Introduction to Computer Science
Lecture 11: Theory of Computation

Tian-Li Yu

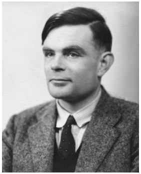
Taiwan Evolutionary Intelligence Laboratory (TEIL)
Department of Electrical Engineering
National Taiwan University
tianliyu@cc.ee.ntu.edu.tw

Slides made by Tian-Li Yu, Jie-Wei Wu, and Chu-Yu Hsu

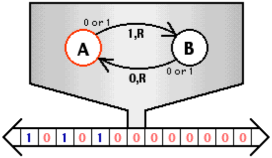
Computability

- Well-defined input and output
- Computation of these functions lies beyond any algorithmic system $\rightarrow$ noncomputable.
- Hold it... but algorithms are defined on some particular primitives, and primitives are defined on some particular machine.
- We need a universal machine to define computation.

Turing Machine



- Alan Turing, 1936
- Finite state automata + infinite R/W tape
  - Finite states.
  - A tape with infinite cells.
  - R/W head moving one cell per time (left/right).
  - Finite alphabet (0,1,*).



Incrementing a Value

Current state	Current cell content	Value to write	Direction to move	New state to enter
START	*	*	Left	ADD
ADD	0	1	Right	RETURN
ADD	1	0	Left	CARRY
ADD	*	*	Right	HALT
CARRY	0	1	Right	RETURN
CARRY	1	0	Left	CARRY
CARRY	*	1	Left	OVERFLOW
OVERFLOW	*	*	Right	RETURN
RETURN	0	0	Right	RETURN
RETURN	1	1	Right	RETURN
RETURN	*	*	No move	HALT

Church-Turing Thesis

The functions that are computable by a Turing machine are exactly the functions that can be computed by any algorithmic means.

Bare Bones Language

- One of the universal programming languages
  - Simple imperative programming languages
  - Rich enough to compute all Turing-computable functions
  - Bare bones → minimal set

- clear name;
- incr name;
- decr name; /* **remains 0 if already 0 !!!***/
- while name not 0 do; ... end;

Examples

```
clear Z;
while X not 0 do;
   clear W;
   while Y not 0 do;
      incr Z;
      incr W;
      decr Y;
   end;
   while W not 0 do;
       incr Y;
       decr W;
    end;
    decr X;
end;
```

```
clear Aux;
clear Tomorrow;
while Today not 0
do;
    incr Aux;
    decr Today;
end;
while Aux not 0 do;
    incr Today;
    incr Tomorrow;
    decr Aux;
end;
```

The Halting Problem

- Are all algorithms (functions) computable?

- **Input:** encoding of a program.
- **Output:** 1 if the program halts; 0 otherwise.

- Is it possible to write such an algorithm?
  - Suppose $S(p)$ is such an algorithm.
  - $S(p)$ returns 1 if p halts.
  - $S(p)$ returns 0 if p doesn't halt.

1^{st} known incomputable problem

Proof (Short, Conceptual Version)

$S(p)$: The solution to the halting problem

$N(p)$

1. $x = S(p)$
2. **while** x **not** 0 **do**
3. **end**

Does $N(N)$ halt?

If $N(N)$ halts $\rightarrow$ $S(N)$ returns 1 $\rightarrow$ $N(N)$ does not halt

If $N(N)$ doesn't halt $\rightarrow$ $S(N)$ returns 0 $\rightarrow$ $N(N)$ halts

Gödel Number & Incomplete Theory

- All Turing machines (computable functions) can be mapped (1-to-1) to natural numbers.
  - The set of Turing machines is countable infinite.
  - The number is called the Gödel number.
- Gödel's incomplete theory (Kurt Gödel, 1931)
  - Later used by Turing.
  - "Any effectively generated theory capable of expressing elementary arithmetic cannot be both consistent and complete.
  - In particular, for any consistent, effectively generated formal theory that proves certain basic arithmetic truths, there is an arithmetical statement that is true, but not provable in the theory."

Halting Problem: 1^{st} Incomputable

- Is the following function computable?
  - x and i are integers.

Procedure $g(i)$

1. **if** $h(i,i) == 0$
2. **return**
3. **else**
4. loop forever

$$h(x,i) = \begin{cases} 1, & \text{if program } x \text{ halts on input } i \\ 0, & \text{otherwise} \end{cases}$$

Let g's Gödel number be e

- Diagonalization proof
  - $h(e,e) = 0 \rightarrow g$ doesn't halts on $e \rightarrow$ but g actually halts.
  - $h(e,e) = 1 \rightarrow g$ halts on $e \rightarrow$ but g actually doesn't halts.

