Reductions

- designing algorithms
- proving limits
- classifying problems
- NP-completeness

Bird's-eye view

Desiderata.
Classify problems according to their computational requirements.

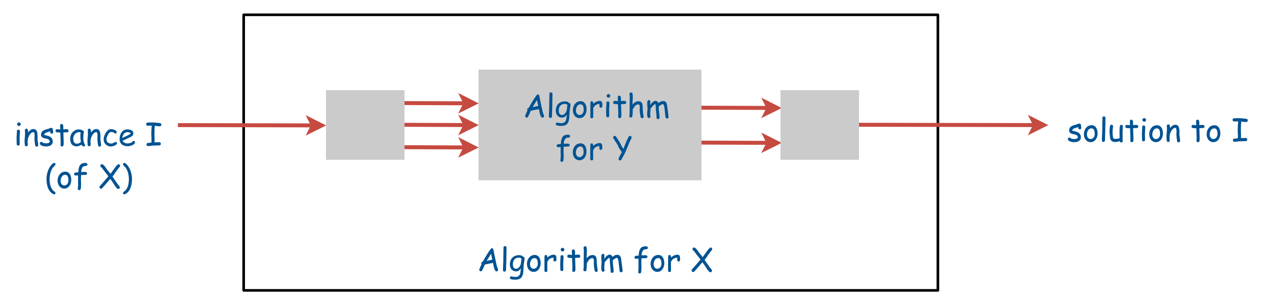
Frustrating news.
Huge number of fundamental problems have defied classification

Desiderata'.
Suppose we could (couldn't) solve problem X efficiently.
What else could (couldn't) we solve efficiently?

Give me a lever long enough and a fulcrum on which to place it, and I shall move the world. -Archimedes

Reduction

Def. Problem X reduces to problem Y
if you can use an algorithm that solves Y to help solve X



Ex. Euclidean MST reduces to Voronoi.

To solve Euclidean MST on N points

- solve Voronoi for those points
- construct graph with linear number of edges
- use Prim/Kruskal to find MST in time proportional to N log N

Reduction

Def. Problem X reduces to problem Y
if you can use an algorithm that solves Y to help solve X

Cost of solving X = M*(cost of solving Y) + cost of reduction.

number of times Y is used

Applications

- designing algorithms: given algorithm for Y, can also solve X.
- proving limits: if X is hard, then so is Y.
- classifying problems: establish relative difficulty of problems.

- designing algorithms
- proving limits
- classifying problems
- NP-completeness

Reductions for algorithm design

Def. Problem X reduces to problem Y if you can use an algorithm that solves Y to help solve X

Cost of solving X = M*(cost of solving Y) + cost of reduction.

number of times Y is used

Applications.

- designing algorithms: given algorithm for Y, can also solve X.
- proving limits: if X is hard, then so is Y.
- classifying problems: establish relative difficulty of problems.

Mentality: Since I know how to solve Y, can I use that algorithm to solve X?

Programmer's version: I have code for Y. Can I use it for X?

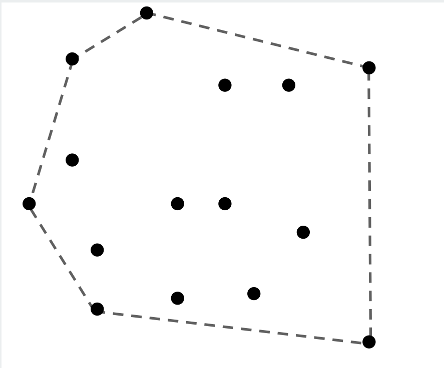
Reductions for algorithm design: convex hull

Sorting. Given N distinct integers, rearrange them in ascending order.

Convex hull. Given N points in the plane, identify the extreme points of the convex hull (in counter-clockwise order).

Claim. Convex hull reduces to sorting.

Pf. Graham scan algorithm.



