12. "HARD?" PROBLEMS

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

November 2024

We ended the previous lecture by identifying a large class of problems that we deem *"tractable"*. The class P, consisting of all problems that can be solved in polynomial time might seem too generous: a problem which is in $\mathbf{DTIME}(n^8)$, but not in $\mathbf{DTIME}(n^7)$ (or lower) certainly doesn't seem feasible from a practical point of view.

Perhaps less controversial is the fact that any problem *outside of* P is intractable[†]. For example, consider a problem whose optimal solution needs 2^n transitions. For an input size of $n = 100$, the number of transitions needed to find the answer is astronomical: 1267650600228229401496703205376.

And 100 is not a large input size; it's easy to envision a graph with much more than 100 nodes (e.g. the web, the "friendship" relations in a social network) or a list with much more than 100 items (e.g. client data for an online retail service) etc. In comparison, even a large polynomial like n^8, already has a lower value at 100, namely 10000000000000000.

In fact, no matter how large the power of the polynomial, there is always some input size at which the exponential gets larger: $\forall k \in \mathbb{N}, n^k \in o(2^n)$.

Surprisingly, proving that a problem is or isn't in P is a very difficult task. In this lecture we shall do a tour-de-force of several well-known, practically important problems, whose precise difficulty is not known! No polynomial-time solutions are known for these problems; during many decades of study by top researchers in the fields of complexity and algorithms, no one has managed to find such a solution. Which is why it is widely believed, with varying levels of justification, that such solutions don't exist and these problems are not in P.

All this is related to perhaps the most famous open-problem in computer science: *the P versus NP* problem. In the next few lectures, we shall find out exactly what the class NP is[‡]. We will also introduce a more precise terminology for problems considered to be difficult: they are *NP-complete* and we shall see that if any of them is proven to be in P, then it follows that all of them are.

For each of the *"hard?"* problems presented here, we will follow the same procedure.

First, we will *define* the problem: on what kinds of inputs it operates, what it wants to determine etc. Most of the problems here have a simple description that depends on a specialized term that we will define alongside the problem. All the problems have something else in common: they ask for the existence of some object(s) that depend on the input. If such objects exist, the answer is `TRUE`, else it is `FALSE`.

We will then informally describe a simple algorithm to solve the problem in more-than-polynomial time. All of our algorithms will follow the same pattern: they generate a list of all possible *candidate solutions*, then check whether a particular candidate fits the criteria required by the problem. If any candidate does, the algorithm provides a positive answer; if all candidates fail, the algorithm provides a negative answer.

You shall see that all the "candidate verifications" run in polynomial time; later on we will formalize this as the class NP. So the "hard" part consists of generating all the candidates.

1 Graph problems

Before proceeding to see some example problems on graphs, it is worth recapitulating what graphs are and consider how a Turing Machine could work with them.

We consider graphs to be pairs $G = (V, E)$ of a set of *nodes* V and a set of edges E. We refer to each node as a number from 1 to $|V|$. An edge is a pair of nodes. We will generally be talking about *undirected graphs*, where an edge is an unordered pair.

[†]However, in real life, we often do employ solutions that have exponential worst-case time complexity.

[‡]Suffice it to say for now that its name stands for "Nondeterministic Polynomial time" and it is, roughly speaking, the class of problems with "easily-verifiable solutions".

We will use *adjacency matrices* to represent graphs. An adjacency matrix is a $|V|$ by $|V|$ table, where the nodes index both the lines and the columns; the i, j entry of the matrix is a boolean value which shows whether $(i, j) \in E$. Note that for undirected graphs, the matrix is symmetric with respect to its main diagonal.

The adjacency matrix itself contains all the information to describe the graph, so it can serve as the input for our algorithms.

We could represent each row of the matrix as a series of $|V|$ bits; then all $|V|$ rows can be separated by a special symbol, such as ";". This representation of a graph G can serve as the input to a Turing Machine.

If a problem requires other inputs, such as one or more node indices, or other integers, we can provide the binary encoding of them.

2 VERTEX COVER

Definition 12.1. VERTEX COVER: Does graph $G = (V, E)$ have a *cover* of size K?
A *cover* is a subset of nodes $C \subseteq V$, such that each edge has at least an end in C:

$$\forall (u, v) \in E, (u \in C \vee v \in C)$$

We can consider each subset of K nodes ($U \subseteq V, |U| = K$) and check if it is a cover.

