Computational Complexity; slides 4, HT 2022
nondeterminism, Cook-Levin theorem

Prof. Paul W. Goldberg (Dept. of Computer Science,
University of Oxford)

HT 2022

Nondeterministic Turing Machines

Definition.

A non-deterministic (1-tape) Turing machine is a 6-tuple $(Q, \Sigma, \Gamma, \Delta, q_0, F)$ where

- Q is a finite set of states
- Σ is a finite alphabet of symbols
- $\Gamma \supseteq \Sigma \cup \{\Box\}$ is a finite alphabet of symbols
- $\Delta \subseteq (Q \setminus F) \times \Gamma \times Q \times \Gamma \times \{-1, 0, 1\}$ transition **relation**
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of final states

As before, we assume $\Sigma := \{0, 1\}$ and $\Gamma := \Sigma \cup \{\Box\}$.

The computation of a non-deterministic Turing machine $M = (Q, \Sigma, \Gamma, \Delta, q_0, F)$ on input w is a "computation tree"
analogy with NFA, (N)PDA

Non-Deterministic Turing Acceptor

Computation path:

Any path from the start configuration to a stop configuration in the configuration tree.

accepting path: the stop configuration is in an accepting state. (also called an accepting run)

rejecting path otherwise

Language accepted by an NTM M:

$$\mathcal{L}(M) := \{w \in \Sigma^* : \text{ there } \textbf{exists} \text{ an accepting path of } M \text{ on } w\}$$

Simulation: Variants of definition of NTM can be simulated with polynomial (runtime) overhead.

Can also simulate with deterministic TM, but not in poly-time

From **P** to **NP**

NP: languages accepted by NTM in polynomially-many steps; equivalently, decision problems whose yes-instances are accepted by (poly-time) NTM

- e.g. 3-SAT, 3-COLOURABILITY, TSP, SAT, etc
- No polynomial time algorithms for these problems are known
- but are in **NP**

"Guess and test": generic **NP** algorithm. As for **P**, pseudocode algorithms are convenient, but don't forget underlying TM model

Non-Deterministic Complexity Classes

- Time classes:
  - NP (a.k.a. NPTIME) $:= \bigcup_{d \in \mathbb{N}} \text{NTIME}(n^d)$
  - NEXPTIME $:= \bigcup_{d \in \mathbb{N}} \text{NTIME}(2^{n^d})$
- Space classes:
  - NLOGSPACE $:= \bigcup_{d \in \mathbb{N}} \text{NSPACE}(d \log n)$
  - NPSPACE $:= \bigcup_{d \in \mathbb{N}} \text{NSPACE}(n^d)$
  - NEXPSPACE $:= \bigcup_{d \in \mathbb{N}} \text{NSPACE}(2^{n^d})$

where NTIME(T) (etc.) means what you think it means. Note that all accepting/non-accepting computations of a NTIME(T) TM should have length at most T

- Time classes:
  - NP (a.k.a. NPTIME) $:= \bigcup_{d\in\mathbb{N}} \mathrm{NTIME}(n^d)$
  - NEXPTIME $:= \bigcup_{d\in\mathbb{N}} \mathrm{NTIME}(2^{n^d})$
- Space classes:
  - NLOGSPACE $:= \bigcup_{d\in\mathbb{N}} \mathrm{NSPACE}(d \log n)$
  - NPSPACE $:= \bigcup_{d\in\mathbb{N}} \mathrm{NSPACE}(n^d)$
  - NEXPSPACE $:= \bigcup_{d\in\mathbb{N}} \mathrm{NSPACE}(2^{n^d})$

where NTIME(T) (etc.) means what you think it means. Note that all accepting/non-accepting computations of a NTIME(T) TM should have length at most T

We have:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{NPSPACE} \subseteq \mathrm{EXP}$$

(hierarchy: sort-of good news)

More Examples of Problems in **NP**

COMPOSITE (NON-PRIME) NUMBER

Input: A positive integer $n > 1$

Question: Are there integers $u, v > 1$ such that $u \cdot v = n$?

SUBSET SUM

Input: A collection of positive integers $S := \{a_1, \ldots, a_k\}$ and a target integer t.

Question: Is there a subset $T \subseteq S$ such that $\sum_{a_i \in T} a_i = t$?

Deterministic vs. Non-Deterministic Time

Clearly, P$\subseteq$NP.

Question: The question P $\stackrel{?}{=}$ NP is among the most important open problems in computer science and mathematics.

