**, which may or may not be solvable in polynomial time, depending on the relationship between **P** and **NP**.

1 Polynomial-time many-one reductions

In this section, let $f : \Sigma^* \to \{\texttt{FALSE}, \texttt{TRUE}\}$ and $g : \Sigma^* \to \{\texttt{FALSE}, \texttt{TRUE}\}$ be two decision problems over some alphabet Σ.

Definition 13.1.

$$f \leq_P g \overset{\text{def}}{=} \exists t : \Sigma^* \to \Sigma^*, \text{ s.t. } \begin{cases} t \text{ is computable in polynomial time} \\ \forall w \in \Sigma^\star, f(w) = g(t(w)) \end{cases}$$

Read as "*f is polynomial-time reducible to g*"[†].

Note that this is almost the exact same definition we presented before for the notion of reducibility; the only difference is the polynomial-time requirement for computing t.

Theorem 13.1. $(g \in \mathbf{P}) \wedge (f \leq_P g) \Rightarrow f \in \mathbf{P}$

Proof. If $g \in \mathbf{P}$, then there exists a TM M_g that decides g in polynomial time $T_{M_g}(n)$.

If $f \leq_P g$, then there exists a TM M_t which computes a transformation from inputs of f to inputs of g in polynomial time $T_{M_t}(n)$.

The composition of M_t and M_g is a TM that decides f in polynomial time $O(T_{M_t}(n) + T_{M_g}(n))$.

□

Theorem 13.2. $(f \notin \mathbf{P}) \wedge (f \leq_P g) \Rightarrow g \notin \mathbf{P}$

This is a direct consequence of our previous theorem.

In a future lecture, we will refine this theorem to tells us something more useful about the class **NPC**.

[†]It would be more correct to read this as "f is polynomial-time *many-one* reducible to g". There are other types of polynomial-time reductions; however, we won't address them at this course, so we will omit this extra qualification.

2 VERTEX COVER $\leq_P$ CLIQUE

Our first reduction will be an easy one. We will go from searching for a cover to searching for a clique.

The input to VERTEX COVER consists of a graph and a number, same as the input to CLIQUE. Which means that the identity function is a decent candidate: it is computable in polynomial time (simply do nothing) and it turns a valid input of one problem into a valid input for the other.

It fails in the second criteria: a graph with a K-sized cover, doesn't necessarily have a K-sized clique and vice-versa.

The transformation we need is still quite simple. We need the *complement graph* of G.

Definition 13.2. Let $G = (V, E)$ be a graph; its complement is:

$$\overline{G} \stackrel{\text{def}}{=} (V, \{(u, v) \mid u, v \in V \wedge (u, v) \notin E\})$$

The complement of a graph G is another graph with the same nodes but with exactly the edges that are missing from G.

The transformation we need is $t(G = (V, E), K) = (G', |V| - K)$.

The first claim here (the "$\Rightarrow$" direction) is that if a graph with $|V|$ nodes has a cover of size K, then its complement has a clique of size $|V| - K$. Why is that?

Imagine we delete all the nodes in the cover and all the edges attached to them. But, because it's a cover, that means we delete all the edges of the graph! What we are left with is $|V| - K$ nodes with no edge between any pair of them. So in the complement graph, all these nodes will have an edge towards all other nodes – a clique!

The second claim (the "$\Leftarrow$") is that if the complement has a clique of size $|V| - K$, then the original graph has a cover of size K. In the original graph, there is no edge between any of the nodes in the clique; so all the other nodes (K of them!) must form a cover.

3 CLIQUE $\leq_P$ VERTEX COVER

Notice that there is a very beautiful symmetry in the previous transformation, which leads to the observation that we can use the same argument to show that CLIQUE $\leq_P$ VERTEX COVER.

The transformation is exactly the same function: we take the input $(G = (V, E), K)$ for CLIQUE and turn it into an input $(G', |V| - K)$ for VERTEX COVER.

So a graph with $|V|$ nodes has a clique of size K only if its complement has a cover of size $|V| - K$.

Indeed, we can say that these two problems are *polynomially equivalent*. This is the case for all the problems that we introduced under the label "hard?". However, it is not always the case that both reductions are so similar or even that they are as easy to come up with.

4 SAT $\leq_P$ CNF SAT

First, we need to address how exactly a boolean formula can be represented as the input to a Turing Machine and consider what is its length.

Each formula has a set of n variables; instead of thinking of them as having unique names (such as x, y, z etc.), we can rename them all using a single number from 1 to n. Thus each variable can be represented by a $\lceil \log_2(n) \rceil$ bits binary number.

A formula is a string over the alphabet $\Sigma_f = \{0, 1, \neg, \wedge, \vee, (,)\}$.

