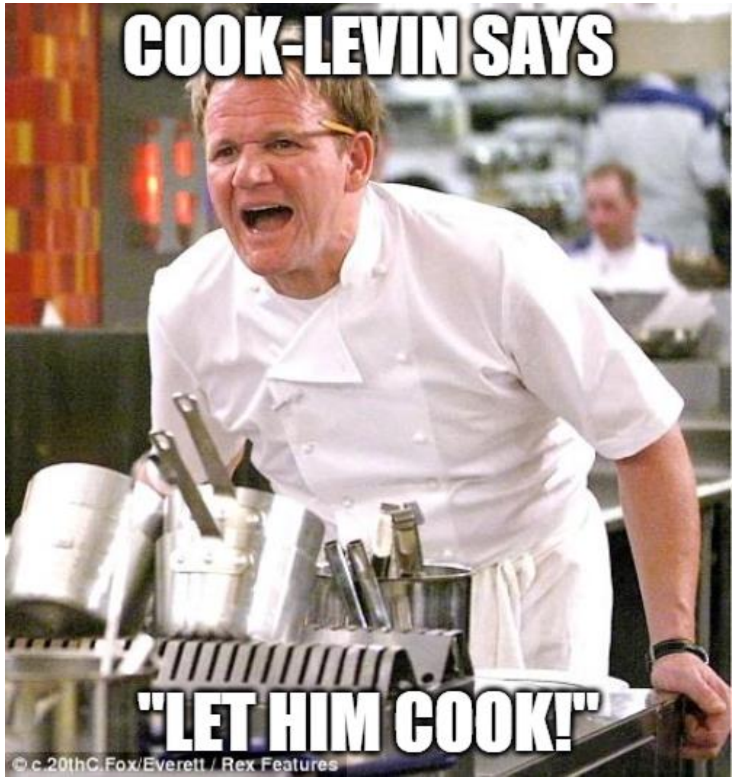
D8: NP-Completeness



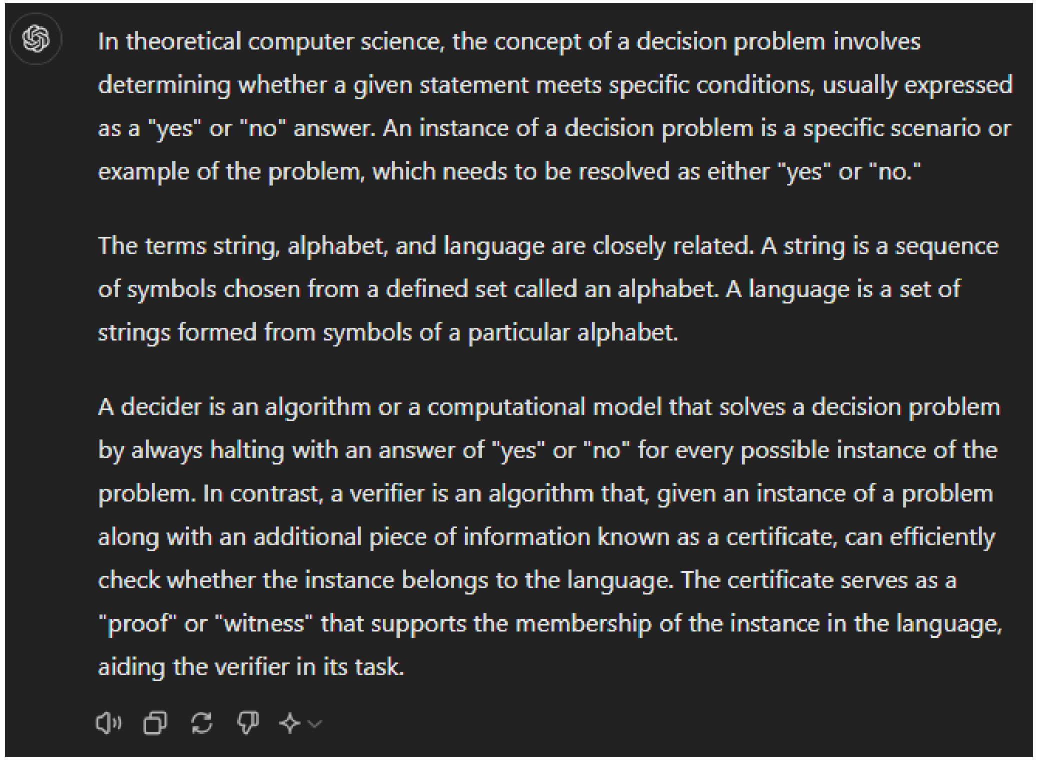
Sec 101: MW 3:00-4:00pm DOW 1018
IA: Eric Khiu

Announcement

- Thanks for the midterm feedback!
- Midterm Regrade Request open till next Monday

Vocabulary Checkpoint

- In your own words, relate the following concepts in a few sentences:
  - Decision problem
  - String
  - Alphabet
  - Language
  - Instance
  - Decider
  - Verifier
  - Certificate



In theoretical computer science, the concept of a decision problem involves determining whether a given statement meets specific conditions, usually expressed as a "yes" or "no" answer. An instance of a decision problem is a specific scenario or example of the problem, which needs to be resolved as either "yes" or "no."

The terms string, alphabet, and language are closely related. A string is a sequence of symbols chosen from a defined set called an alphabet. A language is a set of strings formed from symbols of a particular alphabet.

