ITCS 532:
8. **NP**-completeness

Rob Egrot

Verifiers

Definition 1 (Verifier)

Given a language L over a finite alphabet Σ, a verifier for L is a modified Turing machine V, where V takes two inputs instead of one (consider the inputs to be separated by a special symbol), and such that

$$x \in L \iff \text{there is } y \in \Sigma^* \text{ such that } V(x, y) \text{ accepts}$$

- Here y is called a *certificate* for x.
- A verifier must halt for all inputs.
- We say a verifier for L runs in p-time if it runs in p-time with respect to the length of x for all inputs (x, y).
- The certificate is like a proposed solution.
- The verifier accepts x, which encodes an instance of a decision problem, if and only if there's some proposed solution y that it can check is actually a solution.

Example

Example 2 (Hamiltonian circuits)

- An instance of the Hamiltonian circuit problem is a graph G.
- A certificate for G is just a sequence s of vertices (in coded form).
- A verifier V for the Hamiltonian circuit problem would accept on input $(\mathbf{code}(G), \mathbf{code}(s))$ if and only if s defines a Hamiltonian circuit in G.

Theorem 3

A language has a verifier if and only if it is recursively enumerable.

The Class **NP**

Definition 4

NP is the class of all decision problems that have a *p*-time verifier.

- Note that **NP** does *not* stand for *non-polynomial*.
- **NP** actually stands for *non-deterministic polynomial*.
- The name comes from an alternative definition of the class **NP**, which we will see now.

Non-deterministic Turing machines

- A non-deterministic Turing machine (*NTM*) is a Turing machine variant whose transition function δ has the form:

$$\delta : Q \times \Sigma \cup \{:, \llcorner\} \to \wp\Big(Q \times \big(\Sigma \cup \{\llcorner, \leftarrow, \rightarrow\}\big)\Big)$$

- That is, δ is now a function that takes as input the state of the machine and the symbol being read by the tape head, and returns a *set* of pairs of new states and tape head actions.
- Obviously it doesn't make sense for a machine to enter a set of states, or write a set of symbols on the tape.
- We interpret the output of δ as a *choice*.
- In a run of an NTM, at every step the machine picks one pair of states and tape head actions from the set of possibilities defined by δ.
- This is why these machines are called non-deterministic.

Non-deterministic Turing machines

- For example, suppose the NTM is in state q and reading symbol σ, and that $\delta(q, \sigma) = \{(q_1, \sigma_1), (q_2, \sigma_2)\}$.
- Then the machine can either go into state q_1 and write σ_1 on the tape, or go into state q_2 and write σ_2 on the tape.
- Given an input I and a non-deterministic Turing machine N, we say $N(I)$ *accepts* if there is some sequence of δ choices resulting in the accept state.
- We say $N(I)$ *rejects* if every possible sequence of δ choices results in rejection.
- If neither of these is true then $N(I)$ is undefined.
- A sequence of δ choices for $N(I)$ that either reaches a halt state or continues forever is called a *run* of $N(I)$.
- If every possible run of $N(I)$ halts for all possible inputs I then we say N is a *decider*.

The Run Time of an NTM

- It's not obvious how the run time of an NTM should be defined in general, but for deciders we do so as follows:
  - For an input I the run time of $N(I)$ is defined to be the length of the longest possible run of $N(I)$.
  - We define the run time of N as a function $f : \mathbb{N} \to \mathbb{N}$ by

$$f(n) = \max\{\text{run time of } N(I) \text{ such that length } I \text{ is } n\}$$

  - As usual we use big O notation so we can talk about f without having to explicitly define it.
- Note that in this definition of run time we use the longest run, even if there's a shorter run that accepts.
- For example, if for some N and some I there are exactly two runs, one which accepts after 2 steps and one which rejects after 10 steps, then the run time of $N(I)$ is 10.

Determinism vs Non-Determinism

The following theorem says NTMs are not more powerful than TMs (if we don't care about run time).

Theorem 5

Let L be a formal language over a finite alphabet. Then D_L (the decision problem associated with L) is solvable by a Turing machine if and only if it is solvable by a non-deterministic Turing machine.

Proof.

- Clearly every ordinary (i.e. deterministic) Turing machine T is equivalent to a non-deterministic Turing machine T'.
- Just define δ' by e.g. setting $\delta'(q, \sigma) = \{(q', \sigma')\}$ when $\delta(q, \sigma) = (q', \sigma')$.
- The converse is more difficult, but the basic idea is that we can use dovetailing to compute every possible run of an NTM using a specially designed standard TM.

