Interval Computations Is NP-Hard

In this lecture, we provide an example of a problem which is NP-hard but may not be NP-complete:

- every problem from the class NP can be reduced to this problem, but
- this problem may *not* be in the class NP – since in this problem, once someone presents a candidate for a solution, there is no known feasible algorithm for checking whether this candidate is indeed a solution.

1 What Is Interval Computations Problem

Computers process measurement results. Computers were originally designed to process data, to process values of physical quantities – and these values come from measurement. For example, a computer system for a self-driving car relies on such information as the distances to different objects, the velocities of these objects, etc.; all these values come from measurements.

Measurement are never absolutely accurate. Measurements are always approximate. For example, when a person has height 170 cm, it does not mean 170.00000 cm, it means approximately 170 – i.e., in effect, anywhere from 169.5 to 170.5.

In general, the result $\widetilde{r}$ of the measurement is, in general, somewhat different from the actual (unknown) value r of the measured quantity. The difference $\Delta r = \widetilde{r} - r$ between the measurement result and the actual value is known as the *measurement error*.

Often, we only know the upper bound on the measurement error. Sometimes, we know the probabilities of different values of the measurement error. However, in many other cases, all we know is the upper bound on the measurement errors, i.e., the value $\Delta > 0$ for which $|\Delta r| \le \Delta$.

Enter interval uncertainty. For example, if the measured value is 1.0 and we know that the measurement error cannot exceed 0.1, this means that the actual value can be anywhere between $1 - 0.1 = 0.9$ and $1 + 0.1 = 1.1$.

In general, if the measurement value is $\widetilde{r}$, and we know that the measurement error cannot exceed Δ, this means that the actual value r can take any value between $\underline{r} = \widetilde{r} - \Delta$ and $\overline{r} = \widetilde{r} + \Delta$. In other words, the set of possible values of r is the *interval* $[\underline{r}, \overline{r}]$.

Interval computations. Processing data means computing the value of some quantity s which is related to the measured quantities $r_1, \ldots, r_n$ by a known dependence $s = f(r_1, \ldots, r_n)$.

An example of such a relation is Ohm's law, where the voltage V is equal to the product of the current I and the resistance R: $V = I \cdot R$. Here $r_1 = I$, $r_2 = R$, and $f(r_1, r_2) = r_1 \cdot r_2$.

In practice, we do not know the actual values, we only know the measurement results $\widetilde{r}_1, \ldots, \widetilde{r}_n$. So, all we can do is apply the algorithm $f(r_1, \ldots, r_n)$ to these measurement results and get the value $\widetilde{s} = f(\widetilde{r}_1, \ldots, \widetilde{r}_n)$.

As we have mentioned, the actual values r_i are, in general, different from the measurement results. Thus, the value $\widetilde{s}$ obtained by processing the measurement results is, in general, different from the desired actual value $s = f(r_1, \ldots, r_n)$. A natural question is: how different are they? What is the range $[\underline{s}, \overline{s}]$ of possible values of $s = f(r_1, \ldots, r_n)$ when each r_i is in the corresponding interval $[\underline{r}_i, \overline{r}_i]$?

The problem of computing this range is known as the problem of *interval computations*. Let us formulate this problem for the case when $f(r_1, \ldots, r_n)$ is a polynomial.

Interval computations problem.

- *given:* a polynomial $f(r_1, \ldots, r_n)$ with rational coefficients and n rational-valued intervals $[\underline{r}_i, \overline{r}_i]$;
- *find:* the endpoints $\underline{s}$ and $\overline{s}$ of the range

$$[\underline{s}, \overline{s}] = \{f(r_1, \ldots, r_n) : r_i \in [\underline{r}_i, \overline{r}_i] \text{ for all } i\}.$$

Sometimes, this problem is easy to solve. Something, this problem is easy. For example, if $f(r_1, r_2) = r_1 + r_2$, then:

- the smallest possible value of the sum if when each of the variables is the smallest, i.e., if $r_1 = \underline{r}_1$ and $r_2 = \underline{r}_2$;
- similarly, the largest possible value of the sum if when each of the variables is the largest, i.e., if $r_1 = \overline{r}_1$ and $r_2 = \overline{r}_2$.

Thus, in this case, $\underline{s} = \underline{r}_1 + \underline{r}_2$ and $\overline{s} = \overline{r}_1 + \overline{r}_2$.

However, in general, as we will show, the problem is complex – it is NP-hard.

How we will prove NP-hardness. We will prove NP-hardness the same way we proved NP-completeness of all the previous problems – by showing that 3-SAT can be reduced to this problem.

2 Reducing 3-SAT to Interval Computation: Algorithm U_1

We start with a 3-CNF formula. We start with a 3-CNF formula, i.e., with a formula of the type $C_1 \,\&\, \ldots \,\&\, C_k$, where each clause C_j either has a form

$a \vee b$ or the form $a \vee b \vee c$, and a, b, and c are literals, i.e., either the Boolean variables v_i or their negations $\neg v_i$.

