Formalising Computation

Matvey Soloviev (Cornell University)

CS 4820, Summer 2020

Last time

Last time:

- The fastest algorithm that solves a problem is a measure of how hard it is.
- Reductions: use a solution to one problem as a building block (subroutine) to solve another.
- Start with a hard problem, only do a little work $\Rightarrow$ still have a hard problem.
- But how do we know that any problem is hard?
- Want a subroutine that lots of problems can be reduced to. One of them is bound to be hard, and that would be enough to imply that the subroutine is hard.

Today

- Hope to find a subroutine that works for all problems that have solutions a computer can verify to be correct in poly-time.
- No reason to assume that all of these would be easy.
- Need a formal definition of what it means for a computer to do stuff.

Goals

We want something that is

- Powerful enough that it can run any algorithm,

Goals

We want something that is

- Powerful enough that it can run any algorithm,
- but simple enough that we can define it formally and reason about it mathematically.

Enter Turing Machines

A Turing Machine is an idealised abstract model of computation that satisfies the two.

The second point will become apparent when we define them. The first one is the subject of another scientific claim:

Church-Turing Thesis

Every algorithm that can be run on any physically realisable computer can be run on a Turing machine with at most polynomial overhead.

The problem of language (1)

In the context of complexity theory, we often make our formal life easier by only looking at yes-or-no problems. This sounds like a serious limitation, but actually isn't:

The problem of language (1)

In the context of complexity theory, we often make our formal life easier by only looking at yes-or-no problems. This sounds like a serious limitation, but actually isn't:

Any problem that asks for a solution that can be written out as a string can be reduced to a logarithmic number of instances of a yes-or-no problem using binary search. Specifically, we can reduce any problem of the form "find a solution s", where s is a string, to "does there exist a solution $s < s'$?", where the comparison string s' is part of the input.

The problem of language (2)

For historical reasons, we also call yes-or-no problems languages, and identify them with the set of strings on which the answer is “yes”.

The problem of language (2)

For historical reasons, we also call yes-or-no problems languages, and identify them with the set of strings on which the answer is "yes".

For example: consider the problem "find the sum of a list of integers in decimal notation".

- The corresponding yes-or-no (decision) problem is: "given an integer n and a list of integers $m_1, m_2, \ldots, m_k$, is the sum of the list $\leq n$?"

The problem of language (2)

For historical reasons, we also call yes-or-no problems languages, and identify them with the set of strings on which the answer is "yes".

For example: consider the problem "find the sum of a list of integers in decimal notation".

- The corresponding yes-or-no (decision) problem is: "given an integer n and a list of integers $m_1, m_2, \ldots, m_k$, is the sum of the list $\leq n$?"
- Suppose we encode the input as a string $n; m_1, m_2, m_3, \ldots, m_k$. Then the language corresponding to the problem is the set

$$S = \{n; m_1, \ldots, m_k \mid \sum_{i=1}^{k} m_i \leq n\}.$$

Turing machines, roughly (1)

A Turing Machine is a state machine that operates by reading and writing from a tape that is partitioned into discrete cells, and unbounded in one direction. Each cell of the tape contains exactly one symbol from the set $\Sigma = \{\triangleright, \sqcup, \ldots\}$.

At any point in time, the machine is in one state from a set $Q = \{q_1, q_2, \ldots\}$ or one of the two special states q_{yes} and q_{no}, and points at one cell of the tape.

Turing machines, roughly (2)

Depending on the contents of the cell and its current state, it:

Turing machines, roughly (2)

Depending on the contents of the cell and its current state, it:

- Picks a new symbol to replace the contents of the cell it is pointing at with;

Turing machines, roughly (2)

Depending on the contents of the cell and its current state, it:

- Picks a new symbol to replace the contents of the cell it is pointing at with;
- Chooses a new state to transition to:

Turing machines, roughly (2)

Depending on the contents of the cell and its current state, it:

- Picks a new symbol to replace the contents of the cell it is pointing at with;
- Chooses a new state to transition to:
- Moves the head either left or right by one cell, or stays in place.

Turing machines, roughly (2)

Depending on the contents of the cell and its current state, it:

- Picks a new symbol to replace the contents of the cell it is pointing at with;
- Chooses a new state to transition to:
- Moves the head either left or right by one cell, or stays in place.

Execution stops when the machine reaches either the state q_{yes} or q_{no}.

