Many textbooks define the class NP by a machine model, instead of the definition of an NP-language by polynomially bounded existential quantification. In my opinion the model using existential quantification is more natural. We present the alternative definition for completeness (and to be able to read the textbook easier.)

A nondeterministic Turing machine is defined analogously to the (deterministic) Turing machine, except that instead of the transition function $\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times DIRECTION$ (where $Q$ is the set of states, and $\Sigma$ is the tape alphabet) we have a relation. We can think of this as a function $\delta : Q \times \Sigma \rightarrow 2^{Q \times \Sigma \times DIRECTION}$ (a set of possible moves.) Each of these possible moves is a valid move. In other words, from a given pair (state, symbol) is compatible the machine has a valid transition to possibly several triples (state, symbol, direction). The machine nondeterministically chooses one of these valid moves at each step of the computation. The input is accepted if there is a sequence of valid moves from the initial configuration to the accepting configuration and the resources (time, space) used on a given input $x$ are the minimum resources used among valid accepting computation with input $x$.

The collection of valid computations on a given input $x$ is, in general a complicated object. We can define a computation graph, where the vertices are the snapshots of the configurations of the nondeterministic Turing machine, and there is a directed edge from configuration $s_i$ to configuration $s_j$.
when there is a valid move of the machine that transforms $s_i$ into $s_j$. The resulting graph can have cycles and infinite paths.

To simplify things, we can add an extra counter tape to the machine that acts as a clock: after each move we add 1 to this counter. This will prevent cycles (Why? Prove!) and the resulting graph will be a dag.

We define NP on this model as the collection of languages that are accepted by such machines if the are polynomial time bound.

We argue that this is equivalent to our definition.

First, note that without loss of generality we can assume that there are at most two valid transitions from each configuration. (Think about expanding a case statement into if .. then ... else conditionals.)

So each choice can be represented by a bit – say 0 if the first choice was taken and 1 if the second was taken. In this way, any sequence of choices can be represented by a bit string, and the length of the string is no more than the number of steps used by the nondeterministic machine. A deterministic Turing machine, given such a string of choices can execute the corresponding steps taken by the nondeterministic machine. So there is an accepting computation by the nondeterministic machine exactly when there is a binary string – a 'proof' in our terminology – that the deterministic machine can use to certify that $x$ is accepted. When the string is of polynomial length – which happens if the nondeterministic machine made a polynomial number of choices – we recover our previous definition of NP.

An alternative proof, found in some texts, is to argue that the nondeterministic machine can write out all nondeterministic choices in the beginning of its computation on a separate tape, then do the choices during its computation as determined by the next bit in this tape. It is then clear that the string on this tape is exactly the ‘proof string’ in our definition of NP by polynomial existential quantification.

2 Turing Reductions

Recall that we have defined polynomial time reductions and NP-hardness by saying that a set $H$ is NP-hard if for every language $L \in NP$ there is a polynomial time computable function $f_L$ such that $\forall x \in L$ iff $f_L(x) \in A$

In some situations the definition is not powerful enough. Consider the NP-complete set 3-SAT. Consider the problem 3- UNSAT: 3-DNF formulae that
are tautologies. (DNF—disjunctive normal form is the collection of Boolean formulas that are ORs of ANDs of literals. A tautology is a Boolean formula that evaluates to 1 (TRUE) for all Boolean assignments.) It is easy to see (De Morgan Laws) that the negation of a CNF formula is a DNF formula. Moreover a formula is a tautology iff its negation is unsatisfiable.

So it is clear that if $3-SAT \in P$ then so are tautologies (and conversely.) Yet a there does not seem to exist a polynomial time function that would reduce TAUTOLOGIES to SAT.

We have seen the definition of oracle Turing machines in the exercises. We define now *Cook reductions*, a more powerful reduction.

**Definition 1.** A language $L$ is Cook reducible to a language $U$ if there is a polynomial time bounded oracle Turing machine $M^{(i)}$ such that $M$ with oracle $U$ accepts $L$. In other words $M^{(U)}$ accepts $L$.

We can define the notions of NP-hard and NP-complete for Cook reductions analogously:

- $U$ is (Cook) NP-hard if every language $L \in NP$ Cook-reduces to it.
- A language $H$ is (Cook) NP-complete if it is (Cook) NP-hard and is in NP.

Our previous definition is more precisely called a Karp-reduction.

### 2.1 Comparisons between Cook and Karp Reductions

- Karp reductions are a special case of Cook reductions where the oracle is called only once, and we are required to ‘pass along’ the answer of the oracle.
- Cook reductions can ask repeated questions of the oracle, and the questions may depend on the previous computation and of the previous oracle queries.
- Cook reductions can deal with complements
- Most reductions in the text are Karp reductions

Note that it is still true that $P = NP$ iff a (Cook) NP-complete language is in P.

**Theorem 2.** If a (Cook) NP-complete set $H$ is in $P$ then $P = NP$
Proof. Consider any language $L \in NP$. If $H$ is NP-complete, there is an oracle Turing machine $M^{(H)}$ such that $M^{(H)}$ is an algorithm for deciding membership in $L$. Moreover there is a constant $c_1$ such that on input $x$, $M$ takes at most $|x|^{c_1}$ steps (recall that getting an answer to an oracle question takes a single step once the question has been written on the oracle tape.)

If $M$ asks a membership question to the oracle about some string $w$, we must have $|w| \leq |x|^{c_1}$, since it takes a step to write a symbol on the oracle tape, and $M$ took no more than $|x|^{c_1}$ steps.

If $H \in P$ then there is an algorithm (Turing machine $K$) and a constant $c_2 > 0$ such that on inputs $z$ of length $m$, $K$ decides membership of $z$ in $H$ in at most $m^{c_2}$ moves.

So, instead of making oracle calls, we use the machine $K$ as a subroutine to obtain the answer. By the arguments above, the resulting algorithm runs in time at most $|z|^{c_1} = (|x|^{c_1})^{c_2} = |x|^c$ time (where $c = c_1 c_2$), so the algorithm runs in polynomial time.

\[\square\]

Lemma 3. Cook reductions are transitive.

The proof is left to the reader.

2.2 Existence vs. Search

The problems we have been looking at were YES/NO questions. In other words, we try to answer the question whether there exists a solution to a problem (is a given graph Hamiltonian? is a given formula satisfiable? is there a traveling salesman solution of cost $k$?). We often also want to find a solution.

We can imagine a universe where we could get answers to NP-complete questions—would we necessarily be able to find a solution, when we know that one exists? Or could it happen that an oracle just tells us – yes the formula is satisfiable, yet it still takes us exponential time to find a satisfying assignment?

Fortunately, if we can ask multiple questions to the oracle, we can find a solution. The trick, self-reducibility, is to use Cook reductions cleverly.

Let us take the example of satisfiability. Let $F(x_1, x_2, \ldots, x_n)$ be the formula.

We ask the oracle: ‘Is $F$ satisfiable?’
If the answer is NO, we’re done. If the answer is YES, we set Boolean variable $x_n$ to 0, simplify the resulting expression, obtaining a new formula $F_{n-1}(x_1, x_2 \cdots x_{n-1})$

Now we ask the oracle: ‘Is $F_{n-1}$ satisfiable?’

If the answer is YES, we made progress. We know that there is a satisfying assignment with $x_n = 0$, and we have a smaller problem (and since satisfiability of formulas with a single variable is easy, we have a recursive algorithm).

The insight is that we can also reduce the size of the problem when the answer is NO—we just set $x_n = 1$ and we are done as before. (Why?)