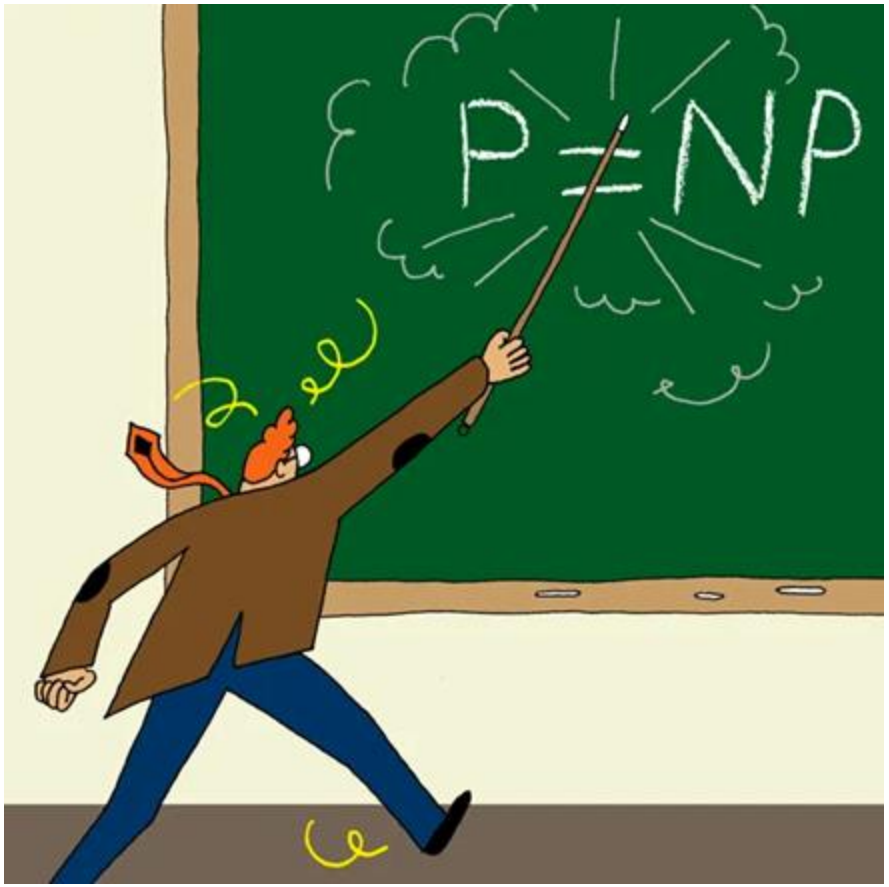
The Cook-Levin Theorem: SAT is NP-complete



The New Yorker, 2013

Millennium Problems

Yang–Mills and Mass Gap

Experiment and computer simulations suggest the existence of a "mass gap" in the solution to the quantum versions of the Yang-Mills equations. But no proof of this property is known.

Riemann Hypothesis

The prime number theorem determines the average distribution of the primes. The Riemann hypothesis tells us about the deviation from the average. Formulated in Riemann's 1859 paper, it asserts that all the 'non-obvious' zeros of the zeta function are complex numbers with real part 1/2.

P vs NP Problem

If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

Navier–Stokes Equation

This is the equation which governs the flow of fluids such as water and air. However, there is no proof for the most basic questions one can ask: do solutions exist, and are they unique? Why ask for a proof? Because a proof gives not only certitude, but also understanding.

Hodge Conjecture

The answer to this conjecture determines how much of the topology of the solution set of a system of algebraic equations can be defined in terms of further algebraic equations. The Hodge conjecture is known in certain special cases, e.g., when the solution set has dimension less than four. But in dimension four it is unknown.

Poincaré Conjecture

In 1904 the French mathematician Henri Poincaré asked if the three dimensional sphere is characterized as the unique simply connected three manifold. This question, the Poincaré conjecture, was a special case of Thurston's geometrization conjecture. Perelman's proof tells us that every three manifold is built from a set of standard pieces, each with one of eight well-understood geometries.

Birch and Swinnerton-Dyer Conjecture

Supported by much experimental evidence, this conjecture relates the number of points on an elliptic curve mod p to the rank of the group of rational points. Elliptic curves, defined by cubic equations in two variables, are fundamental mathematical objects that arise in many areas: Wiles' proof of the Fermat Conjecture, factorization of numbers into primes, and cryptography, to name three.

Simpsons Halloween Special, 2013

BBC | Sign in | News | Sport | Weather | Shop | Mor

NEWS

Home | Video | World | US & Canada | UK | Business | Tech | Science

Magazine

Homer Simpson's scary maths problems

By Simon Singh
Science writer

31 October 2013 | Magazine | Share

For example, in one scene, the letters P and NP can be seen over Homer's right shoulder. Although these three letters would have made no sense to most viewers, they are a deliberate nod towards a statement about one of the most important unsolved problems in theoretical computer science. Indeed, this is such a weighty puzzle that there is a reward of $1m (£623,000) for whoever solves the mystery.