Turing machines (picture)

Switch to camera view :)

Turing machines, formally

Formally, a Turing machine is a tuple $M = (Q, \Sigma, s, \delta)$, where

Turing machines, formally

Formally, a Turing machine is a tuple $M = (Q, \Sigma, s, \delta)$, where

- Q is a set of states including special states q_{yes} and q_{no};

Turing machines, formally

Formally, a Turing machine is a tuple $M = (Q, \Sigma, s, \delta)$, where

- Q is a set of states including special states q_{yes} and q_{no};
- Σ is a set of symbols including special symbols $\triangleright$ and $\sqcup$;

Turing machines, formally

Formally, a Turing machine is a tuple $M = (Q, \Sigma, s, \delta)$, where

- Q is a set of states including special states q_{yes} and q_{no};
- Σ is a set of symbols including special symbols $\triangleright$ and $\sqcup$;
- $s \in Q$ is the starting state;

Turing machines, formally

Formally, a Turing machine is a tuple $M = (Q, \Sigma, s, \delta)$, where

- Q is a set of states including special states q_{yes} and q_{no};
- Σ is a set of symbols including special symbols $\triangleright$ and $\sqcup$;
- $s \in Q$ is the starting state;
- $\delta \in (Q \times \Sigma) \to (Q \cup \{q_{yes}, q_{no}\}) \times \Sigma \times \{L, R, S\}$ is the transition function.

Turing machines, formally

Formally, a Turing machine is a tuple $M = (Q, \Sigma, s, \delta)$, where

- Q is a set of states including special states q_{yes} and q_{no};
- Σ is a set of symbols including special symbols $\triangleright$ and $\sqcup$;
- $s \in Q$ is the starting state;
- $\delta \in (Q \times \Sigma) \to (Q \cup \{q_{yes}, q_{no}\}) \times \Sigma \times \{L, R, S\}$ is the transition function.

For every Turing machine, there is a corresponding set of Turing machine configurations $(q, p, t) \in Q \times \mathbb{N} \times \Sigma^\omega$, where q is the current state, p is the position of the head and t is the current contents of the tape.

Stepping a Turing machine, formally

We define an operator

$$\mathrm{step}_M(q, p, t) = (q', p', t'),$$

where $\delta(q, t[p]) = (q', \sigma', d)$ and $t' = t[[p] \leftarrow \sigma']$, that is, the tape with the pth entry replaced with σ', and

$$p' = \begin{cases} p - 1 & \text{if } d = L \text{ and } p > 0 \\ p + 1 & \text{if } d = R \\ p & \text{otherwise.} \end{cases}$$

The language of a Turing machine

We say that M accepts a string $\alpha \in (\Sigma \setminus \{\triangleright, \sqcup\})^*$ if there exists a finite $n \in \mathbb{N}$ such that $\text{step}_M^n(s, 0, \triangleright\alpha \sqcup \sqcup \ldots) = (q_{yes}, \ldots, \ldots)$.

The language of a Turing machine

We say that M accepts a string $\alpha \in (\Sigma \setminus \{\triangleright, \sqcup\})^*$ if there exists a finite $n \in \mathbb{N}$ such that $\text{step}_M^n(s, 0, \triangleright\alpha \sqcup \sqcup \ldots) = (q_{yes}, \ldots, \ldots)$.

We say that it rejects α (...) if (...) $= (q_{no}, \ldots, \ldots)$.

The language of a Turing machine

We say that M accepts a string $\alpha \in (\Sigma \setminus \{\triangleright, \sqcup\})^*$ if there exists a finite $n \in \mathbb{N}$ such that $\text{step}_M^n(s, 0, \triangleright\alpha \sqcup \sqcup \ldots) = (q_{yes}, \ldots, \ldots)$.

We say that it rejects α (...) if (...) $= (q_{no}, \ldots, \ldots)$.

We say that it halts on α if it either accepts or rejects α. (It could run forever!)

The language of a Turing machine

We say that M accepts a string $\alpha \in (\Sigma \setminus \{\triangleright, \sqcup\})^*$ if there exists a finite $n \in \mathbb{N}$ such that $\text{step}_M^n(s, 0, \triangleright\alpha \sqcup \sqcup \ldots) = (q_{yes}, \ldots, \ldots)$.

We say that it rejects α (...) if (...) $= (q_{no}, \ldots, \ldots)$.

We say that it halts on α if it either accepts or rejects α. (It could run forever!)

The language of the Turing machine M $\mathcal{L}_M$ is the set of all strings that M accepts. If M halts on all strings, we say M decides the language $\mathcal{L}_M$.

Runtime of Turing machines

The number n of steps that it took for M to accept or reject a string α is a measure of how long the machine ran.

Runtime of Turing machines

The number n of steps that it took for M to accept or reject a string α is a measure of how long the machine ran.