Its length is the total number of symbols; it is certainly $\Omega(\lceil \log_2(n) \rceil)$, but there is not much else that we can say about it with regards to n (think of a formula consisting of a single variable negated an arbitrary amount of times).

Any boolean formula can be converted to an equivalent formula in Conjunctive Normal Form; here, by "equivalent" we mean that the resulting expression will have the same truth value as the original one for any truth assignment. But this conversion may result in an exponentially longer formula, so it is not useful for us at this point.

We need to relax the requirement and aim to obtain, for any logical formula, a new one in CNF that is not necessarily equivalent, but either both of them are satisfiable, or neither of them is.

The first step is to "push down" all negations, such that the only negated expressions are single variables.

We use *DeMorgan's laws* and the *law of double negation*, like this:

- $\overline{E_1 \wedge E_2}$ becomes $\overline{E_1} \vee \overline{E_2}$
- $\overline{E_1 \vee E_2}$ becomes $\overline{E_1} \wedge \overline{E_2}$
- $\overline{\overline{E}}$ becomes E

In the worst case, we have $O(|\phi|)$ "outermost" negation operators and for each of them we have to change the entire formula, so this can be done in time $O(|\phi|^2)$.

Now we have an expression that is a mix of conjunctions and disjunctions between literals (which are simple variables or negated variables).

We show next that there exists a transformation t_{CNF} which takes an arbitrary boolean formula ϕ and converts it into a CNF boolean formula $t_{CNF}(\phi)$ that can be computed in polynomial time.
Crucially: ϕ satisfiable $\Leftrightarrow t_{CNF}(\phi)$ satisfiable .

Base case: If E is of the form x or $\overline{x}$ then $t(E) = E$ is already in CNF.

Inductive step: Assume that every expression shorter than E can be converted to one in CNF (strong induction).

- $E = E_1 \wedge E_2$: by the inductive hypothesis, there are expressions $F_1 = t_{CNF}(E_1)$ and $F_2 = t_{CNF}(E_2)$ in CNF, constructable in polynomial time from E_1 and E_2 respectively. We set $F = F_1 \wedge F_2$; F is in CNF.

  If E is satisfiable, there is some truth assignment T that satisfies both E_1 and E_2. We assume that F_1 and F_2 share only the variables in E; if we have to add new variables that were not present in E_1 and E_2, we add distinct variables.

  Then T can be extended to a truth assignment that satisfies F_1 and F_2 and so F.

- $E = E_1 \vee E_2$: by the inductive hypothesis, there are expressions $F_1 = t_{CNF}(E_1)$ and $F_2 = t_{CNF}(E_2)$ in CNF, constructable in polynomial time from E_1 and E_2 respectively. F_1 being in CNF means that it's a conjunction of shorter formulas: $(G_1 \wedge G_2 \wedge ... \wedge G_k)$; same for $F_2 = (H_1 \wedge H_2 \wedge ... \wedge H_l)$.

  We introduce a **new** variable x and let:
  $F = (x \vee G_1) \wedge (x \vee G_2) \wedge ... \wedge (x \vee G_k) \wedge (\overline{x} \vee H_1) \wedge (\overline{x} \vee H_2) \wedge ... \wedge (\overline{x} \vee H_l)$.

  If E is satisfiable, there is a truth assignment T that satisfies either E_1 or E_2.

  Assuming that T satisfies E_1, then T can be extended to assign $x =$ `FALSE`, thus satisfying all the clauses derived from F_2; note that those derived from F_1 are also satisfied.. Symmetrically, assuming that T satisfies E_1, then it can be extended to assign $x =$ `TRUE`, thus satisfying all the clauses derived from F_1.

Note that we treat conjunctions/disjunctions as operators with two operands; any "long-chain" of such operations, such as $E_1 \vee E_2 \vee E_3 \vee ... \vee E_m$ should be though of as a sloppy version of writing $((...((E_1 \vee E_2) \vee E_3) \vee ... \vee E_m)$.

This conversion can also be done in quadratic time, although the rigorous proof of this is more difficult. We will skip it for now and simply point out that at each step we need to split the current expression into two smaller subexpressions which we will process separately then combine into our final result. This gives rise to a recurrence relation that can be proven to have a complexity of $\Theta(|\phi|^2)$.

In a future lecture we will approach the subject of complexities defined by recurrence relations and how to calculate them.

5 `CNF SAT` $\leq_P$ `3SAT`

We push the idea of simplifying the formulas structure further to exactly three literals per clause and show that, under this new constraint, the problem remains just as hard[‡].

The idea of the transformation is to go through each clause of a `CNF SAT` formula ϕ and transform it into one or more clauses that adhere to the `3SAT` structure.