It is surprising to find a reference to P- and NP-type problems in a television sitcom, but not when the writer is Cohen, because he studied them while doing his master's degree at the University of California in Berkeley.

Jack Edmonds
(defined complexity class P)

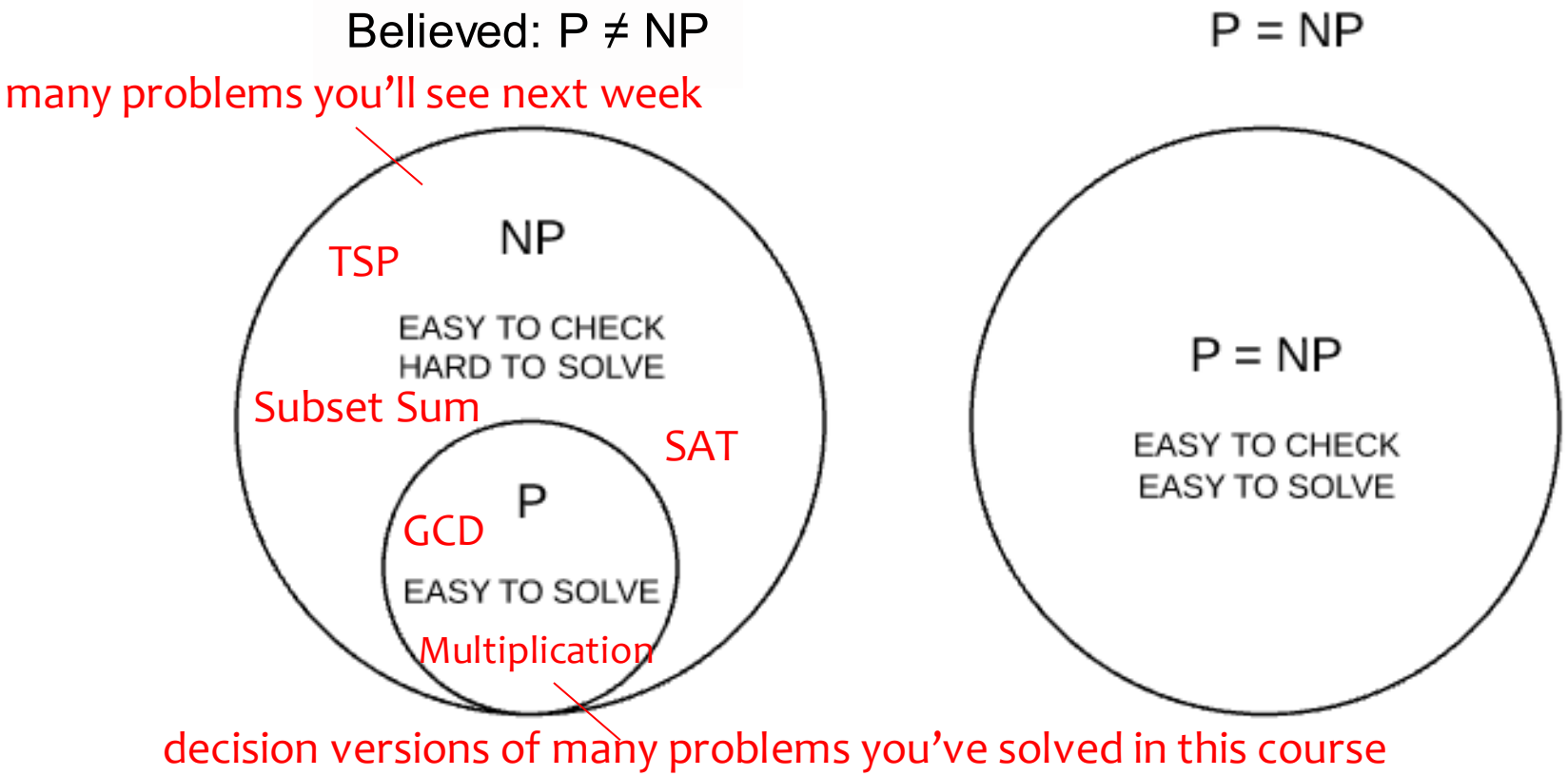
The Complexity Classes **P** and **NP**

Definition: **P** is the set of all decision problems that can be decided in polynomial time.

Definition: **NP** is the set of all decision problems whose solution be verified in polynomial time.

Biggest open problem in computer science: $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$

Two Possible Worlds



Why not define NP like this…

"The set of decision problems that can be solved in exponential time."

EXP $P \neq EXP$

NP-Hardness and **NP**-Completeness

Informal definition: A problem **L** is **NP**-hard if it is at least as hard as EVERY problem in **NP**.