If there exists a polynomial p such that for every $\alpha \in \Sigma^*$, M halts in time $p(|\alpha|)$, then we say that M is a polynomial-time Turing machine.

The class P

The set of all languages that are decided by polynomial-time Turing machines is called P.

This includes decision versions of many familiar problems:

The class P

The set of all languages that are decided by polynomial-time Turing machines is called P.

This includes decision versions of many familiar problems:

- Given a graph G and two vertices v, w, is the shortest path from v to w no longer than n?

The class P

The set of all languages that are decided by polynomial-time Turing machines is called P.

This includes decision versions of many familiar problems:

- Given a graph G and two vertices v, w, is the shortest path from v to w no longer than n?
- Given a flow graph G, is the max flow no greater than n?

The class P

The set of all languages that are decided by polynomial-time Turing machines is called P.

This includes decision versions of many familiar problems:

- Given a graph G and two vertices v, w, is the shortest path from v to w no longer than n?
- Given a flow graph G, is the max flow no greater than n?
- Alternative: ...is there a flow of value at least n?
- Given a sequence of numbers, is there a subsequence of sum at least n?

Let’s take a short break. See the camera view for a question.

The class NP (1)

Problems in P are quite simple, in a sense: there is a polynomial-time algorithm to decide them.

The class NP (1)

Problems in P are quite simple, in a sense: there is a polynomial-time algorithm to decide them.

Recall that our goal was to find a class of problems that we have reason to believe contains at least one hard problem.

The class NP (1)

Problems in P are quite simple, in a sense: there is a polynomial-time algorithm to decide them.

Recall that our goal was to find a class of problems that we have reason to believe contains at least one hard problem.

One such class is the class NP. This is commonly formulated as the class of languages that have polynomial-length certificates for a polynomial-time verifier.

The class NP (1)

Problems in P are quite simple, in a sense: there is a polynomial-time algorithm to decide them.

Recall that our goal was to find a class of problems that we have reason to believe contains at least one hard problem.

One such class is the class NP. This is commonly formulated as the class of languages that have polynomial-length certificates for a polynomial-time verifier.

What does this mean?

The class NP (2)

Intuitively, for every string in the language (every “yes” instance of the problem), there is a proof that it is in fact a “yes” instance, and a Turing machine that verifies that the proof is legitimate in polynomial time. Obviously, no valid proof that it is “yes” may exist for a “no” instance.

The class NP (2)

Intuitively, for every string in the language (every "yes" instance of the problem), there is a proof that it is in fact a "yes" instance, and a Turing machine that verifies that the proof is legitimate in polynomial time. Obviously, no valid proof that it is "yes" may exist for a "no" instance.

For instance, the Hamiltonian Cycle decision problem asks if a graph has a looping path (sequence of edges) that visits every vertex exactly once.

The class NP (2)

Intuitively, for every string in the language (every "yes" instance of the problem), there is a proof that it is in fact a "yes" instance, and a Turing machine that verifies that the proof is legitimate in polynomial time. Obviously, no valid proof that it is "yes" may exist for a "no" instance.

For instance, the Hamiltonian Cycle decision problem asks if a graph has a looping path (sequence of edges) that visits every vertex exactly once.

The list of edges in a cycle can serve as a certificate: an algorithm can quickly check that they are in fact edges of the graph, any two adjacent edges have a vertex in common (so the path is connected), and every vertex is visited exactly once.

The class NP (3)

Formally, we say a language $L \subseteq \Sigma^*$ is in NP if

The class NP (3)

Formally, we say a language $L \subseteq \Sigma^*$ is in NP if

- there exists a polynomial p, function $c : \Sigma^* \to \Sigma^*$ and polynomial-time Turing machine V (the verifier) such that

The class NP (3)

Formally, we say a language $L \subseteq \Sigma^*$ is in NP if

- there exists a polynomial p, function $c : \Sigma^* \to \Sigma^*$ and polynomial-time Turing machine V (the verifier) such that
- $|c(\alpha)| < p(|\alpha|)$;

The class NP (3)

Formally, we say a language $L \subseteq \Sigma^*$ is in NP if

- there exists a polynomial p, function $c : \Sigma^* \to \Sigma^*$ and polynomial-time Turing machine V (the verifier) such that
- $|c(\alpha)| < p(|\alpha|)$;
- for every $\alpha \in L$, V accepts the string $\alpha \sqcup c(\alpha)$;

The class NP (3)

Formally, we say a language $L \subseteq \Sigma^*$ is in NP if

- there exists a polynomial p, function $c : \Sigma^* \to \Sigma^*$ and polynomial-time Turing machine V (the verifier) such that
- $|c(\alpha)| < p(|\alpha|)$;
- for every $\alpha \in L$, V accepts the string $\alpha \sqcup c(\alpha)$;
- for every $\alpha \notin L$, for any string $\beta \in \Sigma^*$, V rejects $\alpha \sqcup \beta$.

