# CSCI3390-Lecture 14: The class **NP**

## 1 Problems and Witnesses

All of the decision problems described below have the form: 'Is there a solution to $X$?' where $X$ is the given problem instance. If the instance is a 'Yes' instance of the problem, then a solution is called a 'witness' to this fact.

### 1.1 Example: Sudoku

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

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

> *Input:* A partially filled-in Sudoku grid. *Output:* Yes if a solution exists, no otherwise.

It happens that the grid on the left is a 'Yes' instance of this problem, and the grid on the right is a witness to this fact.

### 1.2 Hamiltonian circuit

This problem is

> *Input:* A graph. *Output:* Yes if a there is a path, starting and ending at the same vertex, that passes through every vertex of the graph exactly once. No otherwise.

An instance of this problem is shown in Figure 2.

This is a 'Yes' instance, and a witness to this is the sequence of vertices $1, 2, 5, 6, 4, 3, 1$. Observe that there are several different witnesses possible. For this graph, there is, in a sense, only one Hamiltonian circuit, but we can choose any vertex for the start and end point, and make the tour in either of two directions.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

Figure 1: The pair of grids is an instance of the polynomial-time verifier problem for Sudoku

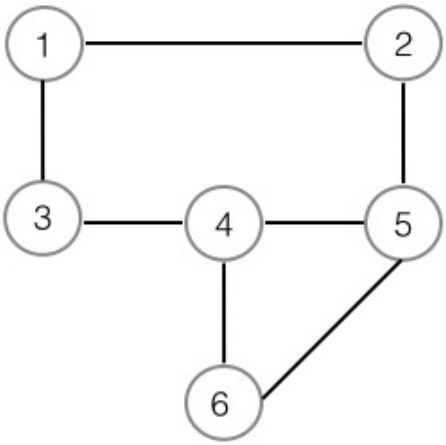


Figure 2: Does this graph have a Hamiltonian circuit?

## 1.3 Graph $k$-coloring

Each value of $k$ gives a different problem:

> *Input:* A graph. *Output:* Yes if a there a way to assign a color to each of the vertices of the graph, so that $k$ or fewer colors are used, and so that adjacent vertices are not assigned the same color. No otherwise.

Consider again the graph in Figure 1. If $k = 2$ this is a 'No' instance, but if $k = 3$, the assignment:

$$1 : \text{green}, 2 : \text{blue}, 3 : \text{blue}, 4 : \text{green}, 5 : \text{red}, 6 : \text{blue}$$

is a witness, representing a legal coloring. Observe there are many different ways to legally color this graph with three colors.

## 1.4 Compositeness

> *Input:* An integer $n$ given in binary. *Output:* Yes if there is an integer $m < n$ such that $m|n$. No otherwise.

A witness is a pair of integers $m, k$ such that $n = mk$ and $m < n$.

## 1.5 Rush Hour

A problem we've seen before:

> *Input:* A setup of cars on an $N \times N$ Rush Hour grid. *Output:* Yes if there is a sequence of moves that gets the red car out of the grid. No otherwise.

Figures 3 and 4 below show a particularly difficult 'Yes' instance and its solution on a $6 \times 6$ board.

## 1.6 Theoremhood

We've seen this before.

> *Input:* A sentence $\phi$ of arithmetic. *Output:* Yes if $\phi$ is a theorem. No otherwise.

A witness to a 'Yes' instance, of course, is a proof of $\phi$.


Figure 3: Can this Rush Hour puzzle be solved?

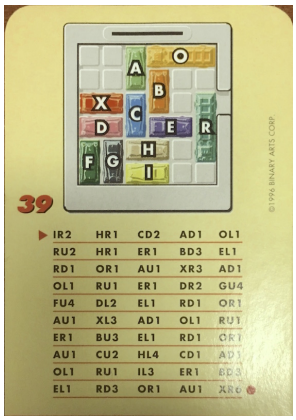


Figure 4: An optimal 50-move solution to the puzzle in the preceding figure. The symbols R,L,U,D mean ‘right’, ‘’left’ ‘up’ and ‘down’, and each instruction includes the number of cells to slide the image.

# 2 NP Problems

All six problems described above share a common feature concerning the ease of verifying a witness, and the first four share another feature concerning the size of witnesses. The two features together define the class of decision problems called $NP$.

