14. P VERSUS NP

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

November 2024

We have laid the foundations necessary to introduce one of the most important open problems in computer science: *"the* **P** *versus* **NP** *problem"*.

This problem is interesting not merely in itself: in the quest of addressing it, the field of complexity theory was greatly developed, with many interesting results being discovered, as well as many more interesting conjectures formulated.

1 Nondeterministic computation

The computational models studied so far are **deterministic**: at any point of computation, knowing the configuration of the system (together with the "yield" rules) allows us to precisely determine the next configuration.

But let's explore what happens if we relax this requirement and allow for **nondeterministic** models, where the next configuration is not uniquely determined.

The concept might seem a bit artificial and in some way it is (more on this later). It lies at the foundation of the definition of the class of problems solvable in *Nondeterministic Polynomial time* – **NP**. However, there is another definition of **NP** which is perhaps more intuitive: involving *polynomial-sized certificates*. We will focus mostly on this latter one, but we think it's important to understood the roots of this concept: the nondeterministic perspective.

1.1 The Nondeterministic Turing Machine

Definition 14.1. A Nondeterministic Turing Machine is a 7-tuple: $(Q, \Sigma, \Gamma, B, q_1, \delta_0, \delta_1)$, where:

- The first five members: $Q, \Sigma, \Gamma, B, q_1$ – have the same significance as for a deterministic Turing Machine.
- There are two transition functions (δ_0 and δ_1), each with the same significance as for the deterministic machine.
- At each step of computation, the machine *simultaneously* applies both transition functions, *branching* into two new configurations.

If, for example, $\delta_0(q_1, 0) = (q_2, 1, \rightarrow)$ and $\delta_1(q_1, 0) = (q_3, 0, \leftarrow)$, then the NTM, when in state q_1 and reading symbol 0, does **both** of these things:

- transitions to state q_2, writes a 1 on the tape and moves the head one cell to the right
- transitions to state q_3, writes a 0 on the tape and moves the head one cell to the left

You can envision that the machine's execution "splits into two alternate realities".

The computation history of an NTM is no longer a sequence of configurations, but rather a tree. Any path from the root to a leaf[†] is "**a** computation" of the NTM.

The leaves are represented by a configuration containing a final state, one of Y, N, H. But each leaf could end up in a different final state, so what does it mean to compute an answer? Let's stick solely to decision problems:

Definition 14.2. For an NTM M and an input w:

- $M[w] \rightarrow$ `TRUE` if **any** of the nondeterministic computations ends up in state Y.
- $M[w] \rightarrow$ `FALSE` if **all** of the nondeterministic computations ends up in state N.

[†]Remember that, at this point, we're only interested in computations that always halt.

You can think of an NTM machine as simultaneously trying multiple ways in which to accept its input; if one of them succeeds, the input is accepted; if all fail, the input is rejected.

Notice that there is an asymmetry in the definition of acceptance: we need just one successful path in order to accept a word; but to reject it, all paths must fail.

Let us now consider the *time complexity* of such a machine. For a deterministic Turing machine, the number of transitions on an input is simply the number of configurations in its computational history.

But now we are dealing with a tree, which does not have a length; we consider the length of *the longest path from the root to a leaf.* Thus $t_N : \Sigma^* \to \mathbb{N}$, maps an input to the number of configurations in the longest computation of N.

Having adequately adapted t_N, we can define the worst-case running time of such a machine in an identical fashion to deterministic machines:

Definition 14.3. A Nondeterministic Turing Machine N's *running time* is a function $T_M : \mathbb{N} \to \mathbb{N}$, where:

$$T_N(n) \overset{\text{def}}{=} \max_{w \in \Sigma^n} (t_N(w))$$

From a computability perspective, the NTM does not offer us any new capabilities; it can't solve more problems. But from a complexity point of view, things might be different...

1.2 Deterministic simulation of an NTM

Before we formulate a theorem, let us address one more detail about the computational history tree of an NTM. On each transition, an NTM has one[‡] or two *"choices"* – the different actions that it can take based on the current (state, symbol) pair; the number of choices in a particular configuration is equal to the number of children that node has in the computational history tree. Then its computational tree has at most $2^{T(n)-1}$ nodes. We *order* the children of a node, such that the one yielded by δ_0 comes first.

It's easy to see that we can simulate the entire run of an NTM by backtracking through all the nodes of its computational tree[§].

