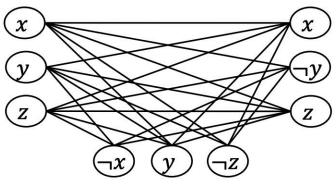
Goal: "*translate*" 3CNF formula φ into (G_φ, k_φ) such that:
- φ satisfiable ⇒ G_φ has k_φ-clique (**clique:** "yes")
- φ *not* satisfiable ⇒ G_φ *doesn't* have k_φ-clique (**clique:** "no")

* $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$ with 3 clauses
* **Idea for G_φ:** make each literal a vertex; add an edge between two literals in *different clauses* only if they're "compatible"

(*refer to different variables* **or** *they're the same*)



Observations:
1) Any clique can have at most one vertex from a clause ($k_\varphi \leq 3$)
2) A clique can have several x or $\neg x$, but not both

Will show: This graph has a clique of size 3 if and only if φ is satisfiable

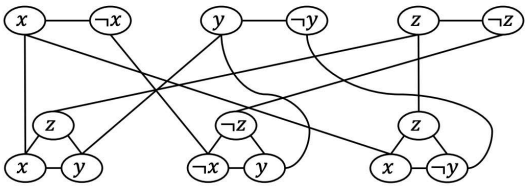
3SAT $\leq_p$ VERTEX-COVER

Goal: "*translate*" φ to (G_φ, k_φ) st:
- φ sat ⇒ G_φ has some k_φ-VC
- φ unsat ⇒ G_φ has no k_φ-VC

* $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$
* **Construction of G_φ: build** *variable gadgets* and *clause gadgets*; **add** edge $\{u, v\}$ if u is in a *variable gadget* and v is in a *clause gadget and* u and v are *labeled the same*
* **Set k_φ** to $n + 2m$ (n – number of variables, m – number of clauses)

variable gadgets

clause gadgets



Example

$\phi = (v_1 \vee v_2 \vee v_3) \wedge (\neg v_2 \vee \neg v_3 \vee \neg v_4) \wedge (\neg v_1 \vee v_2 \vee v_4)$

	v_1	v_2	v_3	v_4	C_1	C_2	C_3
x_1	1				1		
x_2		1			1		1
x_3			1		1		
x_4				1			1
y_1	1						1
y_2		1				1	
y_3			1			1	
y_4				1		1	
w_1					1		
w_2						1	
w_3							1
z_1					1		
z_2						1	
z_3							1
t	1	1	1	1	3	3	3

n+m digits (base 10)

x_3 represents occurrences of "v_3" in ϕ

y_3 represents occurrences of "$\neg v_3$" in ϕ

Two "dummy" numbers w_j, z_j for each clause C_j; help to reach the target t if ϕ is satisfiable

ALL BLANK DIGITS ARE 0

Definition: A *Hamiltonian* path from s to t in a graph G, is a path that starts at s, ends at t, and visits each vertex in G **exactly** once.

Definition: HAMCYCLE = {G : G is an undirected graph with a cycle that visits all the vertices}

Example: Decision version of ***Traveling Salesperson Problem (TSP)***
Given 4 cities and pair-wise distances between them, is there a tour of length at most 100 that visits all the cities?

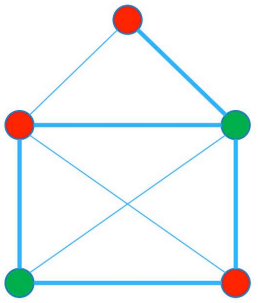
Remark: Here we only care about feasibility, not the actual tour.

Answer: $A \to B \to C \to D \to A$. Cost: 20+30+12+35 = 97.

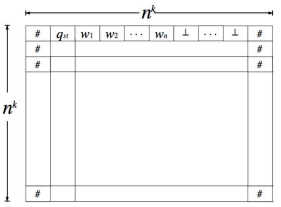
Reply: We can verify and are convinced.

Cuts

* A ***cut*** of a graph is a *partition* of its vertices $(S, \overline{S})$.
* An edge ***crosses*** the cut $(S, \overline{S})$ if one of its endpoints is in S and the other is in $\overline{S}$.
* The ***size*** of a cut $(S, \overline{S})$ is the number of edges crossing it.