How many such subsets are there? We turn to combinatorics:

- we need to fill a list of K elements
- the first element can be any one of the $|V|$ nodes of G
- once we set a first element, we "used up" one node, so we are left with only $|V| - 1$ options for the second node
- once we set a first and second element, we "used up" two nodes, so we are left with only $|V| - 2$ options for the third node and so on

But note that the *order* of the nodes doesn't matter, i.e. we are not interested in *arrangements*, but *combinations*. We can write this succinctly as: $\frac{|V|!}{(|V|-K)!K!}$.

We are now dealing with factorials, so we would like to know what we can say about the growth of the factorial function $n!$.

Theorem 12.1. $\forall k \in \mathbb{N}, n^k \in o(n!)$

So we have a more-than-exponential number of candidate subsets. We can of course generate these (think backtracking), but the time required to do so is *superpolynomial*, so we deem this approach *intractable*.

There is some ray of light here: once we have a particular candidate solution, we can "easily" check whether it is indeed a cover; consider the following pseudocode:

Algorithm 12.1 Polynomial checking of a candidate for VERTEX COVER

```
function CHECK_CANDIDATE(U):
    result ← TRUE
    for each (u, v) ∈ E do
        if u ∉ U ∧ v ∉ U then
            result ← FALSE
    return result
```

For each edge, we check whether it has at least one end inside the candidate subset; if not, then U doesn't cover at least this edge, so it is not a cover.

Remember that our graphs are represented as adjacency matrices; so iterating through all the edges can be done in $O(|V|^2)$ time. Checking if a node is part of U takes $O(|V|)$ ($U \subseteq V$): we check each element of U one by one.

The running time of `check_candidate` is polynomial. All problems presented in this lecture share this particular characteristic: there is a polynomial algorithm that correctly decides whether some candidate solution is actually a solution. In the next lecture, we will formalize this as a definition for the complexity class NP.

3 CLIQUE

Definition 12.2. CLIQUE: Does graph $G = (V, E)$ have a *clique* of size K?
A *clique* is a subset of nodes $C \subseteq V$, such that all nodes in the C are connected to each other:

$$\forall u, v \in C, (u, v) \in E$$

This problem is very similar to VERTEX COVER. Two lectures from now, we shall see that they are *reducible* one to the other (the same thing can be said about the rest of the problems presented here).

We have a superpolynomial number ($\frac{|V|!}{(|V|-K)!K!}$) of possible candidate subsets of nodes of size K.

We can check each of these quickly in polynomial time:

Algorithm 12.2 Polynomial checking of a candidate for CLIQUE

```
function CHECK_CANDIDATE(U):
    result ← TRUE
    for each u ∈ U do
        for each v ∈ U do
            if u ≠ v ∧ (u, v) ∉ E then
                result ← FALSE
    return result
```

4 COLORING

Definition 12.3. COLORING: Can graph $G = (V, E)$ be *colored* with K colors?
A *coloring of K colors* is a function $\texttt{color} : V \to \{c_1, c_2, ..., c_k\}$ such that:

$$\forall u, v \in V; (u, v) \in E \iff \texttt{color}(u) \neq \texttt{color}(v)$$

Informally, we want to color each node of the graph with one of several different colors, such that any adjacent nodes are colored differently.

We can solve this by considering all possible ways to color the nodes and see if any of them matches the adjacency condition.

Each of the $|V|$ nodes can take one of K colors, so we have a total $k^{|V|}$ colorings – an exponential number.

Given a coloring, we can verify its correctness in polynomial time:

Algorithm 12.3 Polynomial checking of a candidate for COLORING

```
function CHECK_CANDIDATE(U):
    result ← TRUE
    for each u ∈ U do
        for each v ∈ U do
            if u ≠ v ∧ (u, v) ∈ E ∧ color(u) = color(v) then
                result ← FALSE
    return result
```

5 HAMILTONIAN PATH

Definition 12.4. HAMILTONIAN PATH: Does graph $G = (V, E)$ have a *Hamiltonian path*?
A *Hamiltonian path* is a path of length $|V|$ with no duplicates.