## 2.1 Easy verification....

As we saw in the last lecture, if I give you the original Sudoku puzzle and the proposed solution, then the algorithm for verifying that it is indeed a solution runs in time polynomial in the size of the problem. This reflects the observed fact that you can verify at a glance that a Sudoku grid has been filled in correctly.

Much the same observation holds for the Hamiltonian circuit problem. If I give you a specification of the graph and a list of vertices, you can verify it as follows:

```
create a checklist of all the vertices in the graph
check off the first vertex s in the list
for each subsequent vertex v in the list:
     if v is not connected by an edge to the preceding vertex, reject
     else:
          check v off
          if v is already checked off:
                if v==s and all vertices have been checked off:
                    accept
                else:
                    reject
reject
```

The number of passes through the loop is no more than the length of the list, which in turn must be equal to the number $n$ of vertices in the graph. Each pass requires checking the graph representation to see if a pair of vertices is an edge, and locating a vertex in the check list. The exact step count depends on how we represent the graph, but in any case there will require $O(n)$ steps. This more detailed analysis again reflects a simple observation: If I give you the list of vertices and the graph, it is an easy matter to scan through the list and verify that it is a Hamiltonian circuit.

An identical observation holds for the $k$-coloring problem. For a relatively small graph, you can spot adjacent vertices with the same color at a glance. A careful algorithm takes time $O(n^2)$ on a graph with $n$ vertices.

For compositeness, we only need to multiply together the two numbers $m, k$ in the witness. Observe that the size of the witness is the number of bits in the representation of the two numbers (and not the values $m$ and $k$ themselves). Multiplication, done the usual way, takes time proportional to the square of the number of bits.

For Rush Hour, once again we have a straightforward algorithm for verifying the solution—we can just take that card with the solutions moves written on it, grab our Rush Hour game, and move the cars around, following the script, and checking that the red car does indeed exit the grid. A computer implementation requires time *linear* in the length of the script, although this again might depend on implementation details (but see below!).

To verify a proof, we need to check that each line is either an axiom, or follows from a previous line or lines by a rule of inference. The exact details depend on the axiomatic system, but typically this will be $O(n^2)$ where $n$ is the *length of the proof*, reflecting the intuition that proofs ought to be easy to verify.

## 2.2 Short witnesses...

While all six problems discussed above share the 'easy to verify' property, there is a stark difference between the first four examples and the last two: In the first four cases, the size of the witness was itself comparable to the size of the input. On the other hand, in sliding-block puzzles like Rush Hour, it is possible that the length of a solution is exponential in the size of the board. For theoremhood, we can't even guarantee that the length of a proof of $\phi$ is bounded by an exponential function, or any computable function, in the length of the sentence: if this were the case, we could in principle survey all possible proofs up to this bound, and decide whether a sentence is a theorem. But we have already seen that this is an undecidable problem.

So we are interested in decision problems that can be settled by verifying a witness, subject to these constraints.

- The size of the witness 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 witness.

The two properties together imply that the verification algorithm also runs in time polynomial in the length of the input.

This class of decision problems is called **NP**.

### 2.3 Formal definition of NP

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 examples above

A language $\mathbf{L} \subseteq \Sigma^*$ 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 $w\#v \in L'$.

In this definition $v$ is the witness, 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 $w\#v \in L'$. This requires $|\Sigma|^{n^K}$ calls to the polynomial-time verifier algorithm.

## 3 What does the 'N' stand for?

**NP** stands for 'nondeterministic polynomial time'. This refers to another, equivalent way of defining this complexity class.

### 3.1 Nondeterministic computation

Imagine a conventional procedural programming language, supplemented with a kind of fantasy control structure: an `either...or` statement:

```
either:
      <sequence S of statments>
or:
      <sequence T of statements>
```

Conventional programs compute *deterministically*: Run the same program twice on the same inputs, and it will do exactly the same thing. The `either...or` statement causes the program to compute *nondeterministically*. The same program run twice on the same inputs will in general give different results, depending on which branch of code it decides to execute.