A ***tableau*** is an array of symbols:
* Rows represent configurations (flanked by # symbols)
* Symbols can be from S = {0,1} ∪ Q ∪ {#, $, ⊥}
* Successive rows correspond to configurations



Tiancheng Jiao
EECS 376
tcjiao

* Given an input x, construct a Boolean formula $\phi_{V,x}$ that represents every valid tableau such that
  * V accepts x for some certificate c ⇒ $\phi_{V,x} \in$ SAT
  * V rejects x for all certificates c ⇒ $\phi_{V,x} \notin$ SAT

* $\phi_{V,x} = \phi_{V,x,start} \wedge \phi_{V,x,cell} \wedge \phi_{V,x,accept} \wedge \phi_{V,x,move}$
1. ϕ_{start} enforces the starting configuration at the top line
2. ϕ_{cell} ensures that every cell contains exactly one symbol
3. ϕ_{accept} ensures that V reaches an accepting configuration
4. ϕ_{move} ensures that each configuration follows from the previous configuration, according to the code of V

ϕ_{start} enforces the starting configuration

#	q_0	x	$	c	⊥	⊥	#

* Initial state q_0,
* Input x, |x| = n; certificate c, |c| = m
* $ - a special symbol that separates x and c
* We do not know c, so we leave a "placeholder"

$\phi_{start} = t_{1,1,\#} \wedge t_{1,2,q_0} \wedge t_{1,3,x_1} \wedge t_{1,4,x_2} \wedge \dots \wedge t_{1,n+2,x_n} \wedge t_{1,n+3,\$} \wedge$ (This fixes the first n+3 symbols)
$(t_{1,n+4,1} \vee t_{1,n+4,0} \vee t_{1,n+4,\perp}) \wedge (t_{1,n+5,1} \vee t_{1,n+5,0} \vee t_{1,n+5,\perp}) \wedge \dots$
(c_1 can be either 1 or 0 or ⊥)

* ϕ_{cell} ensures that every cell contains **exactly** one symbol

cannot contain two different symbols
contains at least one symbol
every cell

$$\bigwedge_{1 \leq i,j \leq n^k} \left[\bigvee_{\sigma \in S} t_{i,j,\sigma} \wedge \bigwedge_{\sigma \neq \tau \in S} \overline{(t_{i,j,\sigma} \wedge t_{i,j,\tau})} \right]$$

What do these parts mean, in English?

ϕ_{accept} ensures that V reaches an accepting configuration

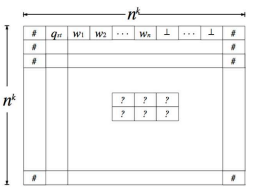
$$\bigvee_{1 \leq i,j \leq n^k} t_{i,j,q_{accept}}$$

ϕ_{move} ensures that each configuration follows from the previous configuration according to the δ function

Definition: A 2x3 "window" is valid if it could appear in a valid tableau

(Basically enforcing valid execution of V)

Theorem: The whole tableau is valid if and only if every 2x3 window is valid



- Given an input x, construct a Boolean formula $\phi_{V,x}$ that represents every valid tableau such that
  - V accepts x for some certificate c $\Rightarrow \phi_{V,x} \in$ SAT
  - V rejects x for all certificates c $\Rightarrow \phi_{V,x} \notin$ SAT

- $\phi_{V,x} = \phi_{V,x,start} \wedge \phi_{V,x,cell} \wedge \phi_{V,x,accept} \wedge \phi_{V,x,move}$
1. ϕ_{start} enforces the starting configuration at the top line
2. ϕ_{cell} ensures that every cell contains exactly one symbol
3. ϕ_{accept} ensures that V reaches an accepting configuration
4. ϕ_{move} ensures that each configuration follows from the previous configuration, according to the code of V

Cook-Levin Outline

Theorem [Cook-Levin]: If SAT ∈ **P**, then **NP** ⊆ **P**.

Proof outline:
- Let D_{SAT} be an efficient decider for SAT.
- Let L ∈ **NP**, so L has an efficient verifier V.
- **Goal:** L ∈ **P** via efficient decider D_L that uses D_{SAT} and V.
- $D_L(x)$:
  - Efficiently construct a poly-size Boolean formula $\phi_{V,x}$ so that:
    - $x \in L \iff \phi_{V,x}$ is satisfiable.
  - Output $D_{SAT}(\phi_{V,x})$.

What nice about $\phi_{V,x}$: Can capture the task of
(i) finding certificate c and
(ii) checking if V ends up in accept state
to finding a satisfying solution to $\phi_{V,x}$

- A ***Turing Machine*** is a 7-tuple $(Q, \Gamma, \Sigma, \delta, q_0, q_{accept}, q_{reject})$:
  - Q is a finite set of ***states***
  - $q_0 \in Q$ is the ***initial state***
  - $F = \{q_{accept}, q_{reject}\} \subseteq Q$ are the ***final*** (***accept/reject***) states
  - Σ is the ***input alphabet***
  - $\Gamma \supseteq \Sigma \cup \{\perp\}$ is the ***tape alphabet*** ($\perp \notin \Sigma$ is the ***blank symbol***)
  - $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the ***transition function***
- **Takeaway:** TMs are a well-defined type of "computer".