A decider is an algorithm or a computational model that solves a decision problem by always halting with an answer of "yes" or "no" for every possible instance of the problem. In contrast, a verifier is an algorithm that, given an instance of a problem along with an additional piece of information known as a certificate, can efficiently check whether the instance belongs to the language. The certificate serves as a "proof" or "witness" that supports the membership of the instance in the language, aiding the verifier in its task.

Agenda

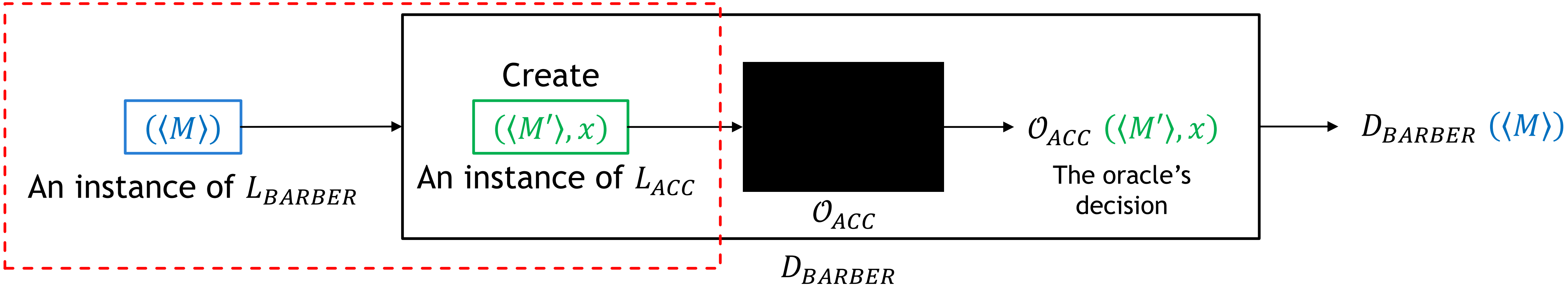
- Mapping Reductions
- NP-hardness and NP-Completeness
- Worksheet problems

Mapping Reductions

Notes

Recap: Notion of Transforming a Problem

- Recall that in Turing Reductions, we “transform” an instance of a decision problem into an instance of another decision problem whose black-box (oracle) decider is available

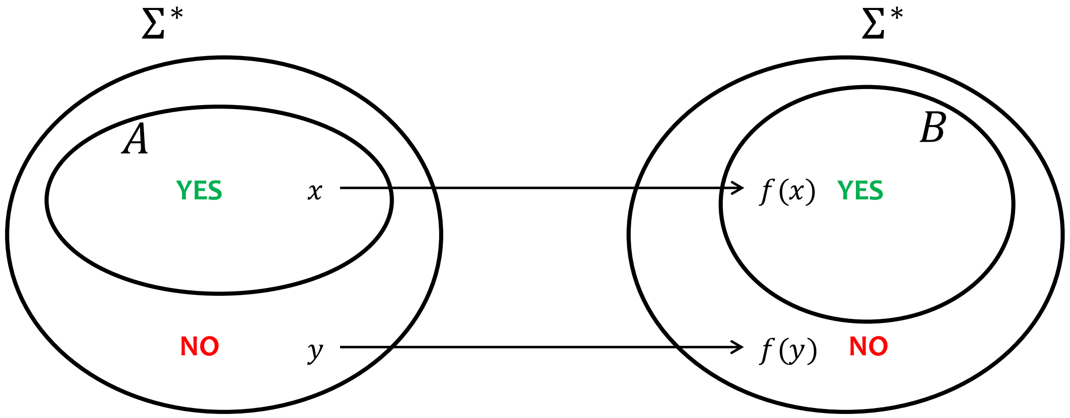


$(\langle M \rangle)$
An instance of L_{BARBER}
Create
$(\langle M' \rangle, x)$
An instance of L_{ACC}
$\mathcal{O}_{ACC}$
$\mathcal{O}_{ACC}\ (\langle M' \rangle, x)$
The oracle’s decision
$D_{BARBER}\ (\langle M \rangle)$
D_{BARBER}

- Now, we focus on the reduction part of this process, i.e., how we map an instance of problem A to an instance of problem B

Mapping Reductions

- **Definition:** A mapping reduction from one language to another maps yes instances to yes instances and no instances to no instances



- **Note:** These reductions do not attempt to decide the problem (i.e. accept or reject), they only map one language to another

Discuss: Compare and contrast Turing Reductions vs Mapping Reductions

Polynomial-Time Mapping Reduction

- Recall that in EECS 203, we define a function as a machine where you can input an element from one given set (the domain) and it outputs an element from another given set (the codomain)
- Now, consider the function whose
  - domain = set of instance in language A
  - codomain = set of instances in language B
- **Definition:** Language A is polynomial time mapping reducible to language B, written as

$$A \leq_p B,$$

if there exits a polynomial time computable function f such that for every w

$$w \in A \Leftrightarrow f(w) \in B$$

- The function f is called the polynomial-time reduction from A to B

Side: Iff and Contrapositive

- To show that $w \in A \Leftrightarrow f(w) \in B$, we need to show two directions
  - $w \in A \Rightarrow f(w) \in B$
  - $f(w) \in B \Rightarrow w \in A$
- For the second direction, we can also proof the contrapositive, which means
  - $w \in A \Rightarrow f(w) \in B$
  - $w \notin A \Rightarrow f(w) \notin B$
- Does this look familiar?
  - Yes! Just like the correctness proof for Turing reductions
  - In fact, Turing reduction is just a special type of mapping reduction