Diagonalization Proof

$h(x,i)$		Procedure x				
		1	2	3	4	5
Input i	1	1	0	1	0	1
	2	1	1	0	0	0
	3	0	0	0	1	1
	4	1	1	0	1	0
	5	0	0	1	1	0

$h(i,i)$	1	1	0	1	0
$g(i)$: halt:1, otherwise:0	0	0	1	0	1

Invert the diagonal. So g can not be any procedure x.

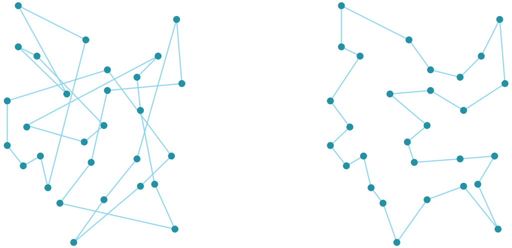
Complexity Classes

- Developed by Cook & Karp in early 70.
- The class $\mathcal{P}$: class of problems that can be solved in polynomial time in the size of input.
  - Problems in $\mathcal{P}$ is considered tractable.
  - Closed under addition, multiplication, composition, complement, etc. (closure property).
- The class $\mathcal{NP}$ (Nondeterministic Polynomial)
  - Polynomial time in the size of input on a nondeterministic Turing machine (nondeterministic finite state automata + infinite tape)

$\mathcal{P}$ vs. $\mathcal{NP}$

- Finding max $\rightarrow \Theta(n)$
- Sorting $\rightarrow \Theta(n \log n)$

- Traveling salesman problem (TSP) $\rightarrow \Theta(n^n)$?



Traveling Salesman Problem

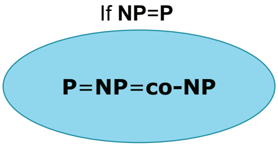
- Traveling salesman problem (TSP)
  - Instance: A set of n cities, distance between each pair of cities, and a bound B.
  - Question: Is there a route that starts and ends at a given city, visits every city exactly once, and has total distance $\leq B$?
- TSP $\in \mathcal{NP}$?
  - Guess a tour, verify if it visits every city exactly once, returns to the start, and total distance $\leq B$.
- co-TSP
  - Are all tours that start and end at a given city, visit every city exactly once, and have total distance $> B$?

Subset Sum Problem

- Subset sum problem (SSP)
  - Given a finite set of integers, is there a non-empty subset which sums to 0?
- SSP $\in \mathcal{NP}$?
  - Guess a set (certificate), verify if it is a subset and sums to 0.
- co-SSP
  - Yes/No $\rightarrow$ No/Yes
  - Does every non-empty subset have a nonzero sum?

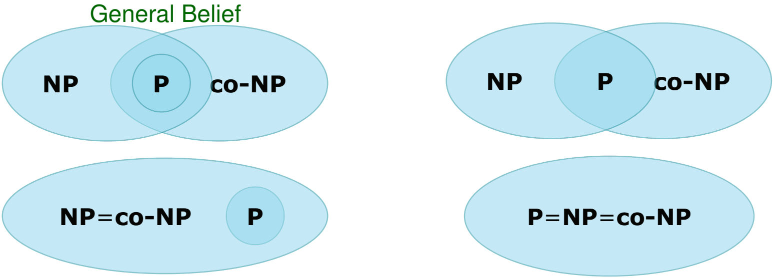
Properties of $\mathcal{NP}$

- All problems in $\mathcal{P}$ are also in $\mathcal{NP}$.
  - $\mathcal{P} \subseteq \mathcal{NP}$
  - $\mathcal{P} = \mathcal{NP}$? No one knows yet. A 7-million dollar question.
- Solutions to problems in $\mathcal{NP}$ can be verified in polynomial time in the size of input.
- $\mathcal{NP}$ is not known to be closed under complement.
  - co-$\mathcal{NP}$
  - $x \in$ co-$\mathcal{NP}$ iff "complement of x" $\in \mathcal{NP}$



$\mathcal{NP}$, co-$\mathcal{NP}$, and $\mathcal{P}$

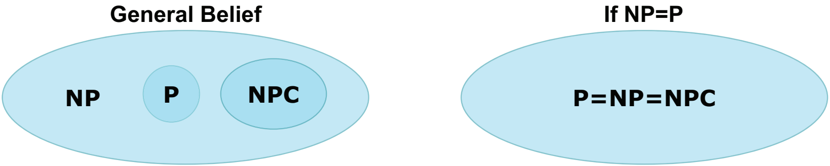
- All these are possible.