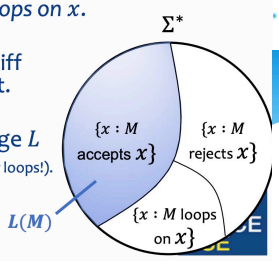
- **Definition:** A TM ***decides*** language if:
  1. *accepts* every string ("accepts"), and
  2. *rejects* every string ("rejects").

We say that is a ***decider*** (for), and is ***decidable***.
- **Note:** By definition, *does not loop* on any input!

- **Definition:** The language of a TM M is $L(M) = \{x \mid M \text{ accepts } x\}$.
- **Question:** What if $x \notin L(M)$? (does not accept.)
- **Answer:** Then M either *rejects* x, or *loops on* x.

- **Conclusion:** TM M decides language L iff $L = L(M)$ and M halts on *every* input.

- Definition: TM M recognizes language L if $L = L(M)$ (regardless of whether ever loops!).
- More on this later...



$$L^*[j] = 1 - T[j,j]$$

	s_1	s_2	s_3	s_4	s_5	s_6	...
$L(M_1)$	1	0	0	1	1	0	...
$L(M_2)$	0	1	1	0	0	0	...
$L(M_3)$	1	1	1	1	1	1	...
$L(M_4)$	0	0	0	0	0	0	...
$L(M_5)$	1	0	1	0	0	0	...
...	...	...	...	...	...	...	...

- **Proof.** If L for some i $L^* \neq L(M_i$

L^*	0	0	

$$L_{BARBER} = \{\langle M \rangle : M \text{ does not accept } \langle M \rangle\}$$

- **Result:** "*Program* B *accepts the source code of those, and only those, who do not accept their own source code.*"

- **Question:** Does B accept its own source code?
- **Answer:** Suppose P is a program.
1. P accepts its own code $\Rightarrow B$ does not accept P's code.
2. P does not accept its own code $\Rightarrow B$ accepts P's code.

- **Question:** What if $P = B$?
1. B accepts its own code $\Rightarrow B$ does not accept B's code.
2. B does not accept its own code $\Rightarrow B$ accepts B's code.

Paradox! (Program B cannot exist)

L_{HALT} is Undecidable

We need to implement:
C is given two input: $\langle M \rangle$ and x
M accepts $x \Rightarrow C$ accepts$(\langle M \rangle, x)$
M does not accept $\langle M \rangle \Rightarrow C$ rejects$(\langle M \rangle, x)$

We have:
H is given two inputs: $\langle M \rangle$ and x
M accepts or rejects $x \Rightarrow H$ accepts $(\langle M \rangle, x)$
M loops on $x \Rightarrow H$ rejects $(\langle M \rangle, x)$

- **In code:**
- C on input$(\langle M \rangle, x)$:
  - Run H on $(\langle M \rangle, x)$ (*Ask H: "does M halt on x"?*)
  - If H rejects, *reject* (*M loops on x, so it can't accept it*)
  - If H accepts, run M on x (*M halts on x, so this is safe to do*)
    - If M accepts, *accept*; If M rejects, *reject* (*answer like M*)

- **Analysis:** C always halts (why?). Moreover:
  - M accepts $x \Rightarrow H$ accepts $(\langle M \rangle, x) \Rightarrow C$ accepts $(\langle M \rangle, x)$
  - M rejects $x \Rightarrow H$ accepts $(\langle M \rangle, x) \Rightarrow C$ rejects $(\langle M \rangle, x)$
  - M loops on $x \Rightarrow H$ rejects $(\langle M \rangle, x) \Rightarrow C$ rejects $(\langle M \rangle, x)$

- **Conclusion:** L_{HALT} decidable $\Rightarrow L_{ACC}$ decidable
- **Contrapositive:** L_{ACC} undecidable $\Rightarrow L_{HALT}$ undecidable

- **Definition:** Language A is ***Turing reducible*** to language B, written $A \leq_T B$, if there exists a program M that decides A using a "*black box*" that decides B.