Theorem 14.1. For any Nondeterministic Turing Machine M_N that computes a problem f in time $T(n) \geq n$, there exists a 3-tape deterministic Turing Machine M_D that computes f in $2^{O(T(n))}$ time.

(We restrict our theorem to only address NTMs which consume more than linear time; machines that run nondeterministically in constant time are uninteresting and their deterministic counterparts would also run in constant time.)

Proof. The tapes of M_D have the following roles:

- input tape, treated as read-only
- the contents of the simulated tape
- a tape with the *"address"* of the currently considered configuration

To represent the "address" of a configuration we will use the binary alphabet: $\Gamma_2 = \{0, 1\}$. The initial configuration (the root of the computation tree) gets assigned the empty string: ε.

Then each node can be described by the path from the root: 001 is the second child of the first child of the first child of the root.

Alternatively, think of every 0 to mean that we follow the transition described of δ_0 and every 1 to mean that we follow the transition of δ_1.

To simulate a single step of M_N on a particular configuration C, M_D does the following:

1. copy the input onto the second tape

[‡]If both transition functions have the same value on the current (state, symbol) pair.

[§]Which shows that, from a computability perspective, nondeterminism doesn't open up a larger class of problems; we can solve exactly what a deterministic machine can solve.

2. if the current configuration is an accepting one, transition to Y

3. if the current configuration is a rejecting one, go to step 5

4. use the contents of the second tape to simulate M_N's actions on one nondeterministic path.

   For each symbol i from the third tape, choose the action of δ_i from M_N's two transition functions; if the current node doesn't have an i^{th} child, abort the current path and go to the next step.

5. replace the contents of the third tape with the next string in lexicographic order; go to step 1

Initially, the third tape corresponds to the encoding of the root, ε.

For each (state, symbol) pair, the "knowledge" of the possible actions of M_N are hardcoded into M_D.

M_D basically does a *Breadth First Search* exploration of the computational tree of M_N; but for each new node, it has to start the entire simulation from the beginning.

In the first step, there is a single configuration of M_N; after that, at most 2 new configurations can be explored, which lead to at most 4 configurations, which lead to at most 8 configurations and so on.

There are at most $2^{T(n)+1} - 1$ computational paths (string configurations for the third tape). For each of these strings, the deterministic simulation does $O(T(n))$ operations (including "incrementing" the string on the third tape and copying the input string onto the second tape). What is interesting to us is the $2^{T(n)}$ part.

$\square$

M_D goes through all the nodes in the computational tree of M_N. If M_N has one possible choice per node, then it acts as a deterministic Turing Machine – an uninteresting case.

If M_N has two choices per node, then its computational tree has on the order of $2^{T(n)}$ nodes, which have to be explored by M_D. Notice the *exponential slowdown*.

Can we do better? Can we come up with some cleverer way to deterministically simulate an NTM with only a polynomial slowdown?

We can improve some aspects of our simulation, but whether a polynomial time simulation exists is **unknown**. Currently, no such simulation has been found; but neither has it been proven that such a simulation cannot exist.

1.3 Nondeterministic pseudocode

An explicit table for an NTM would have every cell contain a list of possible actions. Even for deterministic machines, where we only had a single (state, symbol, direction) tuple in each cell, it was cumbersome to write these tables and we'll try to avoid it entirely for NTMs.

Instead, we will rely on the intuition we gained on mentally relating pseudocode to Turing Machines. Our nondeterministic pseudocode will contain a special construct: **choice**; this splits the execution into two different branches: on one branches it yields the value 0, on the other it yields 1.

Here is a nondeterministic polynomial-time solution to `CLIQUE`:

Algorithm 14.1 Nondeterministic polynomial solution for `CLIQUE`

```
function SOLVE_CLIQUE($G = (V, E), K$):
    $U \leftarrow \emptyset$
    for each $i = \overline{1, K}$ do
        for each $v \in V$ do
            $select \leftarrow$ choice
            if $select = 1$ then
                $U \leftarrow U \cup \{v\}$
    return CHECK_CANDIDATE($U, E$)

function CHECK_CANDIDATE($U$):
    $result \leftarrow$ TRUE
    for each $u \in U$ do
        for each $v \in U$ do
            if $u \neq v \wedge (u, v) \notin E$ then
                $result \leftarrow$ FALSE
    return $result$
```

Note that we employ the "deterministic polynomial-time candidate checking" routine from the previous lecture; nondeterminism's only role is to generate all possible candidates (subsets of K nodes) *fast*.