□

NTMs and Verifiers

Theorem 6

Let L be a formal language over a finite alphabet. Then L has a p-time verifier if and only if L can be decided by a non-deterministic Turing machine in p-time.

Proof

- If L is a language, we must show that:

  1. If L has a p-time verifier V, then we can construct an NTM N_V that decides L in p-time.

  2. If there is an NTM N that decides L in p-time, then we can construct a p-time verifier V_N for L.

Has Verifier Implies Decided by NTM

- Let V be a p-time verifier for L.
- V runs in p-time, so there is $n \in \mathbb{N}$ such that $V(x, y)$ must halt within $C|x|^k$ steps whenever $|x| \geq n$.
- The idea is that we define a non-deterministic Turing machine N_V that, given x, first constructs a string y and then runs $V(x, y)$.
- Implementing this idea is a little tricky, as there are some details we need to be careful with.
- First, note that there are a finite number of strings whose length is less than n.
- For each $x \in L$ with $|x| < n$, define y_x to be a string of minimal length such that $V(x, y_x)$ accepts.
- Since there are a finite number of these y_x elements, define l to be the length of the longest of these.

Has Verifier Implies Decided by NTM

- We define N_V so that a 'run' of $N_V(x)$ is as follows:
  1. Check the length of x.
  2. If $|x| \geq n$ then non-deterministically construct y so that $|y| \leq C|x|^k$. Otherwise non-deterministically construct y such that $|y| \leq l$.
  3. Run $V(x, y)$ (this part is deterministic).
- Consider first the case where $|x| \geq n$.
- If $x \in L$ then there is a certificate y such that $V(x, y)$ accepts within $C|x|^k$ steps.
- The upper bound on the number of steps means that $|y| \leq C|x|^k$ too.
- This means there is an accepting run for $N_V(x)$.
- Alternatively, if $x \notin L$ then $V(x, y)$ rejects for all y, and this is obviously still true if we restrict to y where $|y| \leq C|x|^k$.
- Constructing y takes p-time, and running $V(x, y)$ takes p-time.
- So the longest possible run of $N_V(x)$ is also p-time.

Has Verifier Implies Decided by NTM

- Consider now the case where $|x| < n$.
- Suppose $x \in L$.
- Then there is y_x with $|y_x| \leq l$ and such that $V(x, y_x)$ accepts, so $N_V(x)$ accepts.
- Alternatively, if $x \notin L$ then there is no y such that $V(x, y)$ accepts, so $N_V(x)$ rejects.
- Combining this with the case where $|x| \geq n$ we see that N_V decides L.
- What is the running time of N_V?
- We don't care about the running time of $N_V(x)$ in the case where $|x| < n$.
- We have also shown that, for all x with $|x| \geq n$, a run of $N_V(x)$ is polynomially bounded.
- Thus N_V runs in p-time as required.

Decided by NTM Implies Has Verifier

- Given N that decides L in p-time we define a verifier V_N where a string y that is a potential certificate encodes a sequence $n_0, n_1, \ldots, n_k$ of natural numbers.

- Every time N 'makes a choice', we can assign an ordering of the possible alternatives.

- For example, if

$$\delta : (q, \sigma) \mapsto \{(q_0, \sigma_0), (q_1, \sigma_1), (q_2, \sigma_2)\}$$

we can arbitrarily choose an order and assume without loss of generality that $\{(q_0, \sigma_0), (q_1, \sigma_1), (q_2, \sigma_2)\}$ is the ordered sequence $((q_0, \sigma_0), (q_1, \sigma_1), (q_2, \sigma_2))$.

Decided by NTM Implies Has Verifier

- We define V_N so that $V_N(x, y)$ operates by computing $N(x)$, and such that the 'choices' made by N correspond to the numbers in the sequence $n_0, n_1, \ldots, n_k$ defined by y.
- If y does not correspond to such a sequence, or if this sequence is incompatible with $N(x)$ in any way, then we define V_N so that $V_N(x, y)$ rejects.
- If there is an accepting run of $N(x)$ then this will correspond to a sequence of 'correct' choices, and thus to a y such that $V_N(x, y)$ accepts.
- Alternatively, if there is no accepting run then $V_N(x, y)$ will reject for all y.

Decided by NTM Implies Has Verifier

- As N runs in p-time there are $C, m, k \in \mathbb{N}$ such that $N(x)$ takes at most $C|x|^k$ steps whenever $|x| \geq m$.
- Thus N makes at most $C|x|^k$ choices when computing $N(x)$.
- Assuming we have picked a sensible encoding scheme, it will be possible to look up the numbers of the choices appropriately in p-time and act accordingly.
- Thus V_N runs in p-time as required.