Very Important Theorem 2

Definition: Language A is polynomial time mapping reducible to language B, written as

$$A \leq_p B,$$

if there exits a polynomial time computable function f such that for every w

$$w \in A \Leftrightarrow f(w) \in B$$

- Suppose $A \leq_p B$. If $B \in \mathrm{P}$, then $A \in \mathrm{P}$
  - Contrapositive: If $A \notin P$, then $B \notin P$ (Warning: This is different from if $A \in NP$ then $B \in NP$!)
- **Proof:** Let D_B be an efficient decider that decides B, consider the following decider for A

```
$D_A$ = "On input $w$:
    Compute $f(w)$
    Run $D_B$ on $f(w)$ and return the same"
```

- Correctness:
  - $w \in A \Rightarrow f(w) \in B \Rightarrow D_B\big(f(w)\big)$ accepts $\Rightarrow D_A(w)$ accepts
  - $w \notin A \Rightarrow f(w) \notin B \Rightarrow D_B\big(f(w)\big)$ rejects $\Rightarrow D_A(w)$ rejects
- Runtime:
  - $A \leq_p B \Rightarrow f(w)$ can be computed in polynomial time w.r.t. size of w
  - $B \in P \Rightarrow$ There exists an efficient decider for B that we can use as D_B

Discuss: Are you sure D_A is always efficient? What if $|f(w)|$ is $\Omega\big(2^{|w|}\big)$?

This will never happen because the size of $f(w)$ is beyond polynomial of the size of w (since f is a poly-time reduction, recall how TM writes to the tape)

Example: SAT $\leq_p L_{HALT}$

- Describe a polynomial-time reduction f from SAT to L_{HALT}
- **Step 1: Determine the format of an instance for SAT and L_{HALT}**
  - SAT: A Boolean formula ϕ with n variables
  - L_{HALT}: A machine-string pair $(\langle M\rangle, x)$
- **Step 2: Desired behavior for f**
  - $\phi \in SAT \Rightarrow \cdots \Rightarrow f(\phi) = (\langle M\rangle, x) \in L_{HALT}$
  - $\phi \notin SAT \Rightarrow \cdots \Rightarrow f(\phi) = (\langle M\rangle, x) \notin L_{HALT}$
- **Brainstorm:** What can we do to check if $\phi \in SAT$? (Hint: Think naively)
  - Try out every possible assignment!

Example: $\mathrm{SAT} \leq_p L_{HALT}$

Step 2: Desired behavior for f

- $\phi \in SAT \Rightarrow \cdots \Rightarrow f(\phi) = (\langle M \rangle, x) \in L_{HALT}$
- $\phi \notin SAT \Rightarrow \cdots \Rightarrow f(\phi) = (\langle M \rangle, x) \notin L_{HALT}$

- **Step 3: Construct a TM for f**

```
$f$ = "On input $\phi$:
        Construct a TM $M$ as follows:
        $M$ = "On input $w$:
                Ignore $w$
                for each possible assignment $a$ of $\phi$ do
                    if $\phi$ evaluates to be true with assignment $a$ then accept
                loop"
        return $(\langle M \rangle, \varepsilon)$ // Doesn't have to be $\varepsilon$, could be anything
```

Assignment: An array of truth values where $a[i]$ is assigned to variable x_i

Discuss: How is this efficient? Doesn't looping through each assignment a of ϕ takes $O(2^n)$?

Similar to Turing reductions, f doesn't run M here (nor generating all possible assignments)! It just write the code for M and return its description (along with ε)

- $\phi \in SAT \Rightarrow \exists a$ such that $\phi(a)$ evaluates to be true $\Rightarrow M$ halts on $\varepsilon \Rightarrow f(\phi) = (\langle M \rangle, x) \in L_{HALT}$
- $\phi \notin SAT \Rightarrow \nexists a$ such that $\phi(a)$ evaluates to be true $\Rightarrow M$ loops on $\varepsilon \Rightarrow f(\phi) = (\langle M \rangle, x) \notin L_{HALT}$

TL;DPA

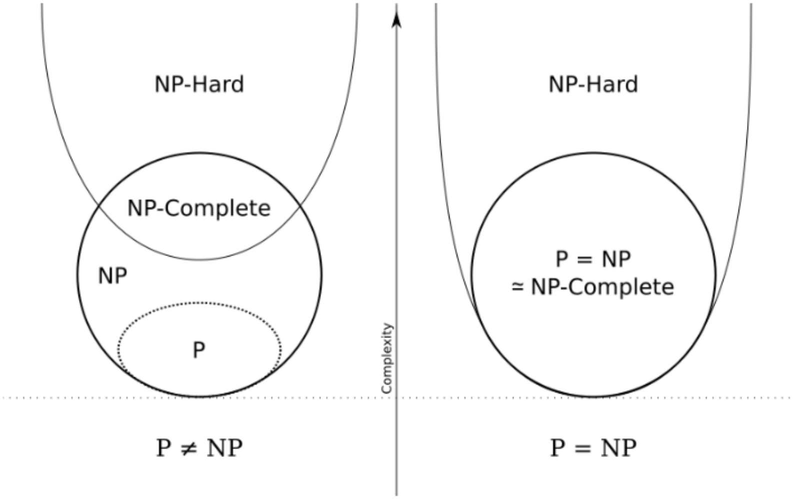
- We discussed *mapping-reduction*: a function to transform an instance of A to an instance of B such that $w \in A \Leftrightarrow f(w) \in B$
  - Two directions: $w \in A \Rightarrow f(w) \in B$ and $f(w) \in B \Rightarrow w \in A$
  - Second direction as contrapositive: $w \notin A \Rightarrow f(w) \notin B$
- Very Important Theorem 2: If $A \leq_p B,\ B \in P \Rightarrow A \in P$
  - Contrapositive: $B \notin P \Rightarrow A \notin P$