convex hull

1251432
2861534
3988818
4190745
13546464
89885444

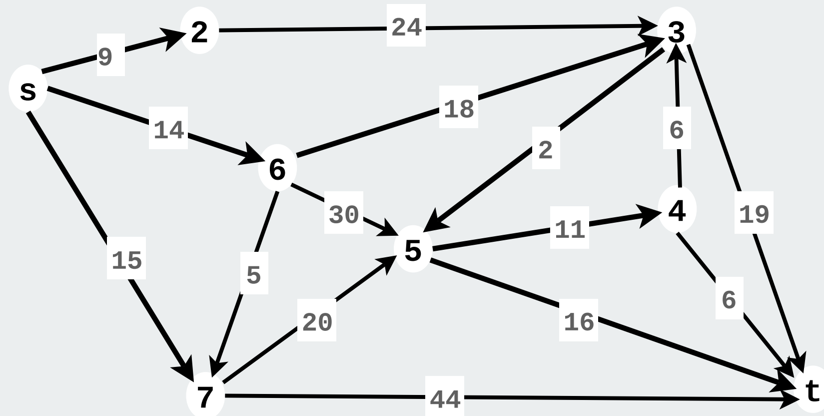
sorting

Cost of convex hull = cost of sort + cost of reduction

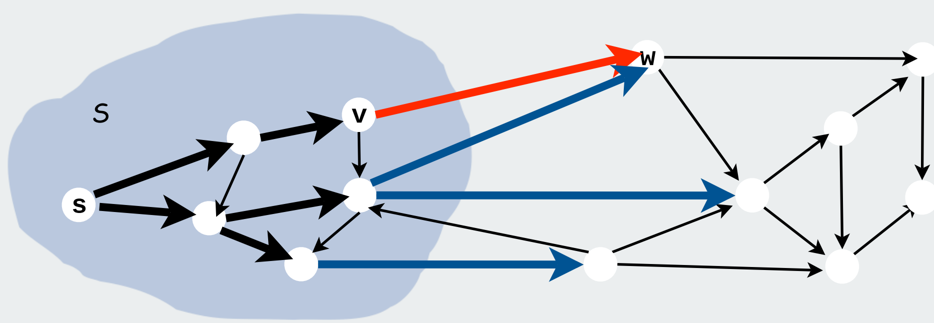
linearithmic linear

Reductions for algorithm design: shortest paths

Claim. Shortest paths reduces to path search in graphs (PFS)



Pf. Dijkstra's algorithm



Cost of shortest paths = cost of search + cost of reduction

linear length of path

Reductions for algorithm design: maxflow

Claim: Maxflow reduces to PFS (!)

A forward edge is an edge in the same direction of the flow

An backward edge is an edge in the opposite direction of the flow

An augmenting path is along which we can increase flow by adding flow on a forward edge or decreasing flow on a backward edge

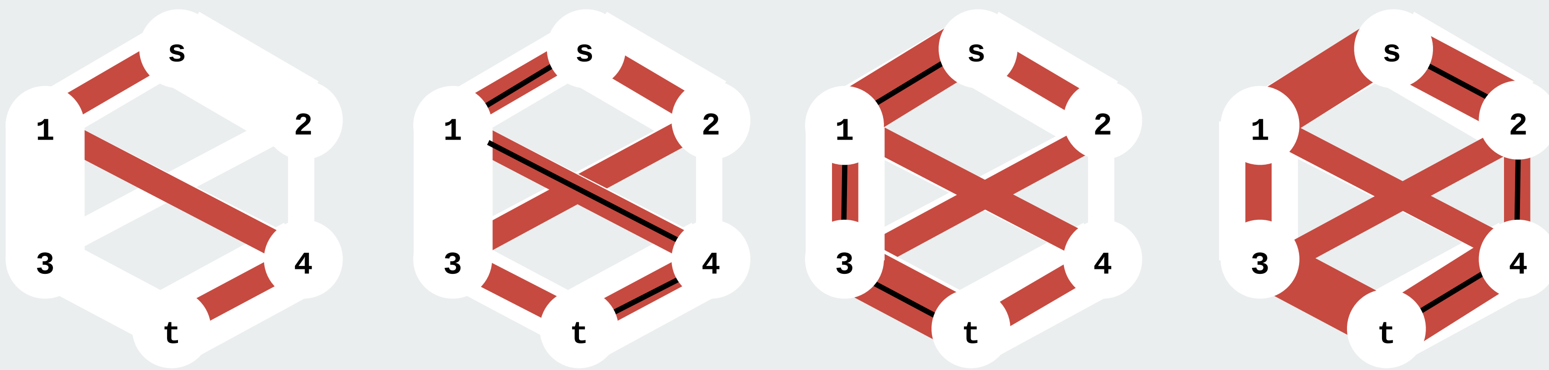
Theorem [Ford-Fulkerson] To find maxflow:

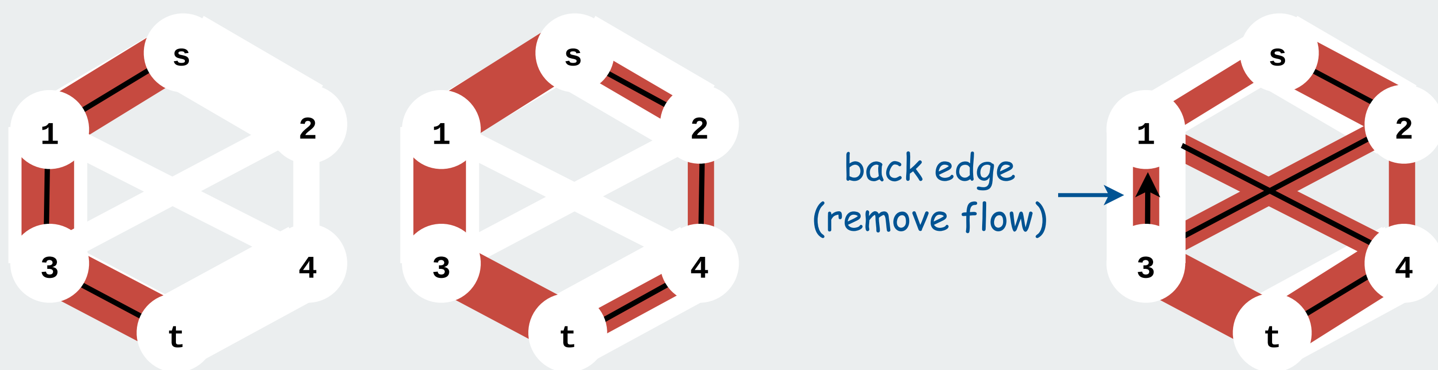
- increase flow along any augmenting path
- continue until no augmenting path can be found