- It is equivalent to determining whether or not the existence of a short solution guarantees an efficient way of finding it.
- Most people are convinced that P $\neq$ NP
  But after $\sim$50 years of effort there is still no proof.
- Resolving the question (either way) would win a prize of $1 million – see `http://www.claymath.org/millennium-problems/`

poly-time reductions amongst NP problems

Recall polynomial-time reduction.

- $\mathcal{A} \leq_p \mathcal{B}$: "$\mathcal{A}$ is poly-time reducible to $\mathcal{B}$": $\mathcal{B}$ is (in a sense) at least as hard as $\mathcal{A}$
- If we have $\mathcal{A} \leq_p \mathcal{B}$ and $\mathcal{B} \leq_p \mathcal{A}$, we can say $\mathcal{A}$ and $\mathcal{B}$ are "inter-reducible", or "polynomial-time equivalent
- Equivalence classes are partially ordered by the reduction relation.
- Problems in the maximal class are called complete for NP (we will see that there is indeed a maximal class!)

NP-Hardness and NP-Completeness

Definition.

1. A language $\mathcal{H}$ is NP-hard, if $\mathcal{L} \leq_p \mathcal{H}$ for every language $\mathcal{L} \in \mathrm{NP}$.
2. A language $\mathcal{C}$ is NP-complete, if $\mathcal{C}$ is NP-hard and $\mathcal{C} \in$ NP.

NP-Completeness:

- NP-complete problems are the hardest problems in NP.
- They are all equally difficult – an efficient solution to one would solve them all.

Lemma. If $\mathcal{L}$ is NP-hard and $\mathcal{L} \leq_p \mathcal{L}'$, then $\mathcal{L}'$ is NP-hard as well.

Proving NP-Completeness

To show that $\mathcal{L}$ is NP-complete, we must show that every language in NP can be reduced to $\mathcal{L}$ in polynomial time.

But if we know one NP-complete language $\mathcal{C}$, we can show that another language $\mathcal{L}'$ is NP-complete just by showing that

- $\mathcal{C} \leq_p \mathcal{L}'$
- $\mathcal{L}' \in \mathrm{NP}$

Hence: The problem is to find the first one (c.f. undecidable problems)

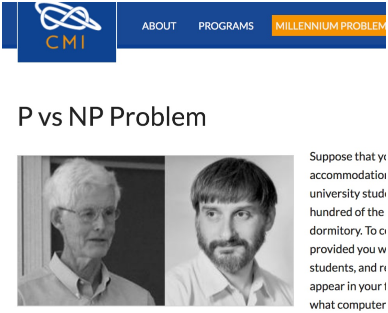
$\rightsquigarrow$ Next: the Cook-Levin Theorem

2 problems involving propositional logic

1. Given a formula φ on variables $x_1, \ldots x_n$, and values for those variables, derive the value of φ — easy!
2. Search for values for $x_1, \ldots, x_n$ that make φ evaluate to TRUE — naive algorithm is exponential: 2^n vectors of truth assignments.

Cook's Theorem (1971)
or, Cook-Levin Theorem

The second of these, called SAT, is **NP**-complete.



Stephen Cook, Leonid Levin

The challenge of solving boolean formulae

(side note:)
There's a HUGE theory literature on the computational challenge of solving various classes of syntactically restricted classes of boolean formulae, also circuits.
Likewise much has been written about their relative *expressive power*
SAT-solver: software that solves input instances of SAT — OK, so it's worst-case exponential, but aim to solve instances that arise in practice.

- "truth table" approach: clearly exponential
- DPLL algorithm; resolution: worst-case exponential, often fast in practice

Next: proof of Cook-Levin, then NP in terms of certificates, verifiers; co-NP

Reducing an **NP** problem to SAT

Goal: fixing non-deterministic TM M, integer k, given w create in poly-time a propositional formula $\mathbf{CodesAcceptRun}_M(w)$ that is satisfied by assignments that code an n^k length accepting run of M on w (where $n = |w|$)

Idea: introduce propositional variables

- $HasSymbol_{i,j}(a)$: "at time i, tape has letter a at location j"
- $HasHead_{i,j}(q)$: "at time i, TM is in location j, state q"

We'll assume M has "stay put" transitions for which it can change tape contents; R and L moves don't change tape. Assume also that to accept, M goes to LHS of tape and prints special symbol.