NP-Hardness & NP-Completeness

Notes

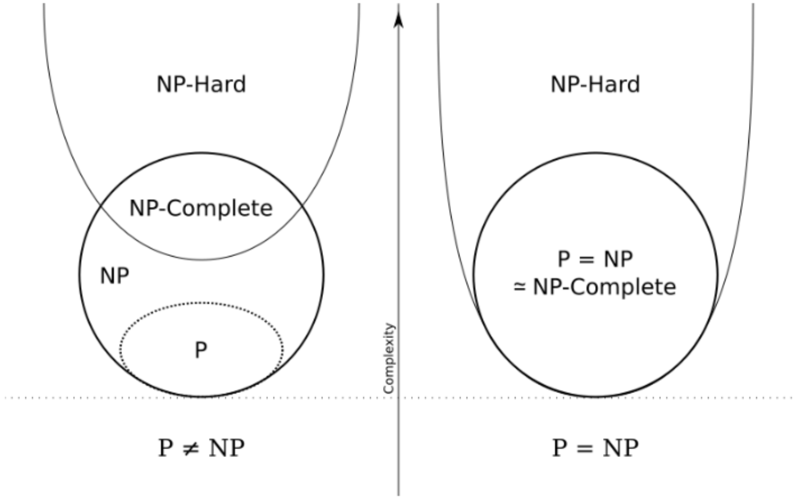
NP-Hardness & NP-Completeness

- **Definition:** A language L is NP-hard if $A \leq_p L$ for every language $A \in NP$
  - NP-Hard languages are “at least as hard as” every language in NP
  - We define NPH as the set of NP-hard languages
- **Theorem (Cook-Levin Core):** SAT is NP-Hard
- **Definition:** A language L is NP-complete if $L \in NP$, and and L is NP-hard
  - We define NPC as the set of NP-complete languages



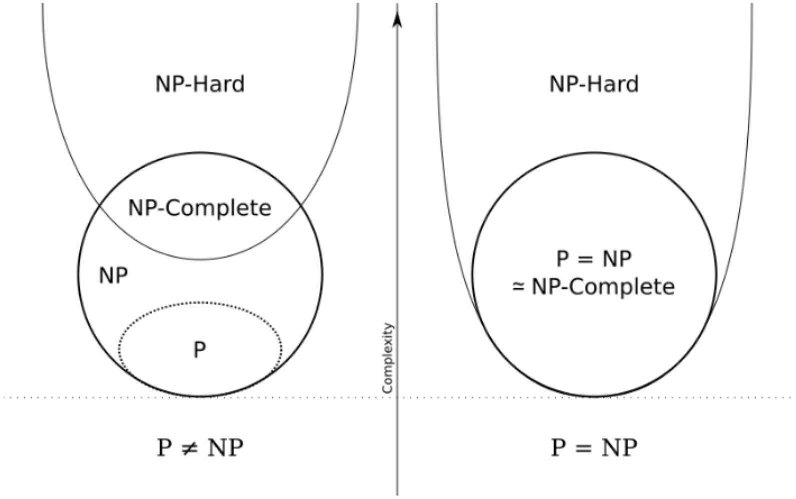
Very Important Theorem 3

- Suppose $A \leq_p B$. If $A \in NPH$, then $B \in NPH$
    - Contrapositive: If $B \notin NPH$, then $A \notin NPH$ (Warning: This is different from if $B \in P$ then $A \in P$)!
- **Corollary:** Suppose $A \leq_p B$. If $A \in NPH$ and $B \in NP$, then $B \in NPC$
    - This is a very powerful result that holds regardless of whether $P = NP$!



P vs NP Revisit

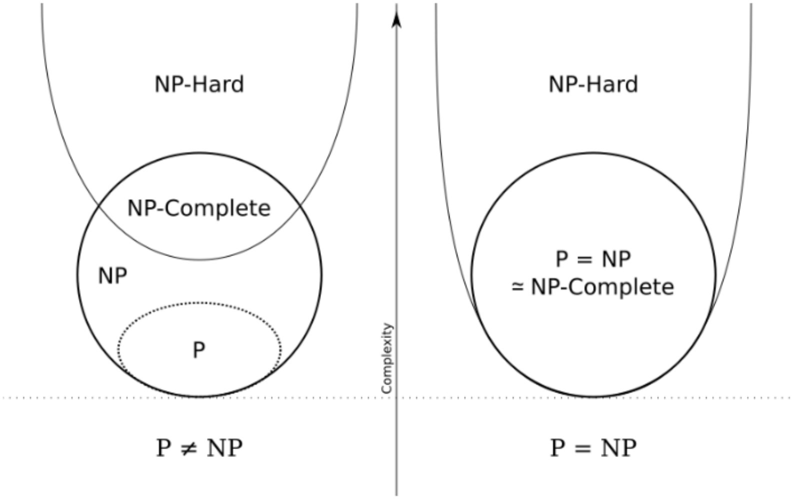
- Are you interested to solve P vs NP? Here's one direction
- **Claim:** If any NP-Complete language is proven to be in P, then P = NP
  - Let $L \in NPC$ and suppose that $L \in P$
  - Since $L \in NPC$, $L \in NPH$
  - By definition of NP-hardness, $A \leq_p L$ for all $A \in NP$
  - By Very Important Theorem 2, if $L \in P$, then $A \in P$
  - So for all $A \in NP$, we have $A \in P$
  - Therefore, $P = NP$