Reduction is not linear because it requires multiple calls to PFS

Reductions for algorithm design: maxflow (continued)

Two augmenting-path sequences





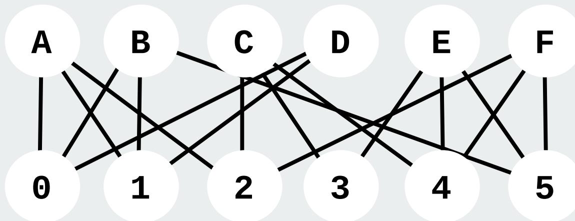
Cost of maxflow = M*(cost of PFS) + cost of reduction

linear linear

depends on path choice!

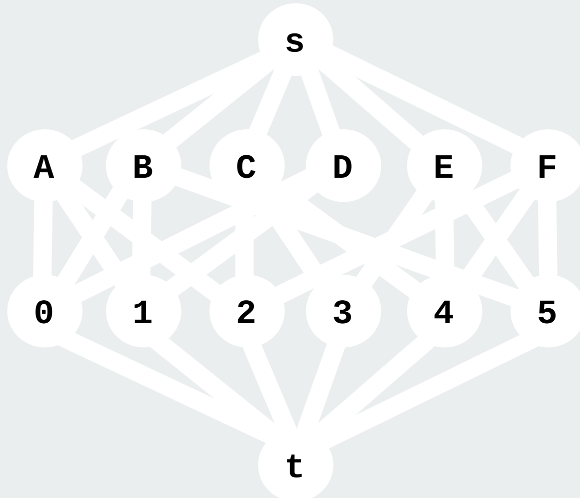
Reductions for algorithm design: bipartite matching

Bipartite matching reduces to maxflow



Proof:

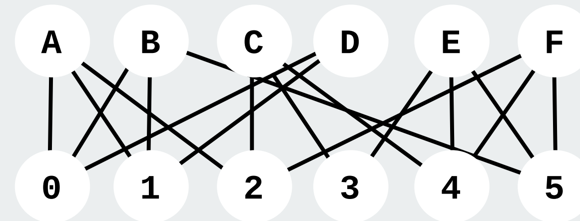
- construct new vertices s and t
- add edges from s to each vertex in one set
- add edges from each vertex in other set to t
- set all edge weights to 1
- find maxflow in resulting network
- matching is edges between two sets



Note: Need to establish that maxflow solution has all integer (0-1) values.

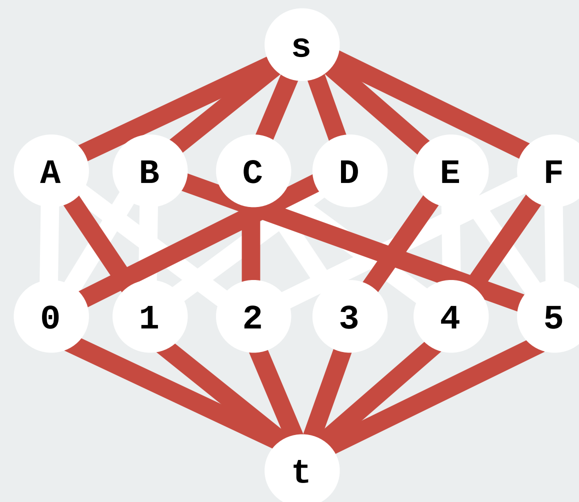
Reductions for algorithm design: bipartite matching

Bipartite matching reduces to maxflow

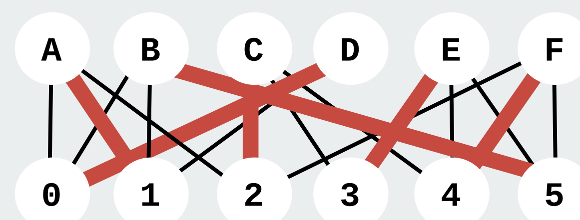


Proof:

- construct new vertices s and t
- add edges from s to each vertex in one set
- add edges from each vertex in other set to t
- set all edge weights to 1
- find maxflow in resulting network
- matching is edges between two sets