Formal definition: A problem **L** is **NP**-hard if: for EVERY problem **X** in NP, $\mathbf{X} \leq_p \mathbf{L}$.

a new type of reduction!

A problem **L** is **NP**-complete if

- **L** ∈ **NP**
- **L** is **NP**-hard

the class of "hardest problems in NP"

A polynomial-time algorithm for any **NP**-hard problem would imply **P** = **NP**.

Polynomial-time mapping reduction from **A** to **B** (denoted **A** ≤p **B**)

Defn: a poly-time-computable function f such that: x in **A** ⇔ f(x) in **B**.

*"Given any instance of **A**, in polynomial time we can construct an instance of **B** whose yes/no answer is the same."*

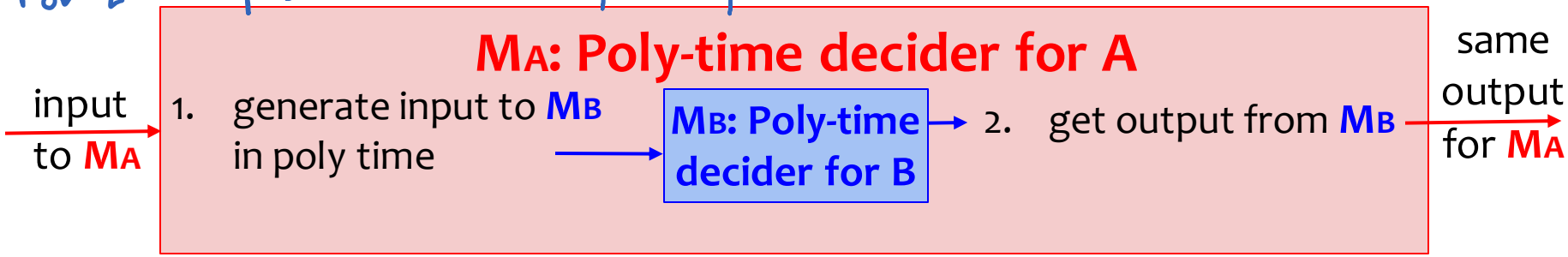
What it implies:

1. We can use a black-box poly-time decider for **B** as a subroutine to decide **A** in poly time.
1. If **B** ∈ **P** then **A** ∈ **P**.
2. If **A** is NP-hard then **B** is NP-hard.

In mapping reductions, there's no "flipping" the answer, and only one call to B

M_A: construct input y to M_B run $M_B(y)$ and return same

For EVERY problem $X \in NP$ $X \leq_p A \leq_p B$



"Problem **B** is at least as hard as Problem **A**"

Polynomial-time mapping reduction from **A** to **B** (denoted **A** ≤p **B**)

Defn: a poly-time-computable function f such that: x in **A** ⇔ f(x) in **B**.

*"Given any instance of **A**, in polynomial time we can construct an instance of **B** whose yes/no answer is the same."*

What it implies:

1. We can use a black-box poly-time decider for **B** as a subroutine to decide **A** in poly time.
1. If **B** ∈ **P** then **A** ∈ **P**.
2. If **A** is NP-hard then **B** is NP-hard.

In mapping reductions, there's no "flipping" the answer, and only one call to B

Instance of **A** →  → Instance of **B** with same answer

fast conversion machine

"Problem **B** is at least as hard as Problem **A**"

Polynomial-time mapping reduction from **A** to **B** (denoted **A** ≤p **B**)

Defn: a poly-time-computable function f such that: x in **A** ⇔ f(x) in **B**.

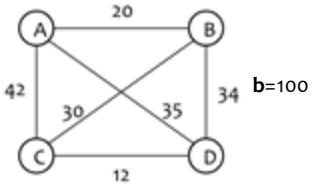
*"Given any instance of **A**, in polynomial time we can construct an instance of **B** whose yes/no answer is the same."*

In mapping reductions, there's no "flipping" the answer, and only one call to B

What it implies:

1. We can use a black-box poly-time decider for **B** as a subroutine to decide **A** in poly time.
1. If **B** ∈ **P** then **A** ∈ **P**. ⇒ Contrapositive: $A \notin P \Rightarrow B \notin P$
2. If **A** is NP-hard then **B** is NP-hard.

Instance of **A**



E.g. TSP Instance

fast conversion machine

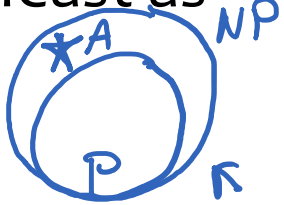
Instance of **B** with same answer

$(x_1 \vee \overline{x}_2 \vee x_4) \wedge$
$(x_2 \vee x_3 \vee \overline{x}_{42})$

SAT Instance

NP-Hardness and **NP**-Completeness

Informal definition: A problem **L** is **NP**-hard if it is at least as hard as EVERY problem in **NP**.