Revisiting **NP**

Corollary 7

NP *is the class of all decision problems that can be solved by a non-deterministic Turing machine in p-time.*

Proof.

- Remember that we defined **NP** to be the class of decision problems for which a p-time verifier exists.
- By theorem 6 this is exactly the class of problems solvable in p-time by an NTM.

$\square$

P vs **NP**

- Question: Why is it obvious that $\mathbf{P} \subseteq \mathbf{NP}$?
- Is it possible that $\mathbf{P} = \mathbf{NP}$?
- This is possibly the most important open question in theoretical computer science.
- The Clay Mathematics Institute lists it as one of their seven 'Millennium Problems'.
- The **P** vs. **NP** question symbolizes something of profound practical significance:
- Is it intrinsically harder to find solutions than to verify them?
- If the answer to this is 'no', i.e. if $\mathbf{P} = \mathbf{NP}$, then there could be efficient algorithms for solving all kinds of currently hard problems (including those used in encryption systems).
- Most computer scientists believe that $\mathbf{P} \neq \mathbf{NP}$.
- This has turned out to be very hard to prove.
- But there are lots of **NP** problems that nobody has ever found a p-time algorithm for.

NP-hardness

Definition 8 (**NP**-hard)

A decision problem B is **NP**-hard if for all decision problems $A \in \mathbf{NP}$ we have $A \leq_p B$.

- Consider the definition above and think about what it means.
- A problem is **NP**-hard if every problem in **NP** reduces to it in polynomial time.
- Informally, we interpret this as meaning an **NP**-hard problem is at least as hard as the hardest problems in **NP**.
- If we could find a p-time algorithm for solving an **NP**-hard problem then we would be able to find p-time algorithms for all **NP** problems.
- In other words it would show that $\mathbf{P} = \mathbf{NP}$.

NP-completeness

- It's not immediately obvious that **NP**-hard problems exist.
- But, as we will see shortly, they do, and, moreover, there are even **NP**-hard problems that are also members of **NP**.
- This leads us to the following definition.

Definition 9 (**NP**-complete)

A decision problem is **NP**-complete if it is **NP**-hard and also a member of **NP**.

NP-completeness

- We will give an example of an **NP**-complete problem soon.
- Once we have one **NP**-complete problem, however, it becomes much easier to find more, due to the following theorem.

Theorem 10

*If B is **NP**-complete and $C \in \mathbf{NP}$, then $B \leq_p C \implies C$ is **NP**-complete.*

Proof.

- Let $A \in \mathbf{NP}$.
- We need to show that $A \leq_p C$.
- But we know $A \leq_p B$ as B is **NP**-complete, and $B \leq_p C$ by assumption, so this follows from transitivity of $\leq_p$.

$\square$

PSAT

- To define our first **NP**-complete problem we need some preliminary definitions from propositional logic.

Definition 11 (Boolean formula)

A Boolean formula is a finite string of propositional variables, brackets, and logical symbols from $\{\vee, \wedge, \neg\}$ constructed according to the following rules:

1. p is a Boolean formula for all propositional variables p.
2. If ϕ and ψ are Boolean formulas then $(\phi \vee \psi)$ and $(\phi \wedge \psi)$ are Boolean formulas.
3. If ϕ is a Boolean formula then $\neg\phi$ is a Boolean formula.

PSAT

Definition 12 (Satisfiable)

A Boolean formula ϕ is satisfiable if there is an assignment of 1 or 0 ('true' or 'false') to every propositional variable appearing in ϕ such that ϕ is 'true' in the resulting truth table.

Definition 13 (PSAT)

PSAT is the decision problem that asks if a given Boolean formula is satisfiable.

PSAT is **NP**-complete

Theorem 14

PSAT *is* **NP***-complete.*

Proof sketch

- We will sketch a proof without going too far into the details.
- First we must check that PSAT is in **NP**.
- By definition, a problem is in **NP** if it has a p-time verifier.
- We can check if a given Boolean formula evaluates to 'true' when given a specific variable assignment in p-time.
- To do this we can rewrite a Boolean formula into something called *reverse Polish notation* using the *shunting yard algorithm*.
- In reverse Polish notation we can evaluate expressions just by reading them from left to right, which takes linear time.
- Since the shunting yard algorithm runs in linear time too, we can evaluate Boolean formulas for specific values of their propositions in linear time.