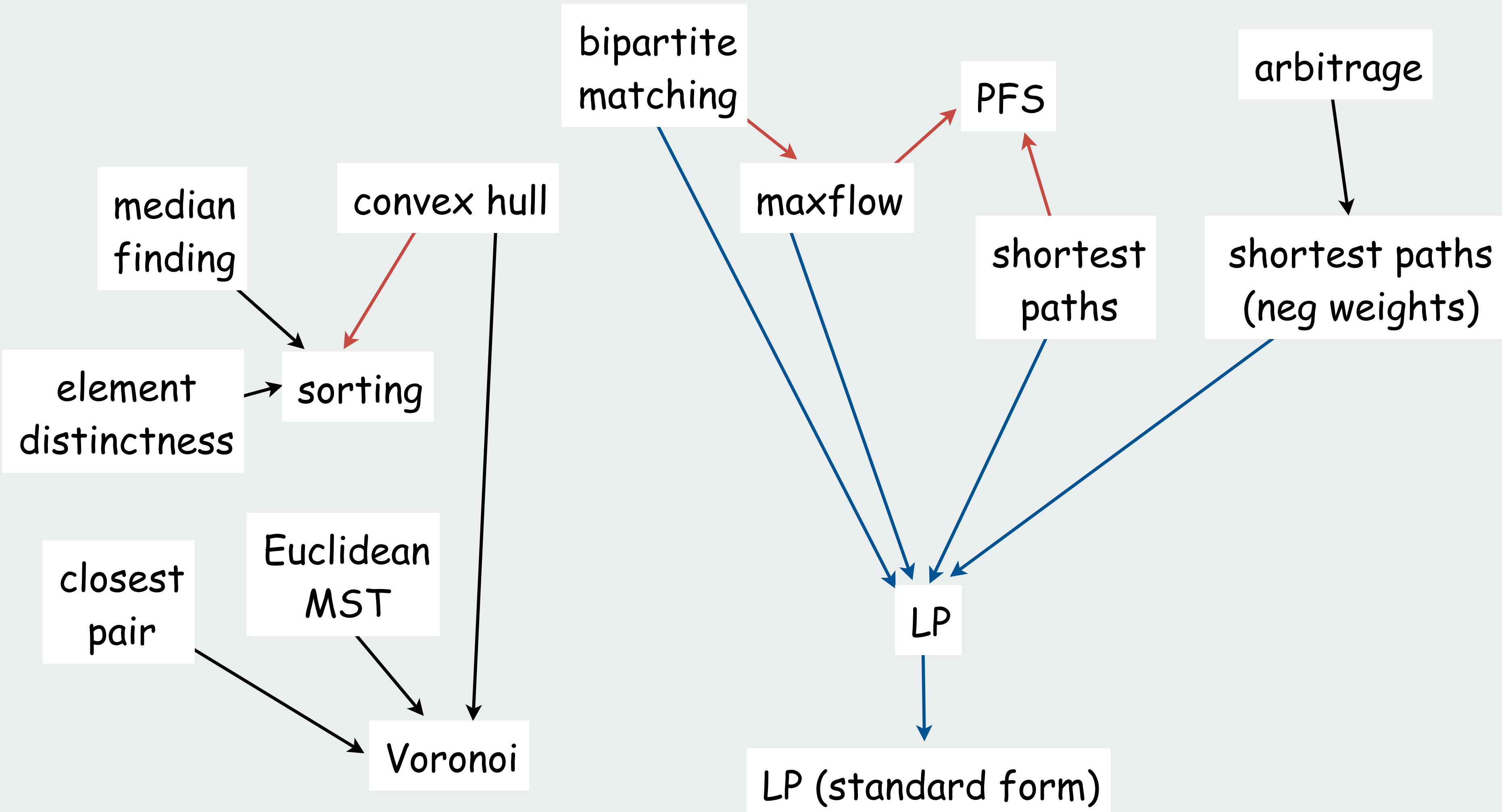
Note: Need to establish that maxflow solution
has all integer (0-1) values.



$$\text{Cost of matching} = \text{cost of maxflow} + \underset{\text{linear}}{\text{cost of reduction}}$$

Reductions for algorithm design: summary

Some reductions we have seen so far:



Reductions for algorithm design: a caveat

PRIME. Given an integer x (represented in binary), is x prime?

COMPOSITE. Given an integer x, does x have a nontrivial factor?

PRIME reduces to COMPOSITE

```
public static boolean isPrime(BigInteger x)
{
   if (isComposite(x)) return false;
   else                return true;
}
```

COMPOSITE reduces to PRIME

```
public static boolean isComposite(BigInteger x)
{
   if (isPrime(x)) return false;
   else            return true;
}
```

$$\begin{tikzcd} \text{PRIME} \arrow[d, shift right] \\ \text{COMPOSITE} \arrow[u, shift right] \end{tikzcd}$$

A possible real-world scenario:

- System designer specs the interfaces for project.
- Programmer A implements **`isComposite()`** using **`isPrime()`**.
- Programmer B implements **`isPrime()`** using **`isComposite()`**.
- Infinite reduction loop!

whose fault?

- designing algorithms
- **proving limits**
- classifying problems
- polynomial-time reductions
- NP-completeness

Linear-time reductions to prove limits

Def. Problem X linear reduces to problem Y if X can be solved with:

- linear number of standard computational steps for reduction
- one call to subroutine for Y.

Applications.