Formal definition: A problem **L** is **NP**-hard if: for EVERY problem **X** in NP, $\mathbf{X} \leq_p \mathbf{L}$.

a new type of reduction!

A problem **L** is **NP**-complete if

- **L ∈ NP**
- **L** is **NP**-hard

complete

Sps A is NP-~~hard~~.

$A \notin P \Rightarrow P \neq NP$

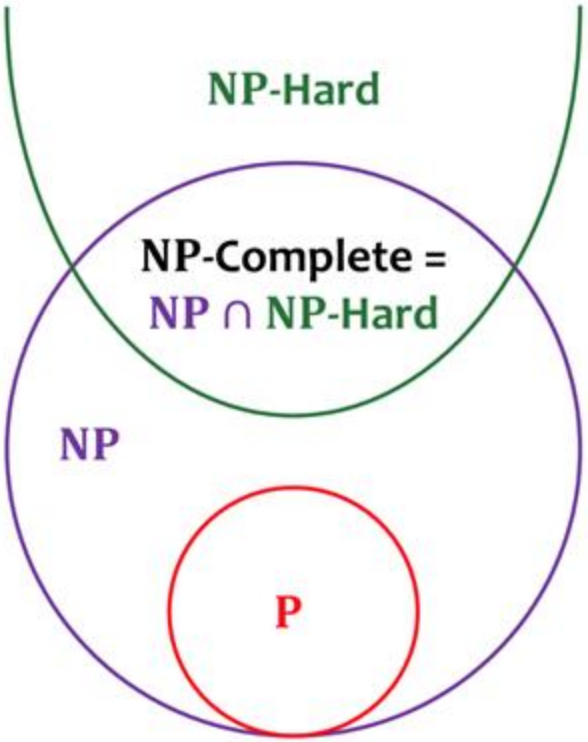
$A \in P \Rightarrow P = NP$

A polynomial-time algorithm for an **NP**-hard problem would imply **P** = **NP**.

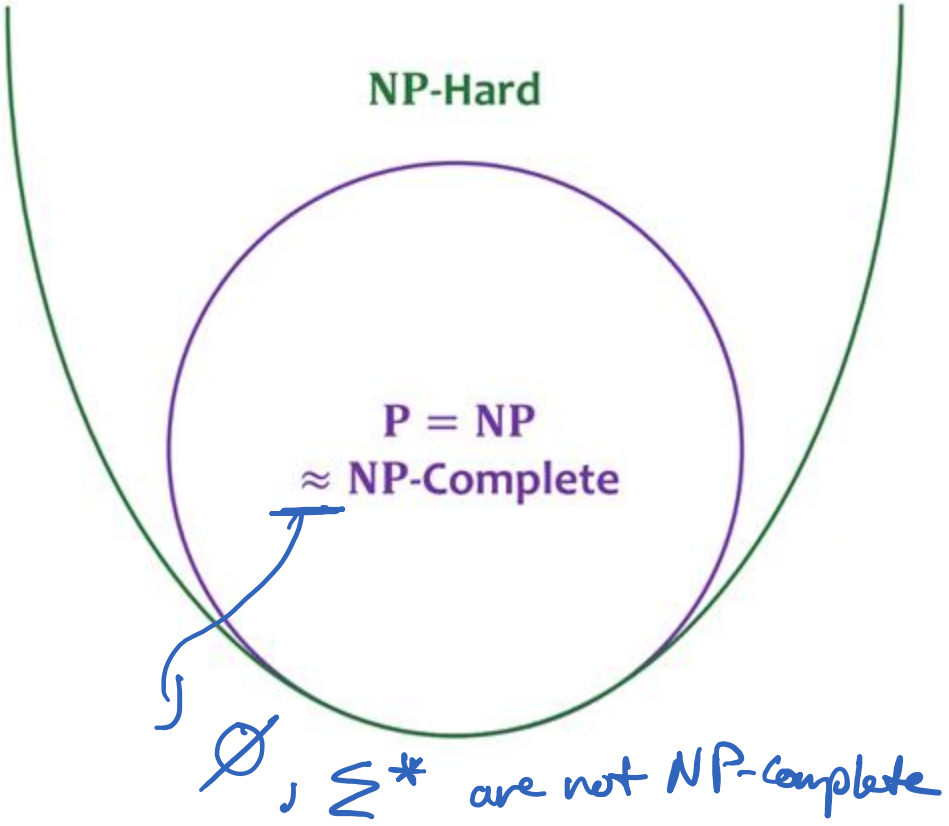
Every problem $X \in NP$: $X \leq_p A \Rightarrow$ If $A \in P \Rightarrow X \in P$

Two Possible Worlds

P ≠ NP



P = NP



The Cook-Levin Theorem (1971)

SAT is **NP**-complete.