Main idea. To each Boolean variable v_i that can take only two values 0 and 1, we put into correspondence a real-valued variable r_i that can take any value from the interval $[0, 1]$.

As we know from studying primitive recursive functions:

- negation can be described as $1 - r$,
- "and" $(r \,\&\, s)$ can be described as $r \cdot s$;
- due to de Morgan laws, "or" $(r \vee s)$ can be described as $\neg(\neg r \,\&\, \neg s)$, i.e., as $1 - (1 - r) \cdot (1 - s)$;
- similarly, $r \vee s \vee t$ can be described as

$$\neg(\neg r \,\&\, \neg s \,\&\, \neg t) = 1 - (1 - r) \cdot (1 - s) \cdot (1 - t).$$

Thus, we arrive at the following algorithm.

Algorithm. For each 3-CNF formula F with n Boolean variables $v_1, \ldots, v_n$, we construct the following polynomial $P(F)$ of n real-valued variables $r_i \in [0, 1]$:

- for literals, $P[v_i] = r_i$ and $P[\neg v_i] = 1 - r_i$;
- for a clause, $P[a \vee b] = 1 - (1 - P[a]) \cdot (1 - P[b])$ and

$$P[a \vee b \vee c] = 1 - (1 - P[a]) \cdot (1 - P[b]) \cdot (1 - P[c]);$$

- for the 3-CNF formula, $P[C_1 \,\&\, \ldots \,\&\, C_k] = P[C_1] \cdot \ldots \cdot P[C_k]$.

Example. For the formula $F = (v_1 \vee \neg v_2) \,\&\, (\neg v_1 \vee v_2 \vee \neg v_3)$, with $C_1 = v_1 \vee \neg v_2$ and $C_2 = \neg v_1 \vee v_2 \vee \neg v_3$, we have:

- $P[v_1] = r_1$, $P[\neg v_1] = 1 - r_1$, $P[v_2] = r_2$, $P[\neg v_2] = 1 - r_2$, $P[v_3] = r_3$, and $P[\neg v_3] = 1 - r_3$;
- $P[C_1] = P[v_1 \vee \neg v_2] = 1 - (1 - r_1) \cdot (1 - (1 - r_2)) = 1 - (1 - r_1) \cdot r_2$;
- $P[C_2] = P[\neg v_1 \vee v_2 \vee \neg v_3] = 1 - (1 - (1 - r_1)) \cdot (1 - r_2) \cdot (1 - (1 - r_3)) = 1 - r_1 \cdot (1 - r_2) \cdot r_3$;
- thus, $P[F] = (1 - (1 - r_1) \cdot r_2) \cdot (1 - r_1 \cdot (1 - r_2) \cdot r_3)$.

Comment. Please note that we did not actually multiply the polynomials

$$(1 - (1 - r_1) \cdot r_2) \text{ and } (1 - r_1 \cdot (1 - r_2) \cdot r_3)$$

term by term, as you would have done in high school, and there is a reason for this:

- we want feasible-time reduction, but
- if we multiply n polynomials, we may get exponentially many terms: e.g., the product
$$(x_1 + y_1) \cdot (x_2 + y_2) \cdot \ldots \cdot (x_n + y_n)$$
has 2^n terms.

What is the reduction: for the polynomial $P[F]$, the largest possible value $\overline{s}$ is equal to 1 if and only if the formula F is satisfiable.

Why this is a correct reduction. The product $P[F]$ of k numbers $P[C_j]$ which are smaller than or equal to 1 is equal to 1 only if all these numbers are equal to 1. So, if the polynomial attains the value 1, this means that $P[C_j] = 1$ for all j.

For $C_j = a \vee b$ we have $P[C_j] = 1 - (1 - P[a]) \cdot (1 - P[b])$, so $P[C_j] = 1$ if and only if $1 - (1 - P[a]) \cdot (1 - P[b]) = 1$, i.e., if and only if $(1 - P[a]) \cdot (1 - P[b]) = 0$. The product is 0 if and only if one of the factor is equal to 0, i.e., either $1 - P[a] = 0$ or $1 - P[b] = 0$:

- In the first case, $P[a] = 1$, i.e., a is true.
- In the second case, $P[b] = 1$, i.e., b is true.

In both cases, $a \vee b$ is true – i.e., the clause C_j is true.

Similarly, for the clause $C_j = a \vee b \vee c$, if $P[C_j] = 1$, this means that C_j is true.

So, all clauses are true – and thus, the formula is satisfiable.

Vice versa, if the formula F is satisfied by some values v_i, we can simply take $r_i = v_i$ and get $P[F] = 1$, thus the value $\overline{s}$ – which is the largest value of $P[F]$ – is also equal to 1.

Try the above algorithm on the formula $F = (\neg v_1 \vee v_2) \,\&\, (v_1 \vee \neg v_2 \vee v_3)$.