- designing algorithms: given algorithm for Y, can also solve X.
- proving limits: if X is hard, then so is Y.
- classifying problems: establish relative difficulty of problems.

Mentality:

If I could easily solve Y, then I could easily solve X
I can't easily solve X.
Therefore, I can't easily solve Y

Purpose of reduction is to establish that Y is hard

NOT intended for use as an algorithm

Proving limits on convex-hull algorithms

Lower bound on sorting: Sorting N integers requires Ω(N log N) steps.

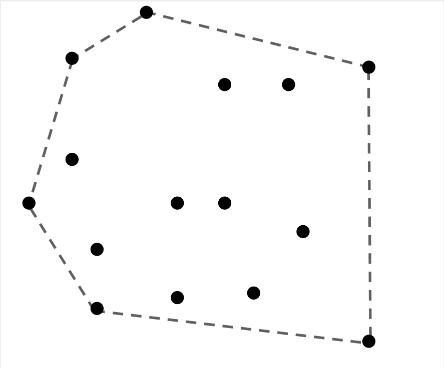
need "quadratic decision tree" model of computation that allows tests of the form $x_i < x_j$ or $(x_j - x_i)(y_k - y_i) - (y_j - y_i)(x_k - x_i) < 0$

Claim. SORTING reduces to CONVEX HULL [see next slide].

Consequence.
Any ccw-based convex hull algorithm requires Ω(N log N) steps.

```
1251432
2861534
3988818
4190745
13546464
89885444
```

sorting

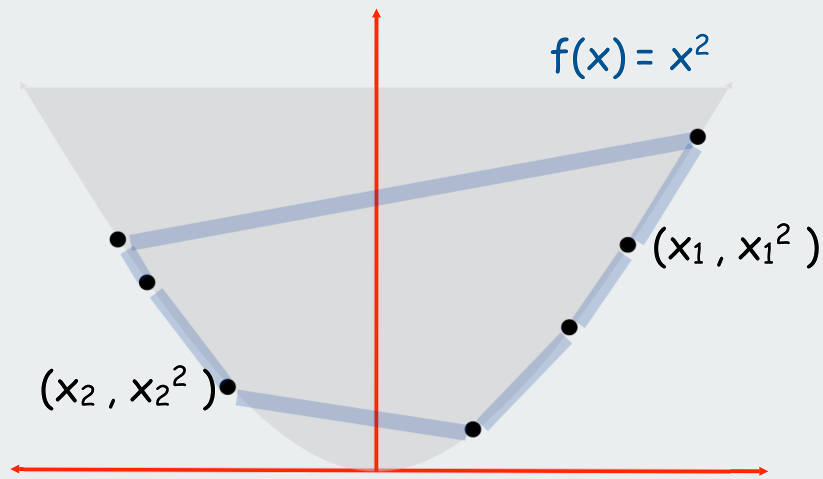


convex hull

Sorting linear-reduces to convex hull

Sorting instance. $X = \{ x_1, x_2, \ldots , x_N \}$

Convex hull instance. $P = \{ (x_1 , x_1^2), (x_2, x_2^2), \ldots , (x_N , x_N^2) \}$



Observation. Region $\{x : x^2 \geq x\}$ is convex $\Rightarrow$ all points are on hull.

Consequence. Starting at point with most negative x,
counter-clockwise order of hull points yields items in ascending order.

To sort X, find the convex hull of P.

3-SUM reduces to 3-COLLINEAR

3-SUM. Given N distinct integers, are there three that sum to 0?

3-COLLINEAR. Given N distinct points in the plane, are there 3 that all lie on the same line?

recall Assignment 2

Claim. 3-SUM reduces to 3-COLLINEAR.

see next two slides

Conjecture. Any algorithm for 3-SUM requires $\Omega(N^2)$ time.

Consequence. Sub-quadratic algorithm for 3-COLLINEAR unlikely.

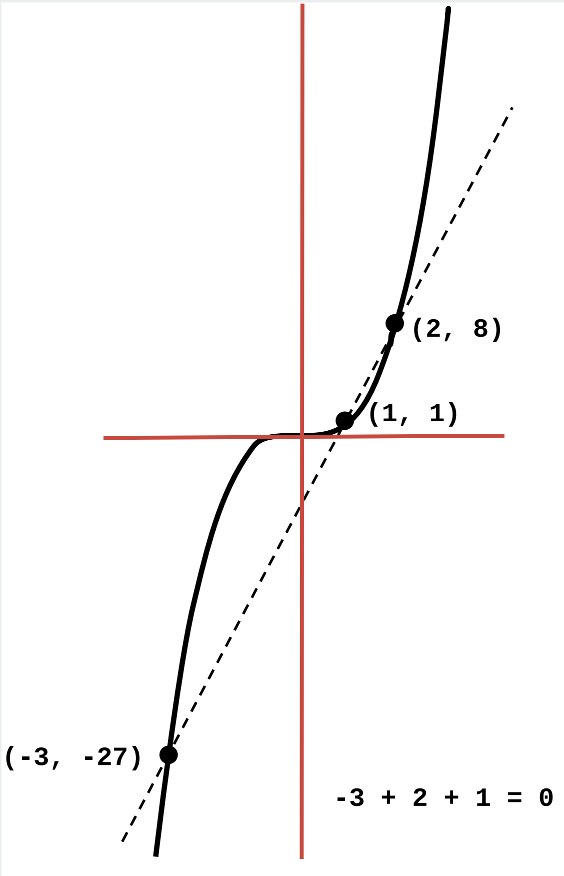
your $N^2 \log N$ algorithm from Assignment 2 was pretty good

3-SUM reduces to 3-COLLINEAR (continued)

Claim. 3-SUM $\le_L$ 3-COLLINEAR.