Informally, we are searching for a path that goes through *each node exactly once*.

Knowing that such a path has $|V|$ nodes we can consider as candidates all permutations of the sequence $U = (1, 2, ..., |V|)$ of all nodes and check if it is a path (there is an edge between any two consecutive nodes).

There are $|V|!$ such permutations.

Given a permutation we can check whether it's a path in polynomial time:

Algorithm 12.4 Polynomial checking of a candidate for `HAMILTONIAN PATH`

```
function CHECK_CANDIDATE((u_1, u_2, ..., u_|V|)):
    result ← TRUE
    for each i ← 1, (|V| − 1) do
        if (u_i, u_{i+1}) ∉ E then
            result ← FALSE
     return result
```

6 TSP

The name stands for *"The Travelling Salesman Problem"*.

Definition 12.5. `TSP`: Given a *weighted* graph $G = (V, E)$ and a budget B, does there exist a Hamiltonian path of total cost lower than B?

A *weighted graph* is one in which each edge has a numerical value associated with it. These can be interpreted as distances, costs etc. We shall call it "cost".

The *total cost* of a path is simply the sum of the costs of all the edges in the path.

A weighted graph can be represented as an adjacency matrix in which each entry is not merely a single bit (0 if there's no edge, 1 if there's an edge), but rather an integer – the cost of an edge. Note that 0 still signifies that there is no edge.

The formulation of this problem seems a bit forced and that is because we wish to focus solely on decision problems. The more well-known version of this problem (and perhaps a more natural one) does not include a budget and simply requires finding the path *with the lowest cost*. We won't spend much time studying such *optimization problems*, but we will show in a future lecture how they relate to their decision version.

Like with `HAMILTON PATH`, we have $|V|!$ candidates.

Algorithm 12.5 Polynomial checking of a candidate for `TSP`

```
function CHECK_CANDIDATE((u_1, u_2, ..., u_|V|)):
    cost ← 0
    isPath ← TRUE
    for each i ← 1, (|V| − 1) do
        if (u_i, u_{i+1}) ∉ E then
            isPath ← FALSE
        else
            cost ← cost + C(u_i, u_{i+1})
     return isPath ∧ cost ≤ B
```

7 CUT

Definition 12.6. `CUT`: Given a graph $G = (V, E)$ and a number K, does there exist a *cut* of size at least K?

A *cut* of size K is a partitioning of V into a pair of complementary subsets V_1, V_2, such that the number of edges $(u, v) \in E$ such that $u \in V_1 \wedge v \in V_2$ is K.

A pair of complementary subsets simply means that $V_1 = V \setminus V_2$.

We are asked to split the graph into two "zones", such that a number of K edges cross the zone boundary.

There are $2^{|V|}$ candidates for V_1.

Algorithm 12.6 Polynomial checking of a candidate for `CUT`

```
function CHECK_CANDIDATE(V_1):
    crossings ← 0
    for each (u, v) ∈ E do
        if u ∈ V_1 ∧ v ∈ V \ V_1 then
            crossings ← crossings + 1
    return crossings ≥ K
```

8 SUBSET SUM

Definition 12.7. `SUBSET SUM`: Given a multiset S of integers and an integer K, is there a subset $S' \subseteq S$ such that: $\sum_{i \in S'} i = K$?

A *multiset* is a set with possible duplicates; what differentiates it from a sequence is that order is not important.

Like in problems concerning graphs, we can consider all possible subsets. There are precisely $2^{|S|}$ subsets of S (the size of its *powerset*) – an exponential number.

We can check if a candidate subset's element sum up to K in polynomial time:

Algorithm 12.7 Polynomial checking of a candidate for `SUBSET SUM`

```
function CHECK_CANDIDATE((s_1, s_2, ..., s_m)):
    sum ← 0
    for each i ← 1, m do
        sum ← sum + s_i
    return sum = K
```

9 PARTITIONING

Definition 12.8. `PARTITIONING`: Given a multiset S of integers, can this be partitioned into two complementary subsets with the same sum of elements?

Again, we can go through all possible $2^{|S|}$ subsets and see if any of them is a solution.