- In 2002, a survey of 100 researchers
  - 61 think No, 9 think Yes, 22 uncertain, 8 think impossible to prove.

$\mathcal{NP}$-Completeness

- The class $\mathcal{NP}$-complete ($\mathcal{NPC}$)
  - Intuitively, if any $\mathcal{NPC}$ problem can be solved in polynomial time $\Rightarrow$ All problems in $\mathcal{NP}$ can be solved in polynomial time.



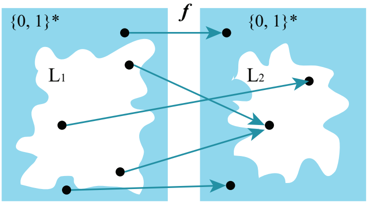
$\mathcal{NPC}$

- Intuitively, $\mathcal{NPC}$ are problems that are the most difficult ones in $\mathcal{NP}$.
- How do we define "difficulty" when we don't know their complexity?
- Key: reduction

Polynomial-Time Reduction

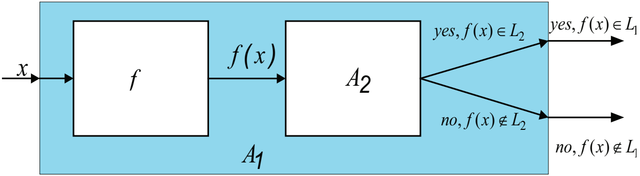
- Motivation:
  - Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 can solve L_2. Can we use A_2 to solve L_1?

- Polynomial-time reduction f from L_1 to L_2: $L_1 \leq_{\mathcal{P}} L_2$
  - x is an "yes" input for L_1 iff $f(x)$ is an yes input for L_2.
  - f is $\mathcal{P}$-time computable.
  - L_1 is $\mathcal{P}$-time reducible to L_2
  - L_2 is at least as hard as L_1
  - f is reduction function.



Significance of Reduction

- $L_1 \leq_{\mathcal{P}} L_2$ implies
  - $\exists\, \mathcal{P}$-time algorithm for $L_2 \rightarrow \exists\, \mathcal{P}$-time algorithm for L_1 ($L_2 \in \mathcal{P} \rightarrow L_1 \in \mathcal{P}$)
  - No $\mathcal{P}$-time algorithm for $L_1 \rightarrow$ no $\mathcal{P}$-time algorithm for L_2 ($L_1 \notin \mathcal{P} \rightarrow L_2 \notin \mathcal{P}$)



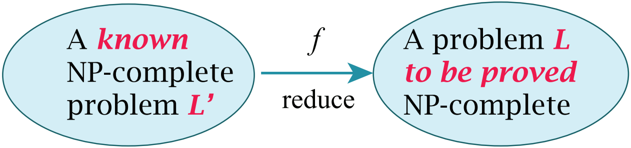
- $\leq_{\mathcal{P}}$ is transitive, *i.e.*, $L_1 \leq_{\mathcal{P}} L_2$ & $L_2 \leq_{\mathcal{P}} L_3 \Rightarrow L_1 \leq_{\mathcal{P}} L_3$

Definition of $\mathcal{NPC}$, $\mathcal{NP}$-Hard

- $L \in \mathcal{NPC}$ iff
  - $L \in \mathcal{NP}$ and $\forall L' \in \mathcal{NP}, L' \leq_p L$
- $L \in \mathcal{NP}$-hard iff
  - $\forall L' \in \mathcal{NP}, L' \leq_P L$
- To prove a problem is $\mathcal{NPC}$, we need one very first $\mathcal{NPC}$ problem and then use $\mathcal{P}$-reduction.
- Now, it's easily seen that the optimization version of a $\mathcal{NPC}$ problem is $\mathcal{NP}$-hard.

Proving $\mathcal{NP}$-Completeness

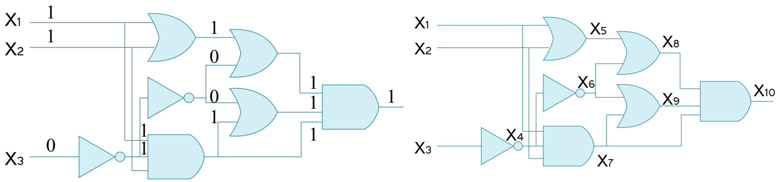
- Five steps for proving that L is $\mathcal{NPC}$:
  - Prove $L \in \mathcal{NP}$.
  - Choose a known $\mathcal{NPC}$ problem L'.
  - Construct a reduction f transforming every instance of L' to an instance of L.
  - Prove that $x \in L'$ if $f(x) \in L$ for all x.
  - Prove that f is polynomial-time computable.