M has a "configuration table"

Tape space j

Time i

	1	2	$\cdots$	n^k
1	(q_0, w_1)	w_2	$\cdots$	
2	w'_1	(q_1, w_2)		
$\vdots$				
$\vdots$				
n^k				

This corresponds to a run where
$HasSymbol_{1,1}(w_1)$
$HasHead_{1,1}(q_0)$
$HasSymbol_{1,2}(w_2)$
$HasSymbol_{2,1}(w'_1)$
$HasSymbol_{2,2}(w_2)$
$HasHead_{2,2}(q_1)$
...are true
(Others, e.g. $HasHead_{1,2}(q_0)$ are false)

Idea: the search for "correct" non-deterministic choices for M shall correspond to search for satisfying assignment for $\mathbf{CodesAcceptRun}_M(w)$.
$\mathbf{CodesAcceptRun}_M(w)$ shall be a conjunction of *clauses*.

Getting started

To write the formula **$\textbf{CodesAcceptRun}_M(w)$**, let's start by writing:

$$HasSymbol_{1,j}(w_j)$$

for each $j = 1, ..., |w|$, where w_j is the j-th letter of input w, also

$$\neg HasSymbol_{1,j}(a)$$

for any a where a is *not* the j-th letter of w.

Similarly

$$HasHead_{1,1}(q_0)$$

says M is in state q_0 at time 1, location 1. Add a bunch of negated "HasHead" variables.

TM head "sanity clauses"

Include the following:

$$HasHead_{i,j}(q) \Rightarrow \neg HasHead_{i,j'}(q')$$

...for all states q, q', for all i, j, j' with $j \neq j'$.

Moving head clauses: leftward-moving State

Leftward moving state. If M has transition rule $(q, a) \to \{(q_1, a, L), (q_2, a, L)\}$ then we write:

$$HasHead_{i,j}(q) \Rightarrow [HasHead_{i+1,j-1}(q_1) \vee HasHead_{i+1,j-1}(q_2)]$$

Write the above for all $i, j \in \{1, 2, 3, \ldots, n^k\}$.

Tape space

Time

	1	$\cdots$	$j-1$	j	$\cdots$	n^k
1						
i			w_2	(q, a)		
$i+1$			$(q_1/q_2, w_2)$	a		
$\vdots$						
n^k						

Moving head clauses: Rightward-moving State or Leftward-moving State

For every rightward or leftward state q, for every a we add the clause:

$$HasSymbol_{i,j}(a) \wedge HasHead_{i,j}(q) \Rightarrow HasSymbol_{i+1,j}(a)$$

meaning: if the head is at place j at step i and we are in a rightward- or leftward moving state, symbol in place j at step $i+1$ is the same.

Tape space

	1	$\cdots$	j		$\cdots$	n^k
1						
Time i			(q,a)	w_2	$\cdots$	
$i+1$			a	(q_1,w_2)	$\cdots$	
$\vdots$						
n^k						

Moving head clauses: stay-same-place state

For every stay-and-write state q, if we have (say) transition $(q, w_0) \rightarrow \{(q_1, w_1, Stay), (q_2, w_1, Stay)\}$ then we add:

$$HasSymbol_{i,j}(w_0) \wedge HasHead_{i,j}(q) \Rightarrow HasSymbol_{i+1,j}(w_1)$$

(new symbol is written) and also:

$$HasHead_{i,j}(q) \Rightarrow [HasHead_{i+1,j}(q_1) \vee HasHead_{i+1,j}(q_2)]$$

(head does not move, although state may change)

	1	$\cdots$	j	$\cdots$	$\cdots$	n^k
1						
i			(q, w_0)	$\cdots$		
$i+1$			(q_1, w_1)	$\cdots$		
$\vdots$						
n^k						

More sub-formulae for Transitions: away from head clauses

Clauses stating that if the head is not close to place j at time i, then symbol in place j is unchanged in the next time.
For any state q and symbol w_3, any $i \leq n_k$ and number h in a certain range we have

$$HasHead_{i,j}(q) \wedge HasSymbol_{i,j+h}(w_3) \Rightarrow HasSymbol_{i+1,j+h}(w_3)$$

If q is a rightward-moving state, do this for $n^k - j \geq h \geq 2$ **and** $-(j-1) \leq h < 0$
If q is a leftward-moving state do this for $n^k - j \geq h \geq 1$ **and** $-(j-1) \leq h < -1$

If q is a stay put state, do this for $h \neq 0$

	1	$\cdots$	j	$\cdots$	$j+h$	$\cdots$	n^k
1							
i			(q, w_0)	$\cdots$	w_3		
$i+1$			(q_1, w_1)	$\cdots$	w_3		
$\vdots$							
n^k							