With regards to the number of variables, a `CNF SAT` can contain several types of clauses; we describe next how to transform them to clauses with exactly three variables:

[‡]Actually, here we are proving that `3SAT` is at least as hard as `CNF SAT`; to support this equivalence statement we need the reduction: `3SAT` $\leq_P$ `CNF SAT`; but here the transformation can be the identity function: any `3SAT` formula is already a `CNF SAT` one.

- clauses with a single literal (of the form "(L)"): create two **new** variables x and y (that are not part of the variables of ϕ) and replace the clause with the following ones: $(L \vee x \vee y) \wedge (L \vee \overline{x} \vee y) \wedge (L \vee x \vee \overline{y}) \wedge (L \vee \overline{x} \vee \overline{y})$. Note that these new clauses together are satisfiable only if (L) is satisfied.

- clauses with two literals (of the form "$(L_1 \vee L_2)$"): create one **new** variable x and replace the clause with the following ones: $(L_1 \vee L_2 \vee x) \wedge (L_1 \vee L_2 \vee \overline{x})$. Note that these new clauses together are satisfiable only if $(L_1 \vee L_2)$ is satisfied.

- clauses with three literals are left as they are; no need to do anything

- clauses with $k > 3$ literals (of the form "$(L_1 \vee L_2 \vee L_3 \vee ... \vee L_k)$"): create $k-3$ **new** variables $x_1, x_2, ..., x_{k-3}$ and replace the clause with the following ones: $(L_1 \vee L_2 \vee x_1) \wedge (\overline{x_1} \vee L_3 \vee x_2) \wedge (\overline{x_2} \vee L_4 \vee x_3) \wedge ... \wedge (\overline{x_{k-3}} \vee L_{k-1} \vee L_k)$.

For the first two cases, it is easy to see (for example, by exhaustively going thorough all possible cases) that the resulting conjunction of clauses is satisfiable if and only if the original clause was satisfied. We must now prove this for the last case.

First, we show that if the initial clause is satisfied so is the resulting conjunction.

There must be some literal L_i whose value is `TRUE`. If $i = 1$ or $i = 2$ the first clause is `TRUE`; set all variables x `FALSE`; except for the first clause, there is a negated x in all other clauses to make them `TRUE`, so the conjunction is `TRUE`. Similarly, if $i = k - 1$ or $i = k$, set all variables x `TRUE`.

If i is between 2 and $k - 1$, there is one clause of the form $(\overline{x_{i-2}}, L_i, x_{i-1})$ that is `TRUE`; set all $x_1, ..., x_{i-2}$ `TRUE` and all $x_{i-1}, ..., x_{k-2}$ `FALSE`.

We now show the contrapositive of the reverse direction: if the original clause is unsatisfied, the resulting clause is unsatisfiable.

For the original clause to be unsatisfied, all literals must be `FALSE`. The first clause of the resulting conjunction, $(L_1 \vee L_2 \vee x_1)$ can only be `TRUE` if $x_1 =$ `TRUE`. But this means the second clause, $(\overline{x_1} \vee L_3 \vee x_2)$ can only be `TRUE` if $x_2 =$ `TRUE` and so on for all xs. But then $x_{k-3} =$ `TRUE` which means that the last clause $(\overline{x_{k-3}} \vee L_{k-1} \vee L_k)$ is `FALSE`.

Lastly, let's consider the time complexity of the transformation: we inspect each clause once and expand it to a number of clauses proportional to the number of literals. This can be done in polynomial time.

6 3SAT $\leq_P$ CLIQUE

This one is our first reduction that involves two very different concepts: we must find a transformation that turns a logical formula into a graph.

Let ϕ be a 3SAT formula with n clauses and m variables.

In the resulting graph G, for each clause of ϕ we build a group of 3 nodes, one for each literal in the clause; $|V| = m$. We then draw edges between any two nodes, except those corresponding to literals in the same clause and to *contradicting literals* (e.g. x and $\overline{x}$).

Finally, we set $K = n$, the number of clauses.

If ϕ has a satisfying assignment, there is at least one `TRUE` literal in each clause; in G, we choose for each group of 3 nodes the one corresponding to this literal[§] – a total of n nodes.

These nodes form a clique: each pair of them is connected by an edge: they are not in the same clause and they are not contradicting literals, because we chose only literals that are `TRUE`.

In the other direction, if G has a clique of size K, then these nodes must necessarily be from different triples. We create a truth assignment for ϕ that sets all the literals in the clique to `TRUE` (we can do this, because if the two nodes correspond to contradicting variables, then there is no edge between them so they can't be part of a clique). All n clauses have a `TRUE` literal, so ϕ is satisfiable.

7 Decision problems and their optimization counterpart

We have seen how to transform the input of a decision problem f into the input for a decision problem g, such that g's answer is the same as f's.