Decidability Revisit

- (All/ some/ no) languages in NP is/are decidable
- (All/ some/ no) languages in NPH is/are decidable
- (All/ some/ no) languages in NPC is/are decidable

- Answer: All, some, no
  - $NP \subseteq EXP$ (decidable in exponential time)
  - $NPC = NP \cap NPH$



TL; DPA

- We discussed the notion of NP-hardness and NP-Completeness.
- Very Important Theorem 3: Suppose $A \leq_p B$. If $A \in NPH$, then $B \in NPH$
  - Corollary: Suppose $A \leq_p B$. If $A \in NPH$ and $B \in NP$, then $B \in NPC$
- If we can prove *any* NP-C language to be in P, then P = NP

Worksheet Problems

WS#8: k-$CLIQUE \leq_P IS$

- Recall that
  - A clique of a graph is a subset of vertices that form a fully-connected (sub-)graph
  - An independent set of a graph is a subset of vertices such that no two vertices in the subset are connected
- Consider the following languages:

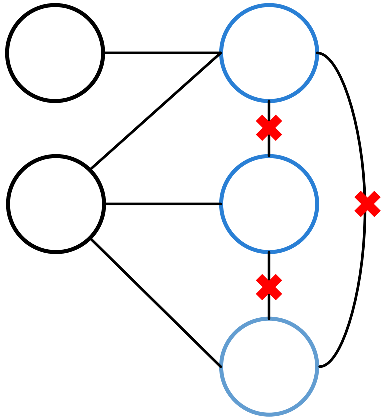
$$k{-}CLIQUE = \{(G = (V, E), k) \colon G \text{ is a graph with a clique of size } k\}$$

$$IS = \{(G = (V, E), K) \colon G \text{ is a graph with independent set of size } k\}$$

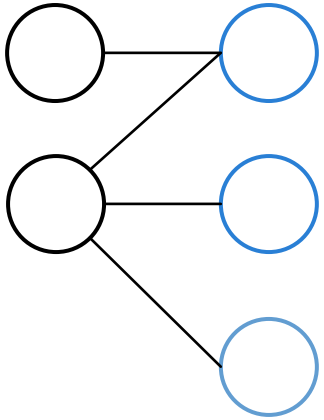
Prove that $k{-}CLIQUE \leq_P IS$.

Hint: How do you make the k vertices that form the k-clique to be an independent set in the new graph (this way you ensure the independent set has k vertices)?

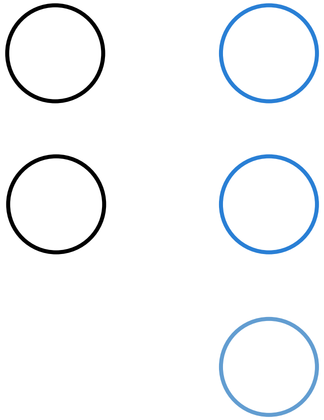
Cooking Time



The three blue vertices form a k-clique ($k = 3$)

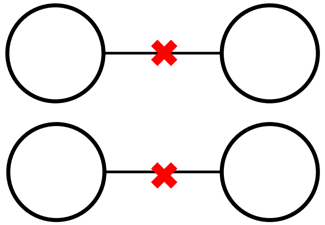


The three blue vertices form an Independent set of size k



Maybe… remove all edges? The three blue vertices still form an IS

This graph doesn't have a k-clique with $k = 3$



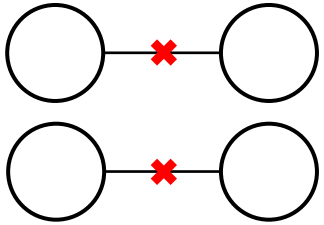
But removing all edges creates an IS of size k

Idea: Remove edges to transform a clique to an IS

Problem: The function f doesn't know which vertices belongs to the k-clique!

Now every graph with $\geq k$ vertices will have IS of size k! (regardless of whether the original graph has a k-clique)
BAD

Cooking Time

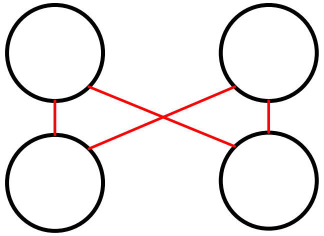


I want to remove edges to transform a clique (if any) to an IS

I don't know which edge to remove so I remove all edges

But I can't let *any* graph to have IS of size k after the transformation

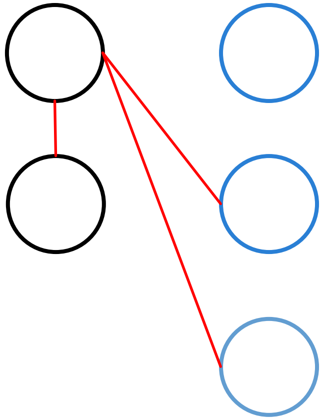
Idea: Need to add edges to prevent the vertices that aren't in the k-clique to form an IS in the new graph



Again, I don't know which vertices aren't in the k-clique.

So, … let's connect every pair of vertices that aren't connected in the original graph

Seems like it works! Let's apply it to the first graph



The three blue vertices still form an IS!