That is, to resolve P $\stackrel{?}{=}$ NP we "just" need to determine the status of SAT.

SAT was the original **NP**-complete problem

Satisfiability (SAT)

Example SAT instance:

$$\Phi = (x \vee y) \wedge (\neg y \vee x \vee \neg z) \wedge (\neg x \vee (y \wedge \neg z))$$

A Boolean formula is made up of:

- "literals": variables and their negations (e.g. x, y, z, ¬x, ¬y, ¬z)
- OR: ∨
- AND: ∧

$x=T, y=T, z=F \leftarrow$

$X \wedge \neg X$

Input: A Boolean formula Φ

Output: Is the formula Φ *satisfiable*? That is, does there exist a true/false assignment to the variables that makes the entire formula true?

What's an assignment that satisfies the above formula?
What's an example of an unsatisfiable formula?

Proving the Cook-Levin Theorem: SAT is NP-complete

Last time we showed that SAT is in NP.

So, to show that SAT is NP-complete, we need to show that SAT is NP-hard i.e. for EVERY problem **L** in NP, **L** $\leq_p$ **SAT**.

?

$$(x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_2 \vee x_3 \vee \overline{x}_{42})$$

Instance of ANY problem **L** in NP

fast conversion machine

SAT Instance with same yes/no answer as the instance of L

Proving SAT is NP-hard

We are given: an arbitrary instance **x** of any problem **L** in **NP**.
Let **Verify-L** be a polynomial-time verification algorithm for **L**.

GOAL: In polynomial time, construct an instance of SAT whose answer is "yes" **if and only if** **x** ~~is a "yes" instance of **L**~~
(i.e. iff there exists **C** such that **Verify-L**(**x**, **C**) accepts)

Verify-L input: instance **y** of the problem **L** and a poly-length **"certificate" C**

Verify-L(**y**, **C**) **output:**

1. If **y** is a "yes" instance: There exists **C** such that **Verify-L**(**y**, **C**) accepts
2. If **y** is a "no" instance: **Verify-L**(**y**, **C**) rejects for every **C**

Proving SAT is NP-hard

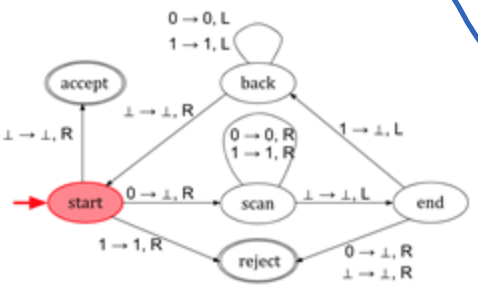
We are given: an arbitrary instance **x** of any problem **L** in **NP**.
Let **Verify-L** be a polynomial-time verification algorithm for **L**.

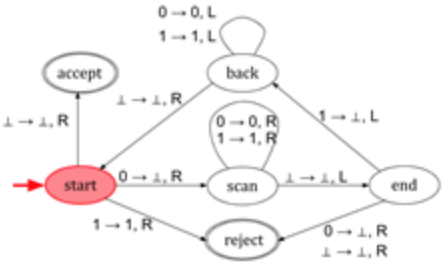
GOAL: In polynomial time, construct an instance of SAT whose answer is "yes" **if and only if** there exists **C** such that **Verify-L**(**x**, **C**) accepts.

e.g. Verify-TSP

Consider **Verify-L** as a TM:

Verify-L takes n^k steps
(for some constant k)

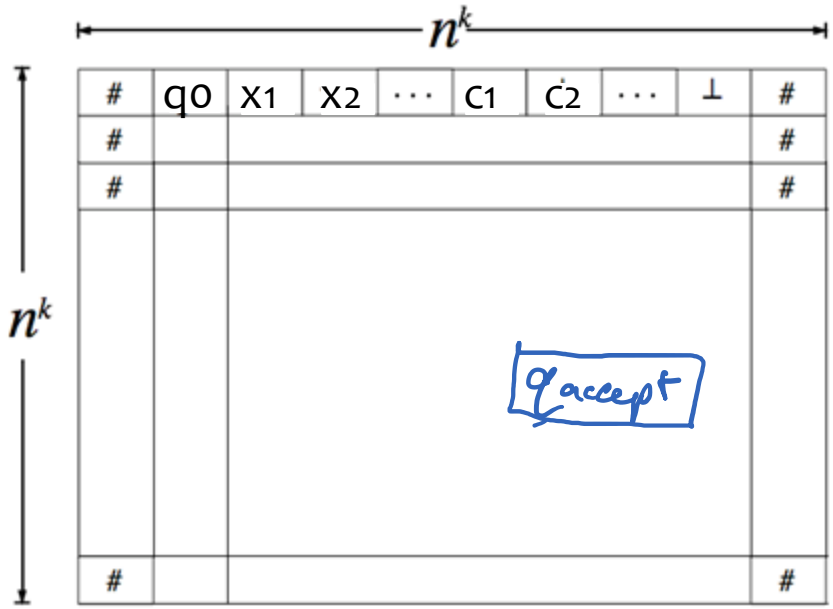




Execution of Verify-L(x,C) as a "Tableau"