Algorithm 12.8 Polynomial checking of a candidate for `PARTITIONING`

```
function CHECK_CANDIDATE(S_1):
    sum_1 ← 0
    sum_2 ← 0
    for each s ∈ S do
        if s ∈ S_1 then
            sum_1 ← sum_1 + s
        else
            sum_2 ← sum_2 + s
    return sum_1 = sum_2
```

10 KNAPSACK

Definition 12.9. `KNAPSACK`: Given a set S of items characterized by *weight* and *value*, a weight limit W and a goal K, can we select a subset of items whose total weight is at most W and whose total value is at least K?

The problem's name hints at how you should visualize it: you have some container with a limited capacity in which you need to place some of your prized possessions. The goal is to get as much value as possible; intuitively, heavy items with low values are bad, while light items of high value are good.

More formally, our collection of items is a set of pairs (w_i, v_i). We are interested in whether a set $S' \subseteq S$ exists, such that:

- $\sum_{i \in S'} w_i \leq W$
- $\sum_{i \in S'} v_i \geq K$

There are $2^{|S|}$ possible subsets. Given one, it should now be easy to imagine how a polynomial-time verification works: we just go through all the elements, maintaining a sum of weights and a sum of values. At the end we return a conjunction of the target inequalities.

10.1 Dynamic programming and pseudopolynomials

If you studied algorithms before, you may be aware of a solution to this problem that leverages the technique of *dynamic programming*.

Algorithm 12.9 Dynamic programming solution for `KNAPSACK`

```
function SOLVE_KNAPSACK(S, W, K):
    Create table V of size (W + 1) × (|S| + 1)
    for each w ← 0, W do
        V[w][0] ← 0
    for each i ← 0, |S| do
        V[0][i] ← 0
    for each i ← 1, |S| do
        for each w ← 1, W do
            if w_i ≤ w then
                V[w][i] ← max(V[w][i − 1],   v_i + V[w − w_i][i − 1])
            else
                V[w][i] ← V[w][i − 1]

    return V[W][|S|] ≥ K
```

It helps to understand the solution if you interpret $V[w][i]$ as being the largest value possible with weight less than w, using any combination of the first i items.

But this solution seems polynomial! Its complexity is given by two nested loops, one ranging over $|S|$ values, the other over W values. So the overall complexity is $O(|S| \cdot W)$.

But remember what W is: an integer input of our problem. Its representation is the binary encoding of W; the length of the binary encoding is around $\log(W)$ and the same can be said about all the weights w_i.

The *length* of the input is actually on the order of $|S| \cdot \log(W)$. Thus $O(|S| \cdot W) = O(|S| \cdot 2^{\log(W)})$ is **exponential in the size of the input**.

The complexity of algorithms like this one for `KNAPSACK` is known as *pseudopolynomial*.

11 SAT

We now move on to discuss boolean formulas. Many problems can be expressed as some sort of boolean formula. Two lectures from now, we shall see that all the problems presented so far can be "reduced" to a boolean problem. We can actually prove that any problem in this category (NP-complete problems) can be reduced to `SAT`.

First, let's set some definitions related to boolean formulas.

Definition 12.10. A truth assignment T on a set of boolean variables X is a function: $T : X' \to \{FALSE, TRUE\}$, where $X' \subseteq X$.

Given a boolean expression ϕ, its set of variables $Var(\phi)$ is defined inductively as follows, for the various possible forms of ϕ:

- $\phi = x$: $Var(\phi) = \{x\}$
- $\phi = \overline{\phi_1}$: $Var(\phi) = Var(\phi_1)$
- $\phi = \phi_1 \wedge \phi_2$ or $\phi = \phi_1 \vee \phi_2$: $Var(\phi) = Var(\phi_1) \cup Var(\phi_2)$

$T \vDash \phi$ (read: "T *satisfies* ϕ") $\stackrel{\text{def}}{=}$

- ϕ is of the form $x : T(x) = TRUE$
- ϕ is of the form $\overline{\phi_1} : T(x) \nvDash \phi_1$
- ϕ is of the form $\phi_1 \wedge \phi_2 : T(x) \vDash \phi_1$ and $T(x) \vDash \phi_2$
- ϕ is of the form $\phi_1 \wedge \phi_2 : T(x) \vDash \phi_1$ or $T(x) \vDash \phi_2$

Definition 12.11. `SAT`: Given a boolean formula ϕ, is it satisfiable?