Using **choice** to create candidate subsets is a common element of nondeterministic pseudocode, so we will enhance our primitive to be a function whose argument is a set S and which splits the execution into $|S|$ branches, on each of them yielding a different element of S.

We can rewrite the `SOLVE_CLIQUE` more succinctly:

Algorithm 14.2 Using **choice** as a function

```
function SOLVE_CLIQUE($G = (V, E), K$):
    $U \leftarrow \emptyset$
    for each $i = \overline{1, K}$ do
        $U \leftarrow U \cup \{$choice$(V)\}$
     return CHECK_CANDIDATE($U, E$)
```

There are many "useless" branches being created this way; for example, after the first choice, on the branch on which U became $\{1\}$, one of the branches spawned by the second choice will have $U = \{1, 1\}$, a candidate with the same node twice. This is nonsensical, but under the convention that there are no "self-edges" (i.e. $\forall v \in V, (v, v) \notin E$), such sets will be filtered out by the `CHECK_CANDIDATE` function.

The number of candidates considered is $|V|!$ instead of $\frac{|V|!}{(|V|-K)!K!}$, but it doesn't matter much for us. We could fix this by choosing from $V \setminus U$ instead of V.

2 The class NP

Definition 14.4.

$$\textbf{NTIME}(T(n)) \overset{\text{def}}{=} \{f : \Sigma^* \rightarrow \{\texttt{FALSE}, \texttt{TRUE}\} \mid f \text{ is decided by some NTM in time } T(n) \}$$

In analogy with **P**, we can now define **NP**:

Definition 14.5.

$$\textbf{NP} \overset{\text{def}}{=} \bigcup_{k \in \mathbb{N}} \textbf{NTIME}(n^k)$$

P is the class of all problems decidable in polynomial time by a deterministic Turing Machine, while **NP** is the class of all problems decidable in polynomial time by a nondeterministic Turing Machine.

3 Certificates

We can provide a different, more intuitive, definition of **NP**.

Definition 14.6. Let $f : \Sigma^* \to \{\texttt{FALSE}, \texttt{TRUE}\}$ be a decision problem.

We say that f is *"polynomially verifiable"* if there exists a Turing machine M which for all words w that have $f(w) = \texttt{TRUE}$:

$$\exists k \in \mathbb{N}, \exists c \in \Sigma^* \text{ s.t. } \begin{cases} M[enc((w,c))] \to \texttt{TRUE} \text{ in } \textit{polynomial time} \\ |c| = |w|^k \end{cases}$$

We call c a *"certificate"* (or "witness"); it is a succinct piece of additional information about w, which lets M quickly determine the answer to $f(w)$.

Of course, by "succinct" we mean that the size of the certificate is polynomial in the size of the input; by "quickly" we mean time polynomial in the size of the input.

3.1 Certificate for CLIQUE

For example, consider the problem CLIQUE. An input is the encoding of a (G, K) pair, where G is a graph and K is the clique-size we are searching for. As mentioned before, it is unknown whether there exists a polynomial-time algorithm that can check whether G has a clique of size K.

But consider the following certificate: a set C of K nodes that form a clique. As K cannot be larger than $|V|$, this is certainly polynomial in the size of the input. The machine only needs to iterate through all K edges in the clique-candidate and check that each of them is connected to all the others; that each pair $(u, v) \in E$ (where $u, v \in C$). Without a need to be precise, we can see that this runs in time $O(|V|^4)$, which is polynomial in the size of the input[††].

In fact, we've already written a procedure for validating a CLIQUE certificate: CHECK_CANDIDATE.

Notice that the asymmetry we pointed out for nondeterminism is also present in this concept: if the answer is TRUE, then there exists such a polynomial-length certificate; but if the answer is FALSE there is no such guarantee[‡‡].

This asymmetry hints at a redefinition of **NP**:

Definition 14.7. $\mathbf{NP} \overset{\text{def}}{=} \{f : \Sigma^* \to \{FALSE, TRUE\} \mid f \text{ is polynomially verifiable } \}$

4 Interpretation of NP

But what is **NP**? The first definition using nondeterminism is admittedly artificial and makes it hard to see any practical relevance of this class of problems.

But the certificate definition should shed some light on the subject. There are many problems that require finding or designing some object: paths, covers, cliques, routes etc.

These objects *are* the certificates in question; and they correspond to real-life physical objects or plans, thus they should have a "reasonable" size.