If a row of the tableau says "#011$q_5$0001#" it means:

- The tape says: "0110001"
- We are in state q_5
- The head is pointing to the symbol right after the state, i.e. 011**0**001



Initial configuration of TM

After 1 step

⋮

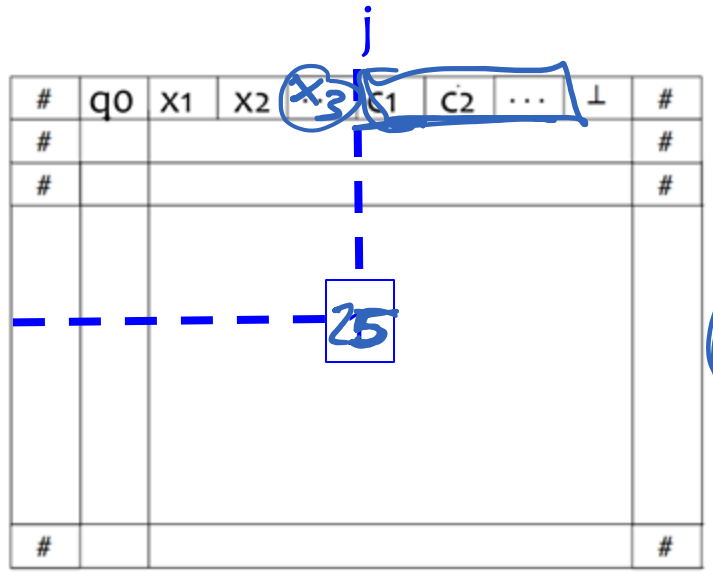
Verify-L halts after at most n^k steps

If we know the top row, we can fill in the entire table

Constructing the SAT instance: Overview

Given: Instance **x** of any problem **L** in **NP** and poly-time algorithm **Verify-L**.

GOAL: In poly time, construct an instance φ of SAT whose answer is "yes" **if and only if** exists **C** such that tableau of **Verify-L**(**x**, **C**) contains q_{accept}.



$t_{i,j,1} = T$

$t_{i,j,2} = F$

φ will be of the form:

$$t_{1,3,x_1} \wedge t_{1,4,x_2} \wedge \cdots \wedge t_{1,n+2,x_n} \wedge t_{1,n+3,\$} \wedge$$

$$\wedge \left(t_{1,n+5,0} \vee t_{1,n+5,1} \vee t_{1,n+5,\perp}\right) \wedge \cdots$$

Variables of φ: variable $\mathbf{t_{i,j,s}}$ will mean:

the symbol in cell (i,j) of the tableau is "s"

Constructing the SAT instance: Overview

Given: Instance **x** of any problem **L** in **NP** and poly-time algorithm **Verify-L**.

GOAL: In poly time, construct an instance φ of SAT whose answer is "yes" **if and only if** exists **C** such that tableau of **Verify-L**(**x**, **C**) contains q_{accept}.

Preview of "⇒" direction of proof:

- If φ has answer "yes" then the values of the variables in the satisfying assignment tell you what symbol to put in each cell of the tableau!
- **By careful construction of φ, this will be a valid tableau of Verify-L(x, C) containing q_{accept}, for some C.**
- Look at the first row cells to find that **C**!

Variables of φ: variable $\mathbf{t_{i,j,s}}$ will mean:

the symbol in cell (i,j) of the tableau is "s"

Constructing the SAT instance: Overview

Given: Instance **x** of any problem **L** in **NP** and poly-time algorithm **Verify-L**.

GOAL: In poly time, construct an instance φ of SAT whose answer is "yes" **if and only if** exists **C** such that tableau of **Verify-L**(**x**, **C**) contains q_{accept}.

$$\varphi = \varphi_{start} \wedge \varphi_{cell} \wedge \varphi_{accept} \wedge \varphi_{step}$$

