CSCI3390-Lecture 12: The class **NP**

1 Example: Sudoku

The pair of grids in the figure below (which you have seen before) is an instance of the following decision problem:

*Input:* A partially filled-in Sudoku grid, and a completed Sudoku grid.

*Output:* Yes if the completed grid is a solution to the puzzle in the partially filled-in grid, no otherwise.

As we discussed earlier, verifying that the second grid is filled in legally takes polynomial time in the size of the grid. We also require only polynomial time to verify that the two grids agree on the filled cells of the first grid. Thus this problem is in \( P \).

The grid on the left is an instance of this problem:


This problem is believed to be outside of \( P \). As we’ve seen, there is a simple guess-and-verify ‘algorithm’ for solving this problem: guess a solution, and then use the (reasonably fast) algorithm for the previous problem to verify that your guess is indeed a solution. But this does not really solve the problem; the only way we can be sure that we have not missed a solution is to try out all possible guesses.

2 NP Problems

Decision problems with this kind of structure are called ‘NP problems’. The requirement is that they can be solved by a guess-and-verify algorithm with these properties:
Figure 1: The pair of grids is an instance of the polynomial-time verifier problem for Sudoku

- The size of the guessed solution is polynomial in the size of the problem instance. That is, there is a positive integer $k$ such that every guess has length $O(n^k)$ for inputs of length $n$.

- The verification algorithm runs in time polynomial in the length of the input.

Observe that in our Sudoku example, the size of the guessed solution is the same as the size of the input—assuming that we are encoding the problem instances by grids of size $N^4$ with 0s in the empty cells. So the first requirement is satisfied as well as the first.

What follows is a more formal definition. It takes a bit of gymnastics to write it down completely correctly. To properly understand the concept you should think about the Sudoku example, and several other ones that will follow.

A language $L \subseteq \Sigma^*$ is in NP if there exist a positive integer $k$, and a language $L' \subseteq \Sigma^*$, such that

- $L' \in \mathbf{P}$

- $w \in L$ if and only if there exists $v \in \Sigma^*$ with $|v| \leq |w|^k$, and $wv \in L'$.

In this definition $v$ is the guessed solution, and $L'$ is the problem of determining whether $v$ is a solution to $w$.

Let’s observe a few things about this definition. First of all $\mathbf{P} \subseteq \mathbf{NP}$, because if $L \in \mathbf{P}$, we can just take $L' = L$ and $v$ to be the empty string, so $L$ satisfies
the definition of languages in NP. Second, every language in NP is decidable by the brute-force algorithm: Given an input \( w \) of length \( n \), generate every string \( v \) of length no more than \( n^k \), and test if \( wv \in L' \). This requires \( |\Sigma|^n \) \( k \) calls to the polynomial-time verifier algorithm.

3 Example: Hamiltonian path

We’ve discussed this problem before. There are a number of different variations, including the Traveling Salesman Problem, which asks for the shortest path in a graph that visits every vertex. The version we will use is this:

**Input:** A graph \( G \) and two vertices \( u \) and \( v \).

**Output:** Yes if there is a path from \( u \) to \( v \) that contains every vertex of \( G \) exactly once. No otherwise.

To see why this is in NP, consider first the corresponding verification problem:

**Input:** A graph \( G \), two vertices \( u \) and \( v \), and a sequence \( S \) of \( n \) vertices, where \( n \) is the number of vertices of \( G \).

**Output:** Yes if \( S \) is a path from \( u \) to \( v \) that contains every vertex of \( G \) exactly once. No otherwise.

Intuitively, it is ‘easy’ to verify if a proposed path is a Hamiltonian path, but let’s verify carefully that this latter problem is in P. The following algorithm gives a solution to the verification problem. It refers to a checklist of the vertices of \( G \).

```
if the first element of S is not u, halt and reject
for each vertex t in S:
    if t is checked, halt and reject
    check t off in the checklist
    let t’ be the vertex in S following t
    if G contains no edge between t and t’, halt and reject
if v is not the last element of S, halt and reject
halt and accept
```

Since \( S \) has \( n \) elements, if we do not reject because a vertex is checked twice, every vertex will have been checked exactly once. Then the for loop will not be executed more than \( n \) times. Each time the body of the loop is executed, the algorithm must find the adjacency list for \( t \) and then traverse this list to see if \( t' \) is present. This can be done in \( O(n) \) steps, so we use \( O(n^2) \) steps in all for the
Thus verifying that a path is Hamiltonian is in \textbf{P}. To see that the existence of a Hamiltonian path is in \textbf{NP}, we only have to note that the size of a guess is polynomial in the size of the input. But here a guess is just the list \( S \) of vertices, which is necessarily smaller than the encoding of \( G \), which contains information about both vertices and edges.

A brute-force algorithm will generate and check every possible sequence of the \( n \) vertices, of which there are \( n^n \). A smarter method will generate every possible permutation of the \( n - 2 \) vertices, excluding \( u \) and \( v \). There are \((n - 2)!\) such permutations, so this is not really much better: \((n - 2)! > 2^n\) for all sufficiently large \( n \), so this is at least exponential running time.