- 3-SUM instance: $x_1, x_2, \ldots, x_N$
- 3-COLLINEAR instance: $(x_1, x_1^3), (x_2, x_2^3), \ldots, (x_N, x_N^3)$

$f(x) = x^3$



Lemma. If a, b, and c are distinct, then a + b + c = 0 if and only if (a, a^3), (b, b^3), (c, c^3) are collinear.

Pf. [see next slide]

3-SUM reduces to 3-COLLINEAR (continued)

Lemma. If a, b, and c are distinct, then $a + b + c = 0$ if and only if (a, a^3), (b, b^3), (c, c^3) are collinear.

Pf. Three points (a, a^3), (b, b^3), (c, c^3) are collinear iff:

$(a^3 - b^3) / (a - b)$	$=$	$(b^3 - c^3) / (b - c)$	slopes are equal
$(a - b)(a^2 + ab + b^2) / (a - b)$	$=$	$(b - c)(b^2 + bc + c^2) / (b - c)$	factor numerators
$(a^2 + ab + b^2)$	$=$	$(b^2 + bc + c^2)$	a-b and b-c are nonzero
$a^2 + ab - bc - c^2$	$=$	0	collect terms
$(a - c)(a + b + c)$	$=$	0	factor
$a + b + c$	$=$	0	a-c is nonzero

Reductions for proving limits: summary

Establishing limits through reduction is an important tool in guiding algorithm design efforts

sorting → convex hull

Want to be convinced that no linear-time convex hull alg exists?
Hard way: long futile search for a linear-time algorithm
Easy way: reduction from sorting

3-SUM → 3-COLLINEAR

Want to be convinced that no subquadratic 3-COLLINEAR alg exists?
Hard way: long futile search for a subquadratic algorithm
Easy way: reduction from 3-SUM

- designing algorithms
- proving limits
- **classifying problems**
- NP-completeness

Reductions to classify problems

Def. Problem X **linear reduces** to problem Y if X can be solved with:
- Linear number of standard computational steps.
- One call to subroutine for Y.

Applications.
- Design algorithms: given algorithm for Y, can also solve X.
- Establish intractability: if X is hard, then so is Y.
- Classify problems: establish relative difficulty between two problems.

Ex: Sorting linear-reduces to convex hull.
Convex hull linear-reduces to sorting.
Thus, sorting and convex hull are equivalent

sorting

convex hull

not a loop because this reduction is not intended to give a sorting algorithm

Most often used to classify problems as either
- tractable (solvable in polynomial time)
- intractable (exponential time seems to be required)

Polynomial-time reductions

Def. Problem X polynomial reduces to problem Y if arbitrary instances of problem X can be solved using:

- Polynomial number of standard computational steps for reduction
- One call to subroutine for Y.

critical detail (not obvious why)

Notation. $X \leq_P Y$.

Ex. Any linear reduction is a polynomial reduction.

Ex. All algorithms for which we know poly-time algorithms poly-time reduce to one another.

Poly-time reduction of X to Y makes sense
only when X or Y is not known to have a poly-time algorithm

Polynomial-time reductions for classifying problems

Goal. Classify and separate problems according to relative difficulty.
- tractable problems: can be solved in polynomial time.
- intractable problems: seem to require exponential time.

Establish tractability. If $X \le_P Y$ and Y is tractable then so is X.
- Solve Y in polynomial time.
- Use reduction to solve X.

Establish intractability. If $Y \le_P X$ and Y is intractable, then so is X.
- Suppose X can be solved in polynomial time.
- Then so could Y (through reduction).
- Contradiction. Therefore X is intractable.

Transitivity. If $X \le_P Y$ and $Y \le_P Z$ then $X \le_P Z$.

Ex: all problems that reduce to LP are tractable

3-satisfiability

Literal: A Boolean variable or its negation. x_i or $\neg x_i$

Clause. A disjunction of 3 distinct literals. $C_j = (x_1 \vee \neg x_2 \vee x_3)$

Conjunctive normal form. A propositional formula Φ that is the conjunction of clauses. $CNF = (C_1 \wedge C_2 \wedge C_3 \wedge C_4)$

3-SAT. Given a CNF formula Φ consisting of k clauses over n literals, does it have a satisfying truth assignment?

yes instance

$$(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)$$

x_1	x_2	x_3	x_4
T	T	F	T

$$(\neg T \vee T \vee F) \wedge (T \vee \neg T \vee F) \wedge (\neg T \vee \neg T \vee \neg F) \wedge (\neg T \vee \neg T \vee T) \wedge (\neg T \vee F \vee T)$$

no instance

$$(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)$$

Applications: Circuit design, program correctness, [many others]

3-satisfiability is intractable

Good news: easy algorithm to solve 3-SAT

[check all possible solutions]

Bad news: running time is exponential in input size.