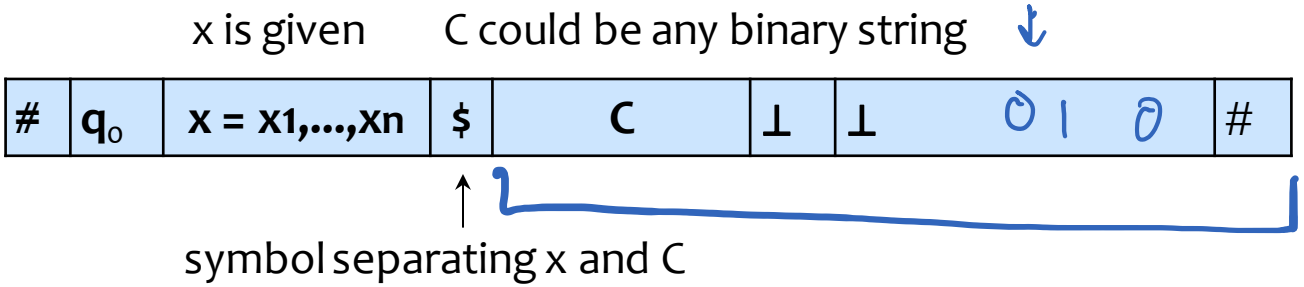
1. φ_{start} specifies the **first row** of the tableau
2. φ_{accept} ensures that the tableau contains the **accepting state q_{accept}**
3. φ_{cell} ensures that there is exactly **one symbol per cell**
4. φ_{step} ensures that each row follows from the previous row according to the **TM transition rules**

$n^k \cdot n^k \cdot$ constant

Variables of φ: variable $\mathbf{t_{i,j,s}}$ will mean:
the symbol in cell (i,j) of the tableau is "s"

How many variables are there?

1. φ_{start} specifies the **first row** of the tableau

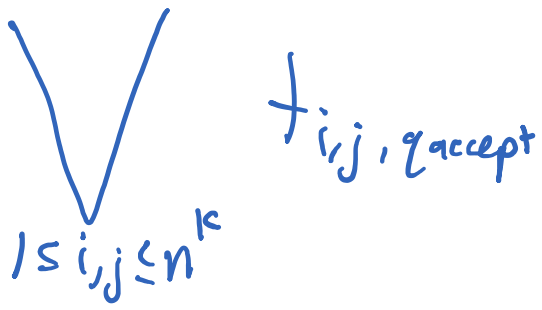


$$\varphi_{start} = t_{1,1,\#} \wedge t_{1,2,q_0} \wedge t_{1,3,x_1} \wedge t_{1,4,x_2} \wedge \cdots \wedge t_{1,n+2,x_n} \wedge t_{1,n+3,\$} \wedge$$

$$\left(t_{1,n+4,0} \vee t_{1,n+4,1} \vee t_{1,n+4,\perp}\right) \wedge \left(t_{1,n+5,0} \vee t_{1,n+5,1} \vee t_{1,n+5,\perp}\right) \wedge \cdots$$

*this is not quite complete since the ⊥s come after binary string…footnote in course notes has the fix

2. φ_{accept} ensures that the tableau contains $\mathbf{q_{accept}}$

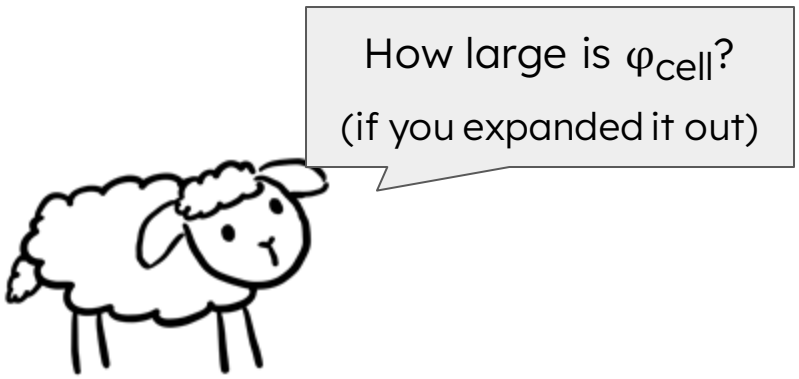


3. φ_{cell} ensures that there is exactly **one symbol per cell**

$$\bigwedge_{1 \le i,j \le n^k} \left[\bigvee_{s \in \Gamma} t_{i,j,s} \right] \quad \bigwedge_{s \neq u \in \Gamma} \left(\neg t_{i,j,s} \vee \neg t_{i,j,u} \right)$$

"Every cell" "has ≥ 1 symbol in it" AND "has ≤ 1 symbol in it"

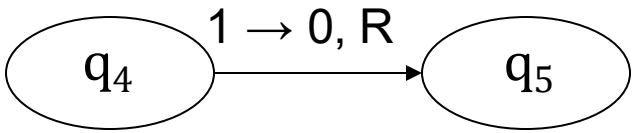
i.e. "for every pair of symbols in Γ, at least one of them is *not* in the cell"



Notation: Γ is the tape alphabet

4. φ_{step} ensures that each row follows from the previous row according to the **TM transition rules**

Example transition rule:



Consider every pair P of strings that could appear in adjacent rows!