Our question, for all such problems in \textbf{NP}, is whether there is a polynomial time algorithm for this problem. But the decision problem is not really the problem that we want to solve—we would really like an algorithm to \textit{find} a Hamiltonian path, if one exists. In fact, \textit{if} we had an efficient solution to the decision problem, we would be able to find the path in polynomial time as well. Here is the reason:

Suppose, contrary to what we believe, that the Hamiltonian path problem, with specified start and end vertices \( u \) and \( v \), can be solved in time \( c \cdot n^k \), where \( n \) is the number of vertices in the graph. We apply this algorithm, and if it says no path exists, we stop. Otherwise, \( u \) will be the first vertex in a Hamiltonian path. Let \( u_1, \ldots, u_k \) be the vertices adjacent to the start vertex \( u \). Now remove \( u \) and all these \( k \) edges from the graph to get a graph \( G' \), and test if \( G' \) has a Hamiltonian path from \( u_1 \) to \( v \), from \( u_2 \) to \( v \), etc. This will take time no more than

\[
k \cdot (n - 1)^k < n^{k+1}.
\]

One these must answer yes, and this gives us our next vertex for the Hamiltonian path. We repeat this \((n - 1) \) times, and we thus construct the Hamiltonian path in \( O(n^{k+2}) \) steps.

\section{Rush Hour: A Non-example}

Consider the problem of determining if a Rush Hour puzzle configuration has a solution. (As usual, we have to imagine a generalized version of the puzzle played on grids of arbitrarily large size, but the problem still makes sense.) If we guess a solution, we can verify that the solution is correct in time \textit{polynomial in the length of the solution}. But solutions to these sliding block puzzles can be very long, and
may not be bounded by a polynomial in the length of the input. Thus we cannot say that this problem is in \( \text{NP} \), and in fact there is reason to believe that it is not.

5 Nondeterministic Turing machines

\( \text{NP} \) stands for ‘nondeterministic polynomial time’. This refers to another, equivalent way of defining this complexity class.

Suppose you took a list of quintuples

\[(q, \sigma, q', \gamma, D)\]

that we ordinarily associate with the transitions of a Turing machine. Does every such list actually define a TM? The answer is no, because it might contain two different quintuples with the same first two components \((q, \sigma)\), and thus the action of the machine when the current state is \( q \) and the currently scanned tape symbol is \( \sigma \) would not be determined.

In a nondeterministic Turing machine (NDTM), such duplication is allowed. Formally, the transition function is

\[
\delta : (Q - \{\text{accept, reject}\} \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})).
\]

If the machine is in a particular configuration, there may be several configurations that can follow it. Thus the computation of the machine from a given starting position is not a sequence of configurations, but a tree of configurations.

It is like a programming language with an ‘either-or’ statement: each time this is encountered, the program can decide which branch to follow.

We say that the NDTM accepts its input, if there is some path from the root of the tree—that is, some sequence of guesses—that leads to the accept state.

A polynomial-time NDTM has the additional property that for some integer \( k > 0 \), every path from the root leads to either acceptance or rejection in \( O(n^k) \) steps, where \( n \) is the length of the original input.

Here is another, equivalent, model of a polynomial-time NDTM: At the start of the computation, on input \( w \), the machine goes through a guessing phase, during which it writes \( |w|^k \) additional symbols on the tape, changing the tape contents to \( wv \). After that, the machine computes deterministically in polynomial time, like an ordinary polynomial-time Turing machine. The machine accepts \( w \) if there is some word \( v \) that leads to acceptance in the deterministic mode. This is just our ‘guess-and-verify’ process that we described originally. This is equivalent to the
other model of NDTM—we are just pre-loading all of our guesses at the beginning
of the computation. We will skip the formal proof that these two versions of
NDTM are equivalent, and stick to this second model.

So another, equivalent, definition of \( \text{NP} \) is: \( L \in \text{NP} \) if and only if there is a
polynomial-time NDTM that accepts exactly the strings in \( L \).

6 A picture of the world of computational problems

We have classified decision problems (=languages) according to their computa-
tional difficulty. We know that every problem in \( \text{P} \) is in \( \text{NP} \), that every problem
in \( \text{NP} \) is decidable and that every decidable problem is Turing-recognizable. So
our picture looks like Figure 2.

But are these inclusions strict? That is, are any of the colored regions actually
empty? We know that the Turing-recognizable languages do not constitute all
languages. (Because, for instance, the language

\[
\{ < \mathcal{M}, w > : \mathcal{M} \text{does not accept } w \}
\]

is not Turing-recognizable.) And the language

\[
\{ < \mathcal{M}, w > : \mathcal{M} \text{accepts } w \}
\]
is a Turing-recognizable language that is not decidable. It is a little harder to separate $\text{NP}$ from the decidable languages, but it is known that this inclusion is also strict.

What about the inclusion $\text{P} \subseteq \text{NP}$? The question of whether this inclusion is strict or not is still open. For all we know, the world might look like Figure 3.

This is the $\text{P} \not= \text{NP}$ problem. The common belief is that the inclusion is strict. If $\text{P} = \text{NP}$, many strange things would follow (including the collapse of cryptographic systems), but the question remains one of the great unsolved problems of mathematics.

Figure 3: ..or is this?