The class NP (3)

Formally, we say a language $L \subseteq \Sigma^*$ is in NP if

- there exists a polynomial p, function $c : \Sigma^* \to \Sigma^*$ and polynomial-time Turing machine V (the verifier) such that
- $|c(\alpha)| < p(|\alpha|)$;
- for every $\alpha \in L$, V accepts the string $\alpha \sqcup c(\alpha)$;
- for every $\alpha \notin L$, for any string $\beta \in \Sigma^*$, V rejects $\alpha \sqcup \beta$.

This is kind of cumbersome.

Nondeterministic Turing machines (1)

Let’s talk about quantum :)

Nondeterministic Turing machines (1)

Let's talk about quantum :)

Folk understanding of Schrödinger's Cat: "It is simultaneously dead and alive"

Nondeterministic Turing machines (1)

Let's talk about quantum :)

Folk understanding of Schrödinger's Cat: "It is simultaneously dead and alive"

Folk understanding of quantum computing: "It is faster because it can check all values at once"

Nondeterministic Turing machines (1)

Let's talk about quantum :)

Folk understanding of Schrödinger's Cat: "It is simultaneously dead and alive"

Folk understanding of quantum computing: "It is faster because it can check all values at once"

(This is wrong.)

Nondeterministic Turing machines (2)

We’re theorists, though, so we can pretend that something like that exists.

Nondeterministic Turing machines (2)

We're theorists, though, so we can pretend that something like that exists.

Imagine a Turing machine that can perform several computations simultaneously.

Nondeterministic Turing machines (2)

We're theorists, though, so we can pretend that something like that exists.

Imagine a Turing machine that can perform several computations simultaneously.

Instead of having a single well-defined transition $\delta(q, \sigma) = (q', \sigma', d)$, what if we could try a bunch of different ones in parallel?

That is, take $\delta(q, \sigma)$ to be a subset of $(Q \cup \{q_{yes}, q_{no}\}) \times \Sigma \times \{L, R, S\}$, rather than a single element?

Nondeterministic Turing machines (2)

We're theorists, though, so we can pretend that something like that exists.

Imagine a Turing machine that can perform several computations simultaneously.

Instead of having a single well-defined transition $\delta(q, \sigma) = (q', \sigma', d)$, what if we could try a bunch of different ones in parallel?

That is, take $\delta(q, \sigma)$ to be a subset of $(Q \cup \{q_{yes}, q_{no}\}) \times \Sigma \times \{L, R, S\}$, rather than a single element?

If we do this, we obtain a nondeterministic Turing machine. We say that this machine accepts a string α if at least one course of computation halts.

Nondeterministic Turing machines (3)

Formally, we now need to look at the set of configurations we could be in at any given point, and have the step function return the set of configurations we could have reached from any of them in one step.

Nondeterministic Turing machines (3)

Formally, we now need to look at the set of configurations we could be in at any given point, and have the step function return the set of configurations we could have reached from any of them in one step.

For a single configuration,

$$\begin{aligned}&\mathrm{step}_M(q,p,t)\\ &= \{(q',p',t') \mid \quad (q',\sigma',d) \in \delta(q,t[p]), t' = t[[p] \leftarrow \sigma'],\\ &\qquad\qquad\qquad\quad t' = f(t,d) \text{ as before}\}.\end{aligned}$$

For a set, $\mathrm{step}_M(C) = \bigcup_{(q,p,t)\in C} \mathrm{step}_M(q,p,t)$.

The language of a nondeterministic Turing machine

We say that a nondeterministic M accepts a string $\alpha \in (\Sigma \setminus \{\triangleright, \sqcup\})^*$ if there exists a finite $n \in \mathbb{N}$ such that $\exists (q, p, t) \in \text{step}_M^n(s, 0, \triangleright \alpha \sqcup \sqcup \ldots).\, q = q_{yes}$.

The language of a nondeterministic Turing machine

We say that a nondeterministic M accepts a string $\alpha \in (\Sigma \setminus \{\triangleright, \sqcup\})^*$ if there exists a finite $n \in \mathbb{N}$ such that $\exists (q, p, t) \in \text{step}_M^n(s, 0, \triangleright \alpha \sqcup \sqcup \ldots).\, q = q_{yes}$.