Here is an example of a function written in this fantasy language that determines whether a graph $G$ is 3-colorable. The function calls the easy verification algorithm described earlier as a subroutine:

```
def three_colorable(G):
    for each vertex v in G:
        either:
            color v red
        or:
            color v green
        or:
            color v blue
        #verify the coloring:
        if the coloring is legal:
            return True
        else:
            return False
```

Each run of the function requires time polynomial in the size of the graph—we have already observed that this is true for the verification algorithm called at the end, and the guessing phase at the beginning takes time proportional to the number of vertices in the graph. What is different is the way in which this peculiar program 'solves' the problem of determining 3-colorability: If there is *some* sequence of choices that leads to the function returning `True`, then the graph is 3-colorable. To turn this into a normal deterministic computation, we would have to try out all $3^n$ possible choices, where $n$ is the number of vertices.

So here is an alternative, equivalent, definition of **NP**: It consists of those decision problems solved by a boolean-valued `either-or` function that runs in polynomial time on every input.

## 3.2 A Turing machine version...

In order to prove things about **NP** problems, we will require a more concrete model. Once again, we turn to Turing machines:

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

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

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 exactly ike the 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.

## 3.3 ...and another.

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 $w\#v$. 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. It's not too hard to show that this is equivalent to the model we described above—what we are doing is

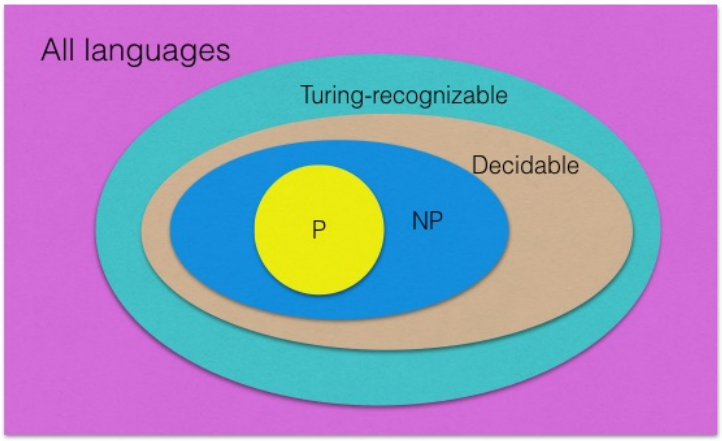


Figure 5: Is this the right picture of the language classes we've studied?

pre-loading all of our guesses at the outset, instead of making them whenever a decision is required.

In either version of the NDTM, another equivalent definition of **NP** is: $L \in \mathbf{NP}$ if and only if there is a polynomial-time NDTM that accepts exactly the strings in $L$.

# 4 A picture of the world of computational problems

We have classified decision problems (=languages) according to their computational difficulty. We know that every problem in $\mathbf{P}$ is in $\mathbf{NP}$, that every problem in $\mathbf{NP}$ 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

$$\{< \mathcal{M}, w >: \mathcal{M} \text{accepts } w\}$$

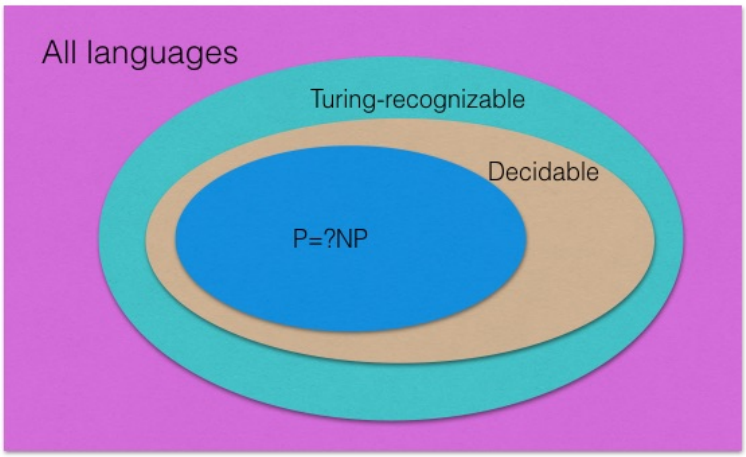


Figure 6: ..or is this?

is a Turing-recognizable language that is not decidable. It is a little harder to separate $\mathbf{NP}$ from the decidable languages, but it is known that this inclusion is also strict.

What about the inclusion $\mathbf{P} \subseteq \mathbf{NP}$? 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem. The common belief is that the inclusion is strict. If $\mathbf{P} = \mathbf{NP}$, many strange things would follow (including the collapse of cryptographic systems), but the question remains one of the great unsolved problems of mathematics.