Strategy: Disconnect pairs that are connected in G; connect pairs that aren't connected in G

WS#8: k-$CLIQUE \leq_P IS$ Solution

- Mapping Reduction

```
f = "On input $(G = (V, E), k)$:
        $E' \leftarrow \emptyset$
        for each pair of vertices $u, v \in V$ do
            if $(u, v) \notin E$ then
                add $(u, v)$ to $E'$ // Connect $u, v$ in $G'$ iff they are not connected in $G$
        $G' \leftarrow (V, E')$
        $k' \leftarrow k$
        return $(G' = (V, E'), k')$"
```

- **Runtime:** Looping through all pairs of vertices takes $O(|V|^2)$; scanning if $(u, v) \in E$ takes $O(|E|)$. Therefore, the overall runtime is efficient w.r.t. the input size

WS#8: $k\text{-}CLIQUE \leq_P IS$ Correctness Analysis

$(\Rightarrow)$ If $(G, k) \in k\text{-}CLIQUE$, then $f(G, k) \in IS$

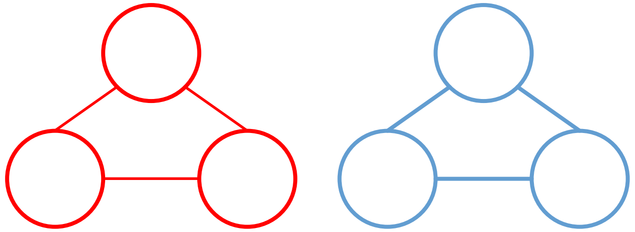
- Assume G has a clique C of size k
  - G' is a complement of G, so the edges of vertices in C are not in G'
  - These vertices in C then correspond to an independent set of size $k' = k$
  - Thus, $f(G, k) \in IS$

$(\Leftarrow)$ If $f(G, k) \in IS$, then $(G, k) \in k\text{-}CLIQUE$

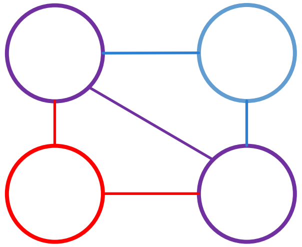
- Assume G' has an independent set S of size $k' = k$
  - G' has no edge between any pair of vertices in S
  - G is the complement of G', so it must have an edge between each of these pairs
  - S constitutes a clique of size k in the original graph, so $(G, k) \in k\text{-}CLIQUE$

WS#7: $HAM\text{-}CYCLE \leq_P BICYCLE$

- Recall that a Hamiltonian cycle is a cycle that visits each vertex exactly once.
- An undirected graph G has a *bicycle* if it has two non-intersecting cycles of equal size that together includes all the vertices.



This graph has a bicycle



This graph does not have a bicycle

- Consider the languages

$$HAM\text{-}CYCLE = \{\text{G: G is graph with a Hamiltonian cycle}\}$$
$$BICYCLE = \{G\text{: G is graph with a bicycle}\}$$

- Prove that $HAM\text{-}CYCLE \leq_P BICYCLE$ (Hint: See examples above)

WS#7: $HAM\text{-}CYCLE \leq_P BICYCLE$ Solution

- Mapping Reduction

  f = "On input $(G = (V, E))$:

  output two copies of G"

- **Runtime:** f is clearly efficient because it just outputs two copies of the input.
- **Correctness:**
  - (⇒) $G \in HAM\text{-}CYCLE$ ⇒ Two copies of G each form a cycle of size $|V|$ ⇒ $G' \in BICYCLE$
  - (⇐) $G' \in BICYCLE$ ⇒ By construction, the two cycles are not connected in any way ⇒ It must be that the two cycles come from the Hamiltonian cycle of the original graph G ⇒ $G \in HAM-CYCLE$

Back Matter

Cook-Levin Theorem: The Goal

- The Cook-Levin Theorem is a very deep and technical theorem, so more than a little bit of struggling is to be expected!
- **Theorem (Cook-Levin Core):** SAT is NP-hard
  - By definition of NP-hard, this means $L \leq_p SAT$ for *every* language $L \in NP$
  - 203 Flashback: This means we should start with any arbitrary language $L \in NP$ and show that $L \leq_p SAT$
  - By definition of $\leq_p$, we need to show that there is a poly-time-computable function f such that
$$x \in L \Leftrightarrow f(x) \in SAT$$
  - We know that an instance of SAT is a Boolean formula, so $f(x) = \phi$ for some Boolean formula ϕ
  - But what about x? It depends on L, but L is arbitrary…, we can't do the typical mapping-reductions
- So far, we have established the *goal* of CL Theorem: $x \in L \Leftrightarrow f(x) = \phi \in SAT$
  - Stating and understand the goal that CL achieves is a major part of understanding it!

Cook-Levin Theorem: Modified Goal

- As mentioned earlier, we don't know anything about L except $L \in NP \Rightarrow$ there exists an efficient verifier V for L, this means
  - [1] $V(x, c)$ runs in polynomial time w.r.t. $n = |x|$, i.e., $O(n^k)$ for some constant k
  - [2] $x \in L$ iff there exists a certificate c that makes $V(x, c)$ accepts
- Recall that our goal is "$x \in L \Leftrightarrow f(x) = \phi \in SAT$". By [2], we can rewrite it as

  there exists a certificate c that makes $V(x, c)$ accepts $\Leftrightarrow f(x) = \phi \in SAT$

- The key idea now is to "translate"/ express the condition "there exist a certificate c…" as a formula ϕ, such that the condition holds iff ϕ is satisfiable

Cook-Levin Theorem: Strategy

- Our goal:

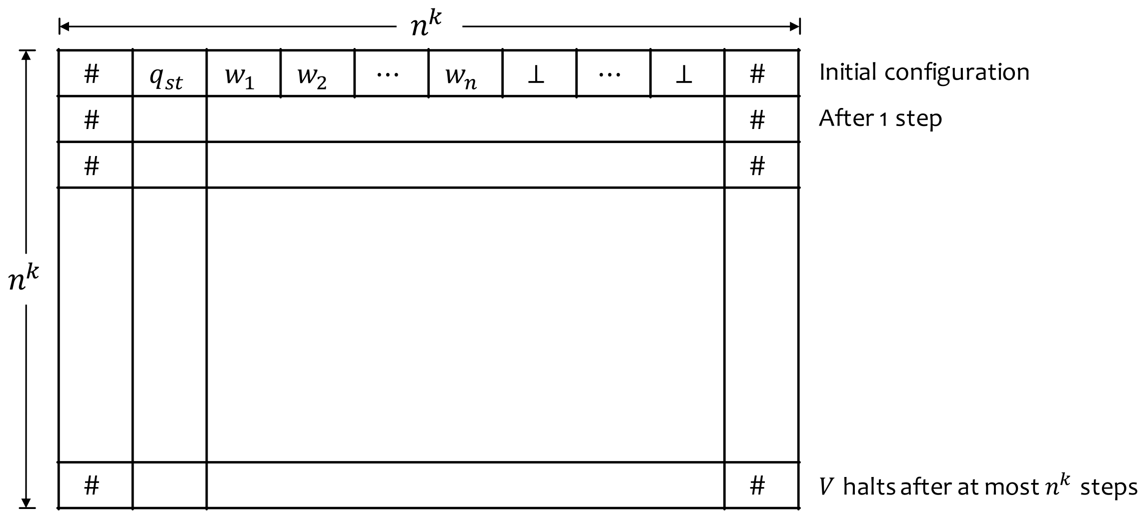
  there exists a certificate c that makes $V(x, c)$ accepts $\Leftrightarrow f(x) = \phi \in SAT$

- **Strategy:** Cleverly construct ϕ such that
  - *any such certificate* c (if exists) will correspond directly to Boolean values that satisfy ϕ, and
  - from *any* satisfying assignment to ϕ, we can directly "read off" a valid certificate c that makes $V(x, c)$ accept
- More specifically, the condition "there exists a certificate c that makes $V(x, c)$ accepts" can be expressed as several sub-conditions:
  - [1] When V is initialized, its tape consists of x, then a separator symbol, then some (unspecified) certificate c
  - [2] Each computational step of V follows from the previous one, according to V's transition function
  - [3] At some point within $O(|x|^k)$ steps, V enters its accept state

Side: The *Tableau*

- The tableau is a grid containing the entire step-by-step “history” of the computation
- Each row contains the tape after each step the TM (in our case, V) takes

At most n^k tape cells for an instance of size $n = |x|$



#	q_{st}	w_1	w_2	$\cdots$	w_n	$\bot$	$\cdots$	$\bot$	#	Initial configuration
#									#	After 1 step
#									#	
#									#	V halts after at most n^k steps

At most n^k computational steps

To be completely accurate, we need $n^k + 1$ rows and $n^k + 3$ columns to represent the tableau:

- $t \leq n^k$ computational steps + initial configuration requires $t + 1$ rows
- n^k tape cells + one for the active state + two for the # symbols requires $n^k + 3$ columns

- For convenience later, we assume that each configuration starts and ends with a # symbol
  - Therefore, the first and last columns of a tableau are all #s

Cook-Levin Theorem: Sub-formulas

- Given an instance x of L, we claim that it is possible to efficiently construct a sub-formula for each of the three conditions mentioned earlier:
  - [1] When V is initialized, its tape consists of x, then a separator symbol, then some (unspecified) certificate c
  - [2] Each computational step of V follows from the previous one, according to V's transition function
  - [3] At some point within $O\left(|x|^k\right)$ steps, V enters its accept state
- Each condition will correspond to one sub-formula, we then take ϕ to be the AND of these sub-formulas
  - This ensures that ϕ is satisfiable iff all three sub-formulas are satisfiable
- Basically, the formula ϕ has one variable for every (tableau cell, symbols*) pair, and the above sub-conditions are represented as Boolean formulas on these variables
  - *symbols contains more that just tape alphabet, will see on the next slide

Cook-Levin Theorem: Notation

- Before we begin to construct the formula, lets define some notations
  - The set of all valid symbols to be on the tableau
    $$S = \Gamma \cup Q \cup \{\#\}$$
    - Γ is the finite tape alphabet of V
    - Q is the finite set of states in V
    - $\# \notin \Gamma \cup Q$ is a special extra symbol
  - The variable corresponding to each (tableau cell, symbol) pair, as mentioned previously

$$t_{i,j,s} = \begin{cases} \text{true} & \text{if cell } (i,j) \text{ on the table contains symbol } s \\ \text{false} & \text{otherwise} \end{cases}$$

Cook-Levin Theorem: [0] Cell Consistency

- On top of the three conditions on the previous slides, we also have to check that each cell of the claimed tableau is well defined, i.e., it contains exactly one symbol
- To check that a given cell i, j has a well-defined symbol, we need exactly one variables $t_{i,j,s}$ to be true, over all $s \in S$The formula

$$\bigvee_{s \in S} t_{i,j,s}$$

  checks that *at least one* of the variables is true, and the formula

$$\neg \bigvee_{\text{distinct } s,s' \in S} \left(t_{i,j,s} \wedge t_{i,j,s'}\right) = \bigvee_{\text{distinct } s,s' \in S} \left(\neg t_{i,j,s} \wedge \neg t_{i,j,s'}\right)$$

  checks *no more than one* of the variables is true (equality holds by De Morgan's Law)