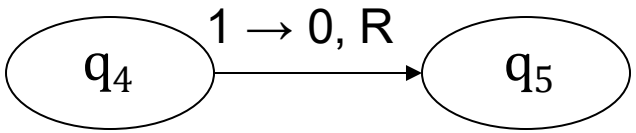
For example:
#01001$q_4$101...#
#010010$q_5$01...#

Take the AND over all corresponding variables to form φ_P, then take the OR over all φ_P.

$\sim 2^{n^k}$

4. φ_{step} ensures that each row follows from the previous row according to the **TM transition rules**

Example transition rule:



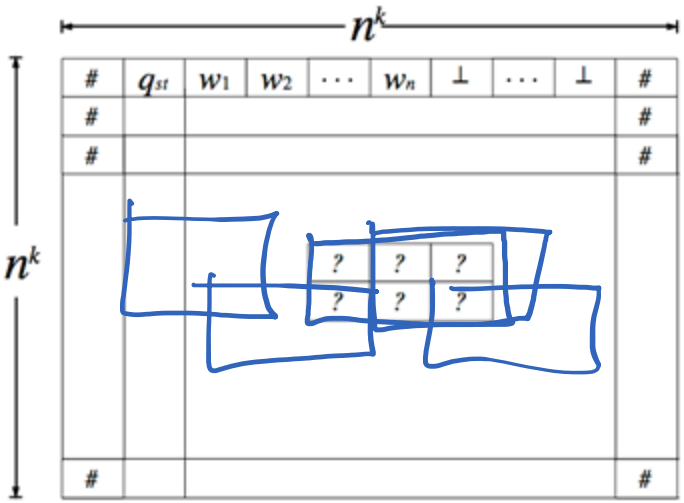
0	q_4	1
0	0	q_5

~~| | | |~~
~~| 0 | q_4 | 1 |~~
~~| 0 | 1 | q_5 |~~

Fix: Only consider 2x3 "windows"!

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

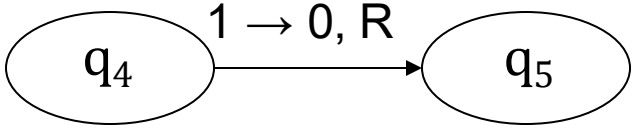
Theorem (we won't prove)**:** The whole tableau is valid if and only if every 2x3 window is valid.



On the HW you will think more about valid 2x3 windows

4. φ_{step} ensures that each row follows from the previous row according to the **TM transition rules**

Example transition rule:



0	q_4	1
0	0	q_5

Fix: Only consider 2x3 "windows"!

another valid 2x3 window

$$\varphi_{step} = \bigwedge_{i,j} \left((t_{i,j,0} \wedge t_{i,j+1,q4} \wedge t_{i,j+2,1} \wedge t_{i+1,j,0} \wedge t_{i+1,j+1,0} \wedge t_{i+1,j+2,q5}) \vee (\ldots) \vee \ldots \right)$$

How large is φ_{step}?

More valid windows:

0	1	1
0	1	1

nothing changes if head isn't around

0	1	1
q_3	1	1

head could enter from the side

Revisiting our Goal

Given: Instance **x** of any problem **L** in **NP** and poly-time algorithm **Verify-L**.

GOAL: In poly time, construct an instance φ of SAT whose answer is "yes" **if and only if** exists **C** such that tableau of **Verify-L**(**x**, **C**) contains q_{accept}.

$$\varphi = \varphi_{start} \wedge \varphi_{cell} \wedge \varphi_{accept} \wedge \varphi_{step}$$

1. φ_{start} specifies the **first row** of the tableau
2. φ_{accept} ensures that the tableau contains the **accepting state q_{accept}**
3. φ_{cell} ensures that there is exactly **one symbol per cell**
4. φ_{step} ensures that each row follows from the previous row according to the **TM transition rules**

Variables of φ: variable $\mathbf{t_{i,j,s}}$ will mean:
the symbol in cell (i,j) of the tableau is "s"

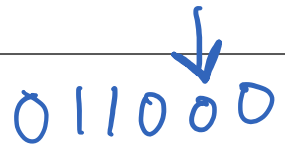
Revisiting our Goal

Given: Instance **x** of any problem **L** in **NP** and poly-time algorithm **Verify-L**.

GOAL: In poly time, construct an instance φ of SAT whose answer is "yes" **if and only if** exists **C** such that tableau of **Verify-L**(**x**, **C**) contains q_{accept}.

Structure of "⇒" direction of proof:

- If φ has answer "yes" then the values of the variables in the satisfying assignment tell you what symbol to put in each cell of the tableau!
- **By careful construction of φ, this will be a <u>valid tableau</u> of Verify-L(x, C) containing q_{accept}, for some C.**
- Look at the first row cells to find that **C**!



*Full proof would also need to include "⇐" direction