A formula is satisfiable if there exists a truth assignment to its variables that makes it `TRUE`.

There are $2^{|Var(\phi)|}$ possible truth assignments.

Given a truth assignment, we can just go through the formula and replace each occurrence of a variable x_i with the truth value $T(x_i)$. We then recursively evaluate each operator (of the forms $\overline{\phi}$, $\phi_1 \vee \phi_2$ and $\phi_1 \wedge \phi_2$.

Algorithm 12.10 Polynomial checking of a candidate for `SAT`

```
function CHECK_CANDIDATE(T):
    Replace each variable x of φ with the boolean value T(x).
    while φ is not of the form TRUE or FALSE do
        Find all subexpressions working on constants and evaluate them
    return what is left of the formula
```

Replacing the variables with their truth value can be done in time linear in $|\phi|$; the while loop finds expressions such as $\overline{\texttt{TRUE}}$, $\texttt{FALSE} \wedge \texttt{TRUE}$ and solves them (based on truth-table logic), replacing them with a boolean constant. At each step, the size of the remaining expression shrinks, so the algorithm will perform $O(|\phi|)$ such iterations.

In the end, a single constant will remain; if it is `TRUE` then the candidate truth assignment T satisfies ϕ, otherwise it doesn't.

12 CNF SAT

Definition 12.12. `CNF SAT`: Given a boolean formula ϕ, in *Conjunctive Normal Form*, is it satisfiable?

A formula is in *Conjunctive Normal Form* if it is a conjunction of *clauses*, with each clause being a disjunction of *literals*. A literal is either a variable, or a negated variable.

In other words, a boolean formula ϕ in CNF with m clauses is of the form:

$$\bigwedge_{i=1}^{m} (\bigvee_{j=1}^{k_i} L_j)$$

Where each clause i has k_i literals L_j and each L_j is either x or $\overline{x}$, for some variable x of ϕ.

The same solution described for `SAT` would also work in this case.

This is a more "restricted" version of `SAT` (the inputs have a special structure), but it is also believed to have no polynomial solution[§].

[§]More importantly, it can be shown that if `CNF SAT` has a polynomial-time solution, then so does `SAT`; and if `SAT` doesn't have a polynomial-time solution, neither does `CNF SAT`.

13 3SAT

Definition 12.13. 3SAT: Given a boolean formula ϕ in *Conjunctive Normal Form*, where each clause has exactly three literals, is it satisfiable?

We see here yet another iteration of the SAT problem with an extra constraint on the number of variables in a clause.

As before, the same algorithm for SAT also solves 3SAT, but does the fixed (and small) number of variables enable a more efficient solution?

The answer might be surprising: we can formally show that 3SAT is *"as hard"* as SAT[††]. This implies that if, as is widely believed, there is no polynomial solution for SAT, then there is no polynomial solution for 3SAT either.

However, constraining the problem even more brings us into the realm of polynomial certainty: the similar problem 2SAT has been proven to be in P!

14 References and further reading

Most of the problems presented here (or a variant of them) have been described in a very influential article by Richard Karp, published in 1972: *"Reducibility Among Combinatorial Problems"* [1]. The article identifies 21 practically-relevant combinatorics problems and shows that SAT could be reduced to any of them. Combined with a previous result by Stephen Cook, this means all these problem are "NP-complete"; the precise definition and importance of this concept will be addressed in the next lecture.

A polynomial time algorithm for the problem 2SAT can be found in the '79 article by Bengt Aspvall, Michael Plass and Robert Tarjan: *"A linear-time algorithm for testing the truth of certain quantified boolean formulas"* [2].

Bibliography

[1] Richard M Karp. *Reducibility among combinatorial problems*. New York: Plenum, 1972, pp. 85–103.

[2] Bengt Aspvall, Michael F Plass, and Robert Endre Tarjan. "A linear-time algorithm for testing the truth of certain quantified boolean formulas". In: *Information processing letters* 8.3 (1979), pp. 121–123.

[††]Two lectures from now, we will provide a reasonable formal definition for what it means for two problems to have the same "hardness", using *polynomial-time many-one reductions*.