Reducing an **NP** problem to SAT (conclusion)

Final configuration clause: let's assume that whenever M accepts, it accepts at LHS of tape and prints special symbol $\Box$ there

$$HasSymbol_{n^k,1}(\Box) \wedge HasHead_{n^k,1}(q_{accept})$$

At time n^k, head is at the beginning and state is accepting with special termination symbol

	1	$\cdots$			$\cdots$	n^k
1	q_0	w_1	w_2	$\cdots$		
$\vdots$						
n^k	$(q_{accept}, \Box)$					

Proof of the construction (overview, not details)

We started with M, w, constructed formula **CodesAcceptRun**$_M(w)$. Two items to establish:

- **CodesAcceptRun**$_M(w)$ is constructed in polynomial time
- **CodesAcceptRun**$_M(w)$ is satisfiable iff M accepts w

For the first item, as I pointed out, many clauses were added, but polynomially-many. (large polynomial blow-up may be counter-intuitive)

For the second, the main point is that an accepting run gives rise to a satisfying assignment of the formula (and vice versa) in a direct way, according to our understanding of what the **HasHead** and **HasSymbol** variables mean, for runs of M.

Certificates

Every yes-instance of such problems has a short and easily checkable certificate that proves it is a yes-instance.

- SAT – a satisfying assignment
- *k*-COLOURABILITY – a *k*-colouring
- HAMILTONIAN CIRCUIT – a Hamiltonian circuit
- TSP (decision-problem version) – a round trip (i.e. permutation)

Verifiers

Definition.

1. A Turing acceptor M which halts on all inputs is called a verifier for language $\mathcal{L}$ if

$$\mathcal{L} = \{w : M \text{ accepts } \langle w, c\rangle \text{ for some string } c\}$$

   The string c is called a certificate (or witness) for w.

2. A polynomial time verifier for $\mathcal{L}$ is a polynomially time bounded Turing acceptor M such that

$$\mathcal{L} = \{w : M \text{ accepts } \langle w, c\rangle \text{ for some string } c \text{ with } |c| \leq p(|w|)\}$$

   for some fixed polynomial $p(n)$.

All problems for the previous slide have verifiers that run in polynomial time.

Equivalent definition of **NP**

The class of languages that have polynomial-time verifiers

Examples.

- SAT is in **NP**

  For any formula that can be satisfied, the satisfying assignment can be used as a certificate.

  It can be verified in polynomial time that the assignment satisfies the formula.

- k-COLOURABILITY is in **NP**

  For any graph that can be coloured, the colouring can be used as a certificate.

  It can be verified in polynomial time that the colouring is a proper colouring.

A Problem (probably) not in **NP**

NO HAMILTONIAN CYCLE	
Input:	A graph G
Question:	Is it true that G has no Hamiltonian cycle?

Note. Whereas it is easy to certify that a graph has a Hamiltonian cycle, there does not seem to be a (general purpose) certificate that it has not.

co-NP

co-NP problem: complement of an NP problem
In a co-NP problem, no-instances have (concise) certificates
Believed that NP is not equal to co-NP

The following result justifies guess and test approach to establishing membership of NP:

NP as languages having concise certificates

Theorem. NP as just defined, is languages having concise certificates

Proof. Suppose $\mathcal{L} \in \mathrm{NP}$.

Hence, there is an NTM M such that

$$w \in \mathcal{L} \iff \text{there is an accepting run of } M \text{ of length } \leq n^k$$

for some k. This path can be used as a certificate for w

(A DTM can check in polynomial time that a candidate for a certificate is a valid accepting computation path.)

NP as languages having concise certificates

Theorem. NP as just defined, is languages having concise certificates

Proof. Suppose $\mathcal{L} \in \text{NP}$.

Hence, there is an NTM M such that

$$w \in \mathcal{L} \Longleftrightarrow \text{there is an accepting run of } M \text{ of length} \leq n^k$$

for some k. This path can be used as a certificate for w

(A DTM can check in polynomial time that a candidate for a certificate is a valid accepting computation path.)

Conversely: If $\mathcal{L}$ has a polynomial-time verifier M, say of length at most n^k,

then we can construct an NTM M^* deciding $\mathcal{L}$ as follows:

1. M^* guesses a string of length $\leq n^k$
2. M^* checks in deterministic polynomial-time if this is a certificate.