[there are 2^n possible solutions]

Worse news:

no algorithm that guarantees subexponential running time is known

Implication:

- suppose 3-SAT poly-reduces to a problem A
- poly-time algorithm for A would imply poly-time 3-SAT algorithm
- we suspect that no poly-time algorithm exists for A!

Want to be convinced that a new problem is intractable?

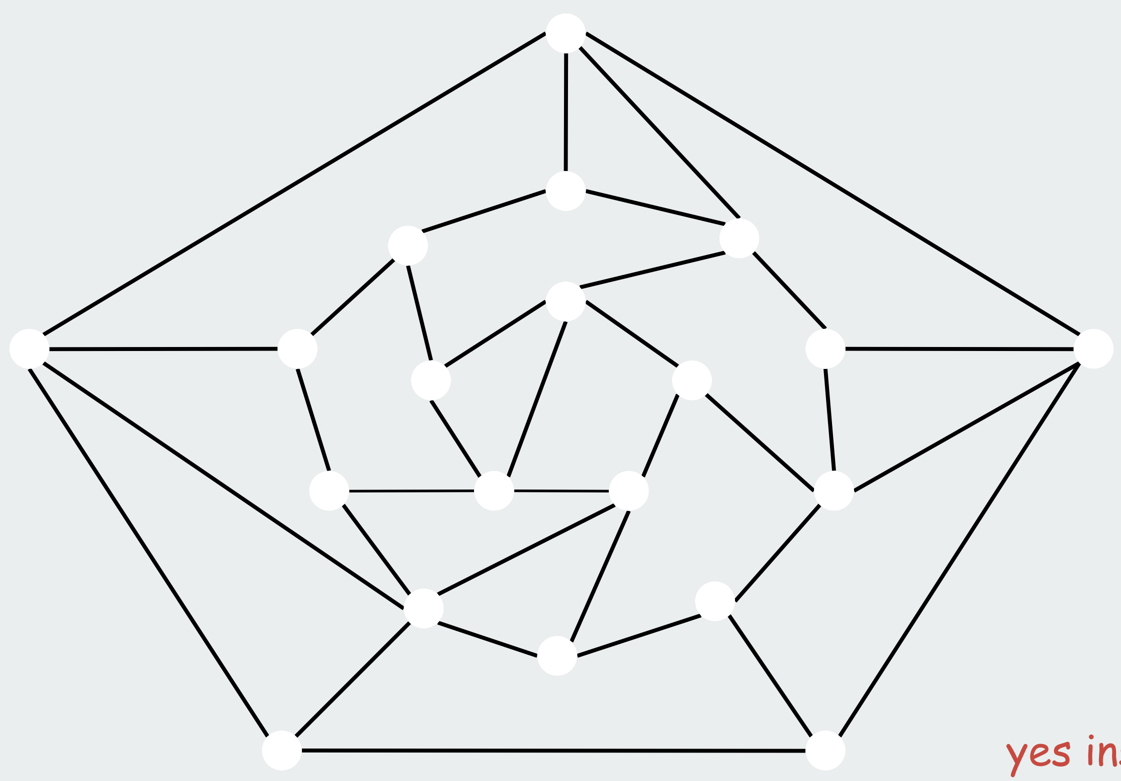
Hard way: long futile search for an efficient algorithm (as for 3-SAT)

Easy way: reduction from a known intractable problem (such as 3-SAT)