We say that it rejects α if there exists a finite $n \in \mathbb{N}$ such that $\exists (q, p, t)\ (\ldots)\ q = q_{no}$?

The language of a nondeterministic Turing machine

We say that a nondeterministic M accepts a string $\alpha \in (\Sigma \setminus \{\triangleright, \sqcup\})^*$ if there exists a finite $n \in \mathbb{N}$ such that $\exists (q, p, t) \in \text{step}_M^n(s, 0, \triangleright\alpha \sqcup \sqcup \ldots).\, q = q_{yes}$.

We say that it rejects α if there exists a finite $n \in \mathbb{N}$ such that $\exists (q, p, t)\ (\ldots)\ q = q_{no}$?

No! $\forall (q, p, t) \in \text{step}_M^n(s, 0, \triangleright\alpha \sqcup \sqcup \ldots).\, q = q_{no}$.

Another take on NP (1)

Claim: NP is the set of all languages that are decided by nondeterministic polynomial-time Turing machines.

Another take on NP (1)

Claim: NP is the set of all languages that are decided by nondeterministic polynomial-time Turing machines.

Why is this equivalent to the previous definition?

Another take on NP (1)

Claim: NP is the set of all languages that are decided by nondeterministic polynomial-time Turing machines.

Why is this equivalent to the previous definition?

One direction: Take the verifier V, and make it nondeterministic.

Another take on NP (1)

Claim: NP is the set of all languages that are decided by nondeterministic polynomial-time Turing machines.

Why is this equivalent to the previous definition?

One direction: Take the verifier V, and make it nondeterministic.

Assume that no certificate was given at all. Start by moving to the end of the input, then nondeterministically generate every possible certificate at once.

Another take on NP (1)

Claim: NP is the set of all languages that are decided by nondeterministic polynomial-time Turing machines.

Why is this equivalent to the previous definition?

One direction: Take the verifier V, and make it nondeterministic.

Assume that no certificate was given at all. Start by moving to the end of the input, then nondeterministically generate every possible certificate at once.

Then move back to the start, and proceed to run the verifier normally.

Another take on NP (2)

The other direction: recall the certificate-generating function c.
Suppose we have an NTM M that decides the language.

Another take on NP (2)

The other direction: recall the certificate-generating function c. Suppose we have an NTM M that decides the language.

For every "yes" instance α, there must be some path of transitions from the starting state to q_{yes}. Let $c(\alpha)$ be a string that, whenever we have multiple possible transitions, tells us which one to take. This is poly-length because M halts in polynomial time.

Another take on NP (2)

The other direction: recall the certificate-generating function c. Suppose we have an NTM M that decides the language.

For every "yes" instance α, there must be some path of transitions from the starting state to q_{yes}. Let $c(\alpha)$ be a string that, whenever we have multiple possible transitions, tells us which one to take. This is poly-length because M halts in polynomial time.

Build a deterministic Turing machine that crosschecks with the "advice" string before every step.

NP is assumed hard

The class NP contains many problems that we do not know to be in P – that is, do not know a polynomial-time algorithm for. In particular, this includes problems like Hamiltonian Cycle, Graph Colouring and all sorts of puzzles.

NP is assumed hard

The class NP contains many problems that we do not know to be in P – that is, do not know a polynomial-time algorithm for. In particular, this includes problems like Hamiltonian Cycle, Graph Colouring and all sorts of puzzles.

In fact, there are deep reasons to believe that it contains some problems that are not solvable in polynomial time – that is, that $P \neq NP$. (But as of last night, there is no accepted proof.)

A universal subproblem for NP?

So if we found a “universal subproblem” that all problems in it can be reduced to in polynomial time, all our belief that the fastest algorithm for at least some problems in NP is slower than any polynomial in the length of its input would translate into belief that the universal subproblem can not be solved by a polynomial-time algorithm: after all, a function that grows faster than any polynomial, minus a polynomial, still grows faster than any polynomial.

A universal subproblem for NP?

So if we found a “universal subproblem” that all problems in it can be reduced to in polynomial time, all our belief that the fastest algorithm for at least some problems in NP is slower than any polynomial in the length of its input would translate into belief that the universal subproblem can not be solved by a polynomial-time algorithm: after all, a function that grows faster than any polynomial, minus a polynomial, still grows faster than any polynomial.

We would call such a subproblem as hard as NP, or NP-hard. If the problem is itself in NP, we call it complete for the class NP, or, in short, NP-complete.

Next time

As we shall see tomorrow, our formal definition of nondeterministic Turing machines will allow us to establish that an NP-complete problem in fact exists.