1^{st} $\mathcal{NPC}$ Problem

- Circuit-SAT (Stephen Cook, 1971)
  - Probably the 1^{st}. He proved 21 $\mathcal{NPC}$ problems in the same paper.
  - Instance: A combinational circuit C composed of AND, OR, and NOT gates.
  - Question: Is there an assignment of Boolean values to the inputs that makes the output of C to be 1?
- Satisfiability (SAT) (Stephen Cook, 1971)
  - Determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE.

Circuit-SAT $\leq_{\mathcal{P}}$ SAT

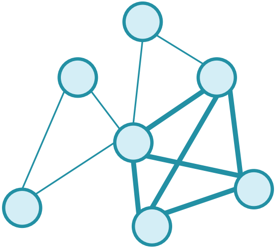


$$\varphi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \wedge (x_6 \leftrightarrow \neg x_4)$$
$$\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \wedge (x_8 \leftrightarrow (x_5 \vee x_6))$$
$$\wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

1. SAT $\in \mathcal{NPC}$
2. Circuit C is satisfiable iff φ is satisfiable
3. φ is $\mathcal{P}$-time constructible and maps every instance.

Clique

- A clique in G is a complete subgraph of G.
- The clique problem
  - Instance: $G = (V, E)$ and a positive integer $k \leq |V|$.
  - Question: Is there a clique $V' \subseteq V$ of size $\geq k$?
- Clique $\in \mathcal{NP}$
  - Can be verified in $O(k^2)$ time.



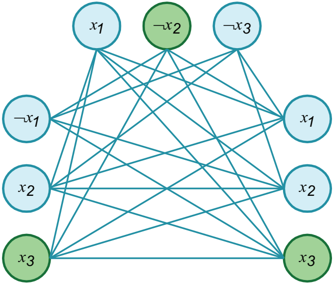
3SAT $\leq_{\mathcal{P}}$ Clique

- Let $\varphi = C_1 \wedge C_2 \wedge ... \wedge C_k$ be a Boolean formula in 3-CNF with k clauses.
- For each $C_r = (l_1^r \vee l_2^r \vee l_3^r)$, introduce a triple of vertices v_1^r, v_2^r, v_3^r in V.
- Build an edge between v_i^r, v_j^s if both of the following hold:
  - v_i^r, v_j^s are in different triples ($r \neq s$)
  - l_i^r is not the negation of l_j^s
- Claim: G can be computed from φ in $\mathcal{P}$-time.

Reduction Example

$$\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$

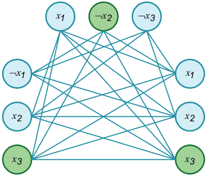


$C_2 = \neg x_1 \vee x_2 \vee x_3$ $\quad$ $C_3 = x_1 \vee x_2 \vee x_3$

φ Satisfiable $\Leftrightarrow$ G Has a k-Clique

- φ satisfiable $\Rightarrow$ each C_r contains at least one $l_i^r = 1$ and each such literal corresponds to a vertex v_i^r.
- Picking a "true" literal from each C_r forms a set of V' of k vertices.
- For any two vertices $v_i^r, v_j^s \in V', r \neq s, l_i^r = l_j^s = 1$ and thus l_i^r, l_j^s cannot be complements. $\Rightarrow$ edge $(v_i^r, v_j^s) \in E$.

$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$



$C_2 = \neg x_1 \vee x_2 \vee x_3$ $\quad$ $C_3 = x_1 \vee x_2 \vee x_3$

Coping with $\mathcal{NP}$-Complete/-Hard

- Approximation algorithms:
  - Guarantee to be “not-too-bad.”
- Pseudo-polynomial time algorithms:
  - *e.g.*, DP for the 0-1 Knapsack problem.
- Probabilistic algorithms:
  - Assume some probabilistic distribution of the instances.
- Randomized algorithms/heuristics:
  - Make use of a randomizer/heuristic:
  - No guarantee of performance.
  - Simulated annealing, genetic algorithms, *etc*.
- $\mathcal{EXP}$-algorithms/branch & bound/exhaustive:
  - Feasible only when the problem is small.