- **Previous results:** $L_{BARBER} \leq_T L_{ACC}$ and $L_{ACC} \leq_T L_{HALT}$
- **Intuition:** B is "*no easier*" than A to decide.
- **Theorem:** Suppose $A \leq_T B$. Then B is decidable $\Rightarrow A$ is decidable.
- **Definition:** A language A is ***recognizable*** if there exists a program M (a "recognizer") that recognizes it: L(M)=A.
- **Theorem:** If a language A and its complement $\overline{A}$ are both *recognizable*, then A is *decidable*.

$L_{\emptyset}$ is Unrecognizable

- **Claim:** $L_{\emptyset} = \{\langle M \rangle : L(M) = \emptyset\}$ is unrecognizable.
- **Proof:** We show that $L_{\emptyset}$ is undecidable ($L_{ACC} \leq_T L_{\emptyset}$) and $\overline{L_{\emptyset}}$ is recognizable.
- **Step 1:** Let N be a decider for $L_{\emptyset}$. Construct a decider C for L_{ACC}:
- $C(\langle M, x \rangle)$:
  1. Construct a program "$M'(w)$: run M on x and answer as M does"
  2. Call N on $\langle M' \rangle$ and return the *opposite* output
- **Analysis:** C halts since N does. Moreover:
  - M accepts $x \iff L(M') \neq \emptyset \iff N$ rejects $\langle M' \rangle \iff C$ accepts $(\langle M \rangle, x)$

"Dovetailing"

- **Claim:** $L_{\emptyset} = \{\langle M \rangle : L(M) = \emptyset\}$ is unrecognizable.
- **Proof:** We show that $L_{\emptyset}$ is undecidable ($L_{ACC} \leq_T L_{\emptyset}$) and $\overline{L_{\emptyset}}$ is recognizable.
- **Step 2:** We need to construct a recognizer for $\overline{L_{\emptyset}} = \{\langle M \rangle : L(M) \neq \emptyset\}$.
- **Idea:** Do step i of $M(s_j)$ in "block" $i + j$ (like in proof that $\mathbb{Q}^+$ is countable).
- $R(\langle M \rangle)$:
  1. For $t = 1, 2, 3, \ldots$:
     2. For $j = 1, 2, \ldots, t$:
        3. Run one (additional) step of $M(s_j)$
        4. If $M(s_j)$ accepts, accept.
- **Analysis:**
  - $\langle M \rangle \in \overline{L_{\emptyset}} \Rightarrow \exists j, k.\ M$ accepts s_j in k steps $\Rightarrow R$ accepts $\langle M \rangle$.
  - $\langle M \rangle \notin \overline{L_{\emptyset}} \Rightarrow L(M) = \emptyset \Rightarrow R$ loops on $\langle M \rangle$.

Berry's Paradox

- **Aside (Berry's Paradox):** "The first positive integer that cannot be defined in <70 characters."
- Let $S \subset \mathbb{Z}^+$ be the set of positive integers that cannot be defined in <70 characters. Let $x = \min(S)$. Then:
  - x cannot be defined in <70 characters, since $x \in S$
  - x can be defined by the sentence "The first positive integer that cannot be defined in <70 characters.", which has 68 characters

Paradox!

$K_U(w)$ is Uncomputable

- **Proof:** Pick the language U. Suppose M is a program that computes $K_U(w)$.
- **Definition:** For every n define new program Q_n:
  - const int LENGTH = n; (*n is "hardcoded"*)
  - Iterate over all $x \in \{0,1\}^{\text{LENGTH}}$: (*n encoded in binary*)
    - Compute $K_U(x)$ (*using M as a black-box*)
    - Output the first x such that $K_U(x) \geq$ LENGTH
- **Observation:** For every n the size of the program Q_n is $O(\log n)$.
- **Analysis:** Let w_n be the output of Q_n. What is $K_U(w_n)$?
- **By definition:** $K_U(w_n) \geq n$, since Q_n outputs an x such that $K_U(x) \geq n$.
On the other hand, Q_n outputs w_n, so Q_n is a program that outputs w_n; by definition of Kolmogorov complexity, $K_U(w_n) \leq |Q_n|$.
- **Therefore:** $K_U(w_n) \geq n$ and $K_U(w_n) \leq O(\log n)$.
- **Contradiction:** Conditions cannot be fulfilled simultaneously!
- **Conclusion:** No such M exists.

Note: this could hide a large constant depending on the size of the program M

- **Definition:** The ***Kolmogorov Complexity*** of w:
$K_U(w) = \min\{|\langle M \rangle| : M \text{ outputs } w, \text{on empty input } \varepsilon\}$.
That is, the size of the shortest U-program that outputs w.