[§]If there is more than one `TRUE` literal, we arbitrarily choose one of them.

But not all real-world problems are decision problems; actually, some of our formulations may seem artificial: for problems such as VERTEX COVER, it seems more natural to ask: "*what is the minimum size of a cover*?". These are *optimization problems*; they seek to find an optimal solution: the smallest or biggest object that satisfies some condition.

To justify restricting our focus on decision problems, we will now show a straightforward way to relate the time-complexity of a decision problem with that of its optimization counterpart. We will show a reduction between VERTEX COVER and MINIMUM VERTEX COVER, as well as vice versa. This won't be a "*many-one reduction*", as the two objects don't share a codomain. But our new reduction $f \leq_T g$ maintains the following characteristic of our previous reductions: **if g can be solved in polynomial time, then f can be solved in polynomial time; if f cannot be solved in polynomial time, then g cannot be solved in polynomial time**.

The key to our reduction is to consider that we have a black box that can solve g. We are not interested at all in the structure of this box: whether it is a Turing Machine or some magical object, whether it runs in polynomial time or exponential time etc. In Computability and Complexity theory, such a black box is called an "oracle". We will then construct an *oracle machine* to solve f; this is a normal Turing Machine that can use the oracle for g as a subroutine. A rigorous definition can be given for oracle Turing Machines, but we shall skip the details and instead simply write pseudocode, where the oracle for g is a subroutine with an unspecified implementation.

For our reduction to be useful, we will posit the following restrictions:

- the oracle machine for solving f can only make a number of calls to the g oracle **polynomial in the size of the input**
- the length of output of the g oracle must be polynomial in the size of its input
- ignoring calls to the oracle for g, the oracle machine for solving f must run in polynomial time (we can also say that a call to the oracle for g has a constant time complexity $\Theta(1)$)

7.1 VERTEX COVER $\leq_T$ MINIMUM VERTEX COVER

This is the simplest direction. We have an oracle – procedure SOLVE_MINIMUM_VERTEX_COVER that takes as input a graph G and returns a number: the size of the minimal cover.

Algorithm 13.1 Oracle machine for VERTEX COVER

```
1: function SOLVE_VERTEX_COVER(G = (V, E), K)
2:     if K > |V| then
3:         return FALSE
4:     min_cover_size ← SOLVE_MINIMUM_VERTEX_COVER(G)
5:     return min_cover_size ≤ K
```

Considering that the call to the oracle takes a single step, we can see that the complexity for this is linear (the complexity of comparing two numbers).

7.2 MINIMUM VERTEX COVER $\leq_T$ VERTEX COVER

We need now to show that having an oracle for VERTEX COVER allows us to solve the optimization problem in polynomial time.

We can simply iterate through all possible cover sizes, from 0 to $|V|$:

Algorithm 13.2 Oracle machine for MINIMUM VERTEX COVER

```
1: function SOLVE_MINIMUM_VERTEX_COVER(G = (V, E))
2:     for K ← 0 to |V| do
3:         if SOLVE_VERTEX_COVER(G, K) then
4:             return K
```

In fact, we can do better, by using binary search:

Algorithm 13.3 Oracle machine for `MINIMUM VERTEX COVER`

```
1: function BINARY_SEARCH(G = (V, E), L, R)
2:     if L = R then
3:         return L
4:     K ← ⌊(L+R)/2⌋
5:     answer ← SOLVE_VERTEX_COVER(G, K)
6:     if answer then
7:         return BINARY_SEARCH(G, L, K)
8:     else
9:         return BINARY_SEARCH(G, K + 1, R)

10: function SOLVE_MINIMUM_VERTEX_COVER(G = (V, E))
11:     return BINARY_SEARCH(G, 1, |V|)
```

In a later lecture, we will learn exactly how to calculate the complexity of such a recursive function. For now, we will consider it common knowledge that binary searching among n items has a complexity of $\Theta(\log n)$.

This completes what we set out to do: to show that if we can solve the decision problem in polynomial time, we can also solve the optimization problem in polynomial time. It's a good illustration of why polynomials are such a convenient class: in this case we composed two polynomials (the decision oracle and the code that calls it) and obtained another polynomial, because polynomials are *closed under composition*.

8 References and further reading

The reduction in section 4 is adapted from [1] (Chapter 10, page 449). To keep things short, we skipped some rigorous details that guarantee the correctness of our transformations, so we recommend reading section 10.3.3 of the book in its entirety.

The reduction in section 6 is adapted from [2] (Chapter 7, page 302).

Bibliography

[1] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to Automata Theory, Languages, and Computation, 3rd Edition*. Addison-Wesley, 2007.

[2] Michael Sipser. *Introduction to the Theory of Computation, Third Edition*. CENGAGE Learning, 2012.