- Putting everything together:

$$\phi_{cell} = \bigwedge_{1 \le i,j \le n^k} \left[\bigvee_{s \in S} t_{i,j,s} \wedge \bigvee_{\text{distinct } s,s' \in S} \left(\neg t_{i,j,s} \wedge \neg t_{i,j,s'}\right) \right]$$

Cook-Levin Theorem: [3] Accepting Tableu

- To check that the claimed tableau is an accepting one, we just need to check that at least one cell has q_{acc} as its symbol, which is done by the following sub-formula

$$\phi_{acc} = \bigvee_{1 \le i,j, \le n^k} t_{i,j,q_{acc}}$$

Cook-Levin Theorem: [1] Starting Configuration

- For the first row of the tableau, which represents the starting configuration, we suppose that the encoding of an input pair (x, c) separates x from c on the tape using a special symbol $\$ \in \Gamma \setminus \Sigma$

#	q_0	x_1	x_2	…	x_n	$	c_1	…	c_m	$\perp$	…	#

- The first cell must be #, so we have $t_{1,1,\#}$
- The second cell must be the initial state, so we have $t_{1,2,q_0}$
- We require the row to have a specific, given instance x in its proper position of the starting configuration, so we have

$$\phi_{instance} = t_{1,3,x_1} \wedge t_{1,4,x_2} \wedge \cdots \wedge t_{1,n+1,x_n}$$

- After x, we should have the separator, so $t_{1,n+3,\$}$

Cook-Levin Theorem: [1] Starting Configuration

#	q_0	x_1	x_2	…	x_n	$	c_1	…	c_m	⊥	…	#

- For the tableau cells corresponding to the certificate, our construction **does not know what certificate(s)** c, if any, would cause $V(x, c)$ to accept, or even what their size are, so we leave a "placeholder" that allows any symbol from $\Sigma \cup \{\perp\}$

$$\phi_{cert} = \bigwedge_{j=n+4}^{n^k-1} t_{1,j,\perp} \vee \left(\bigvee_{s \in \Sigma} t_{1,j,s} \wedge \neg t_{1,j-1,\perp} \right)$$

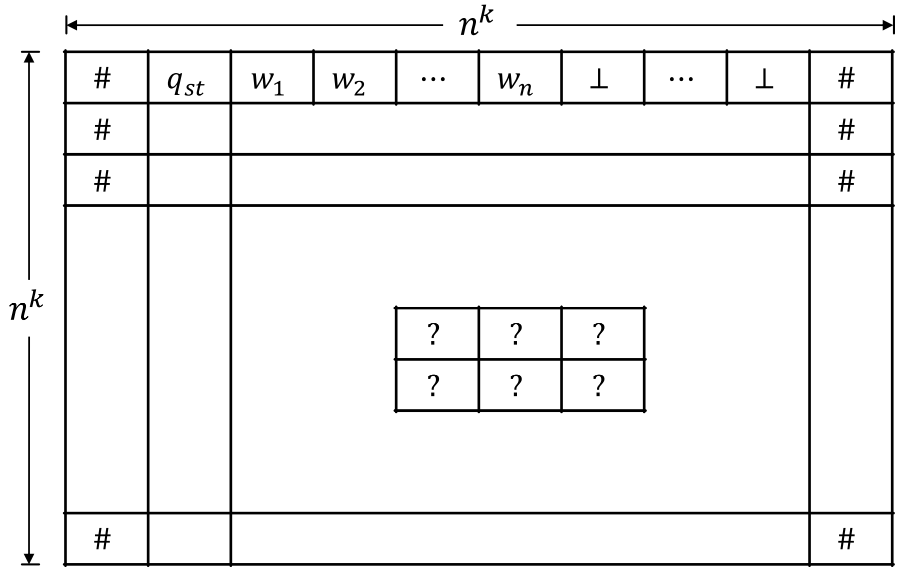
- Finally, the last cell must be #, so we have $t_{1,n^k,\#}$
- Putting everything together

$$\phi_{start} = t_{1,1,\#} \wedge t_{1,2,q_0} \wedge \phi_{instance} \wedge t_{1,n+3,\$} \wedge \phi_{cert} \wedge t_{1,n^k,\#}$$

Cook-Levin Theorem: [2] Transitions

- ϕ_{move} ensures that each configuration follows from previous configuration according to the transition function
- **Definition:** A 2×3 “window” is valid if it could appear in a valid tableau (Basically enforcing valid execution of V)
- **Theorem:** The whole tableau is valid iff every 2×3 window is valid

- You will explore this theorem in HW8!
- See course notes for details



n^k

n^k

| q_{st} | w_1 | w_2 | $\cdots$ | w_n | $\perp$ | $\cdots$ | $\perp$ |

|

|

? | ? | ?

? | ? | ?

|

Cook-Levin Theorem: Closing Remarks

- A very common question we get is “To what extend should we know Cook-Levin”
- The proof is very involved, and we do not expect you to know many of the details. However, you should have a good grasp on the main ideas, the components of the proof, and their consequences.
- Specifically, I would recommend the following from the course notes:
    - a solid grasp of Satisfiability and the Cook-Levin Theorem,
    - a reasonable grasp of Configurations and Tableaus,
    - some familiarity with the top-level summary of Constructing the Formula, and
    - reinforce your understanding of the above with Conclusion.