A certificate of exponential size, for example, would be too large to be of any practical use.

5 $\mathbf{P} \subseteq \mathbf{NP}$

But are **P** and **NP** really different sets of decision problems? They certainly have different *definitions*, but it may be the case that these definitions happen to name the exact same classes.

First, we can point out that every Deterministic Turing Machine can be straight-forwardly reinterpreted as a Non-deterministic Turing Machine which never makes any choice. From this we can conclude that any problem solvable in *deterministic polynomial time* is also solvable in *nondeterministic polynomial time*. In other words, $\mathbf{P} \subseteq \mathbf{NP}$.

[††] $K \le |V|$, so we need to check $O(|V|^2)$ pairs of nodes; to lookup (u, v) in E on a single-tape TM, we have to go through the entries in the adjacency matrix, one by one; so each lookup can be done in $O(|V|^2)$.

[‡‡] If $P = NP$, then there also exists a polynomial-length certificate for FALSE answers; if $P \neq NP$, then there doesn't.

But can we prove the reverse, that $\mathbf{NP} \subseteq \mathbf{P}$? That all problems solvable in nondeterministic polynomial time are also solvable in deterministic polynomial time?

Or maybe that is not true, so can we prove $\mathbf{P} \neq \mathbf{NP}$?

This is an open problem; no one has been able to prove a result either way.

You will see in a future lecture that there are many other open problems that are in some sense a "rephrasing" of the $\mathbf{P} \overset{?}{=} \mathbf{NP}$ question. For example: does there exist a deterministic polynomial time algorithm that solves any of the problems presented in the previous lecture? That is also unknown...

6 References and further reading

The definition of the Nondeterministic Turing Machine as presented in subsection 1.1 is adapted from Arora and Barak's *"Computational Complexity: A Modern Approach"* [1], section 2.1.2.

The deterministic simulation presented in subsection 1.2 is adapted from [2], Chapter 3, page 178.

The notion of nondeterministic algorithms (and of the **choice** primitive used in subsection 1.3) was introduced by Robert W. Floyd in his 1967 paper suggestively entitled *"Nondeterministic Algorithms"* [3]. The purpose of such algorithms is to work as "conceptual devices to simplify the design of backtracking algorithms by allowing considerations of program bookkeeping required for backtracking to be ignored".

The Clay Mathematics Institute famously compiled at the beginning of the 3rd millennium a list of seven important open problems in mathematics, among which "**P** versus **NP**"[§§]. A solution to any of these would be awarded with a million dollar prize (and most likely a Turing award or a Fields medal). As of 2024, only one of them (the Poincaré Conjecture) has been solved.

A good survey of the state of the problem, can be found in a 2017 survey written by Scott Aaronson [4].

Aaronson is of the firm opinion that the problem is solvable, and the answer is that $\mathbf{P} \neq \mathbf{NP}$. His arguments are presented in the survey and summarized more succinctly on his personal blog[‖].

For a contrasting opinion, you can read Donald Knuth's opinions. For example, his answer to question 17 in this interview[**].

William Gasarch has conducted three opinion polls (in 2002 [5], 2012 [6] and 2019 [7]) on the subject, in which he asked around 100 computer scientists several questions about what they think the answer is, when it will be discovered, by what method etc. The numbers vary with each poll, but a big majority seems to lean towards $\mathbf{P} \neq \mathbf{NP}$.

Bibliography

[1] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach.* Cambridge University Press, 2009.

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

[3] Robert W. Floyd. "Nondeterministic Algorithms". In: *J. ACM* 14.4 (Aug. 1967), pp. 636–644. URL: `https://doi.org/10.1145/321420.321422`.

[4] Scott Aaronson. "$P \overset{?}{=} NP$". In: *Open problems in mathematics* (2016), pp. 1–122.

[5] William I Gasarch. "The P=? NP poll". In: *Sigact News* 33 (2 2002), pp. 34–47.

[6] William I Gasarch. "Guest column: The second P=? NP poll". In: *ACM SIGACT News* 43 (2 2012), pp. 53–77.

[7] William I Gasarch. "Guest column: The third P=? NP poll". In: *ACM SIGACT News* 50 (1 2019), pp. 38–59.

[§§] `https://www.claymath.org/millennium-problems/`

[‖] `https://scottaaronson.blogspot.com/2006/09/reasons-to-believe.html`

[**] `https://www.informit.com/articles/article.aspx?p=2213858`