PSAT is **NP**-complete

- Now we must show every problem in **NP** reduces to PSAT.
- Let A be a decision problem in **NP**.
- We know there's an NTM that decides A in p-time.
- We must find a p-time algorithm that takes an instance I of A and turns it into a Boolean formula that is satisfiable if and only if I is a 'yes' instance.
- Since all we know about A is that it is in **NP** we're going to have to use the NTM that decides A in our reduction.
- Let N be an NTM that decides A.
- The idea is that we will create a Boolean formula that expresses the idea that $N(I)$ has an accepting run.
- This formula will be satisfiable if and only if an accepting run exists. I.e. if and only if I is a 'yes' instance.

PSAT is **NP**-complete

- This idea is quite similar to how we showed the entscheidungsproblem is undecidable.
- This time we don't have the expressive power of first-order logic to work with.
- For the entscheidungsproblem proof we used predicates to describe the state of the tape at each step in the computation.
- In propositional logic we don't have predicates, so we have to use propositional variables for the same purpose.
- But our formula can only use a finite number of propositional variables.
- This is where the bounded running time of N helps us.

PSAT is **NP**-complete

- As in the entscheidungsproblem proof, we think of a run of a TM as a two dimensional grid.
- The rows of the grid represent the tape at successive steps of the calculation.
- Since N is non-deterministic, $N(I)$ is associated with multiple different runs, and so with multiple different grids.
- Since N runs in p-time, we can suppose that for large n its run time on an input of length n is less than Cn^k.
- Suppose the length of I is n. Then this means that every grid associated with a run of $N(I)$ must fit into $Cn^k \times Cn^k$.
- Cells outside of this range are inaccessible to a machine halting in fewer than Cn^k steps, so we can ignore them.

PSAT is **NP**-complete

- Let Q be the set of states of N, and let Σ be its (finite) set of symbols.
- Our Boolean formula ϕ will use the following set of propositional variables:
  - For every $i, j \leq Cn^k$ and every $\sigma \in \Sigma \cup \{:, \llcorner\!\lrcorner\}$ we have $p_{i,j,\sigma}$. This variable is supposed to be 'true' if σ is written on the tape at position i and time j.
  - For every $i, j \leq Cn^k$ and every $q \in Q$ we have $p_{i,j,q}$. This variable is supposed to be 'true' if the machine is in state q and the tape head is at position i at time j.

PSAT is **NP**-complete

- The idea is that ϕ will express the following:

1. Every cell in the grid must contain exactly one symbol.
2. The first row of the grid corresponds to the input I.
3. The tape head starts in the right place.
4. The machine is in exactly one state at every step.
5. The machine starts in the start state.
6. The tape head is always somewhere on the tape.
7. Every row except the first must represent a configuration of $N(I)$ that follows from the configuration corresponding to the previous row according to some choice defined by the transition function δ of N.
8. At some point the machine enters the accept state.

PSAT is **NP**-complete

- It helps to break down the construction of ϕ into parts.
- For example, we can express 1 using

$$\phi_1 = \bigwedge_{0 \leq i,j \leq Cn^k} \Big((\bigvee_{\sigma \in \Sigma \cup \{:, \llcorner\!\lrcorner\}} p_{i,j,\sigma}) \wedge \bigwedge_{\sigma \neq \sigma' \in \Sigma \cup \{:, \llcorner\!\lrcorner\}} \neg (p_{i,j,\sigma} \wedge p_{i,j,\sigma'}) \Big)$$

- The construction of ϕ turns out to take $O(n^{2k})$ steps.
- ϕ will be satisfiable if and only if $N(I)$ has an accepting run.
- I.e. a sequence of δ choices leading to an accept state.

PSAT is **NP**-complete

- What about 'short' inputs, such that the run time bound Cn^k does not apply?
- There are only a finite number of these, so there's still an upper bound on the size of the 'grids' representing the computations.
- The construction time of ϕ for these short inputs doesn't matter because the definition of p-time only requires polynomial bounding for 'large' input lengths.
- So we have shown that $A \leq_p$ PSAT.

Some Final Comments

- Using p-time reduction we can use PSAT to show that other problems are **NP**-complete too.
- It turns out that lots of interesting combinatorial problems are.
- Important examples include the Hamiltonian circuit and Hamiltonian path problems, the traveling salesman decision problem, the knapsack problem, and many more.
- Finally, there are problems that are **NP**-hard but not in **NP** (and so not **NP**-complete).
- The halting problem is one example.
- It is obviously not in **NP** as it isn't even decidable, but we can show that PSAT reduces to it in p-time.
- So by theorem 10 the halting problem is **NP**-hard.