hence, intricate reductions are common

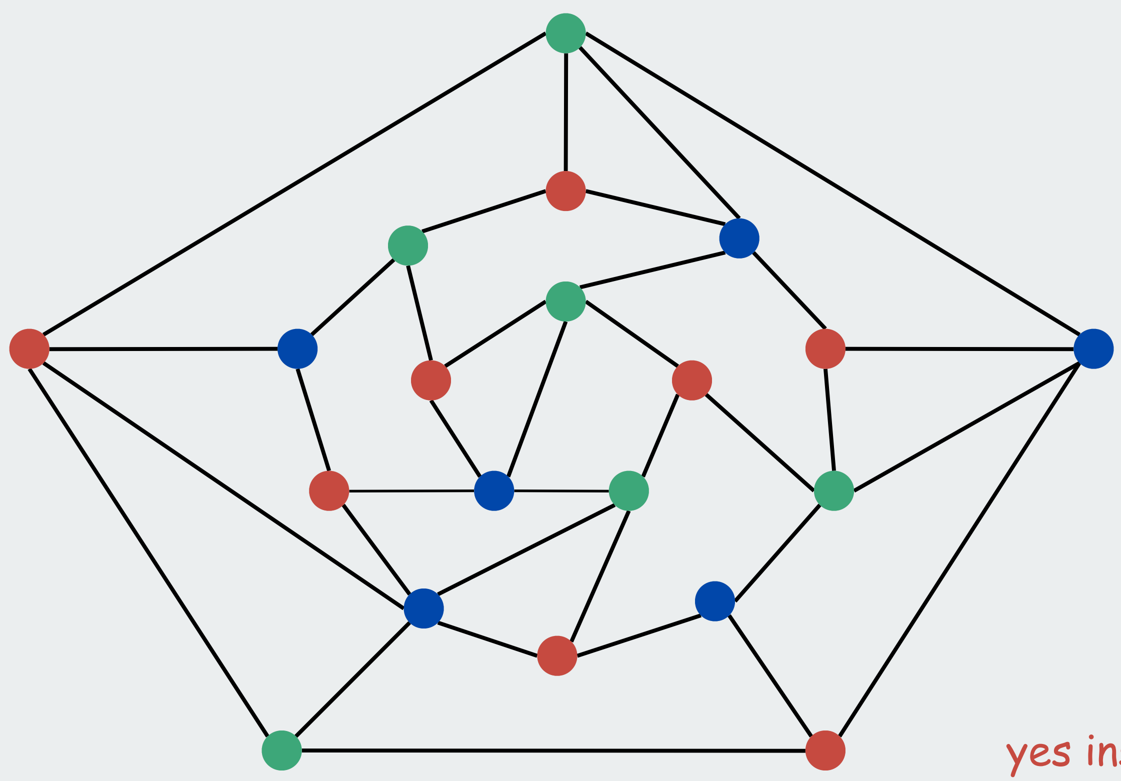
Graph 3-colorability

3-COLOR. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?



yes instance

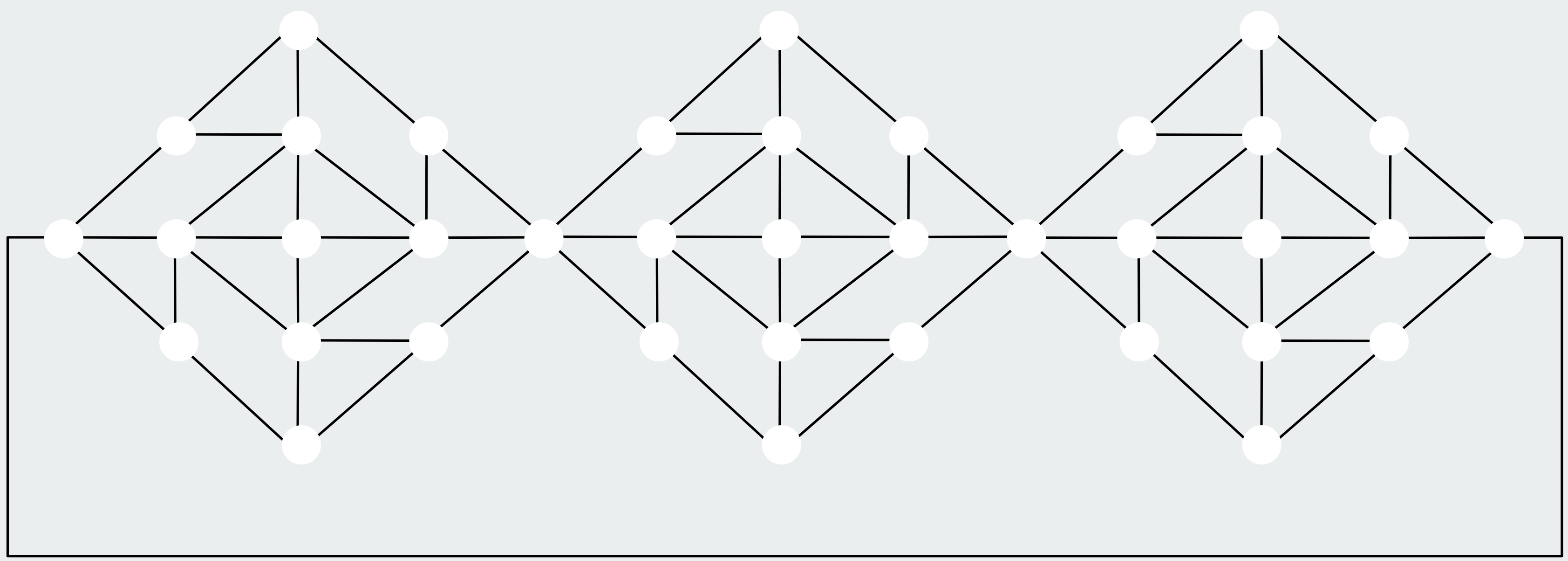
3-COLOR. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?



yes instance

Graph 3-colorability

3-COLOR. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?



no instance

3-satisfiability reduces to graph 3-colorability

Claim. 3-SAT $\leq_P$ 3-COLOR.

Pf. Given 3-SAT instance Φ, we construct an instance of 3-COLOR that is 3-colorable if and only if Φ is satisfiable.

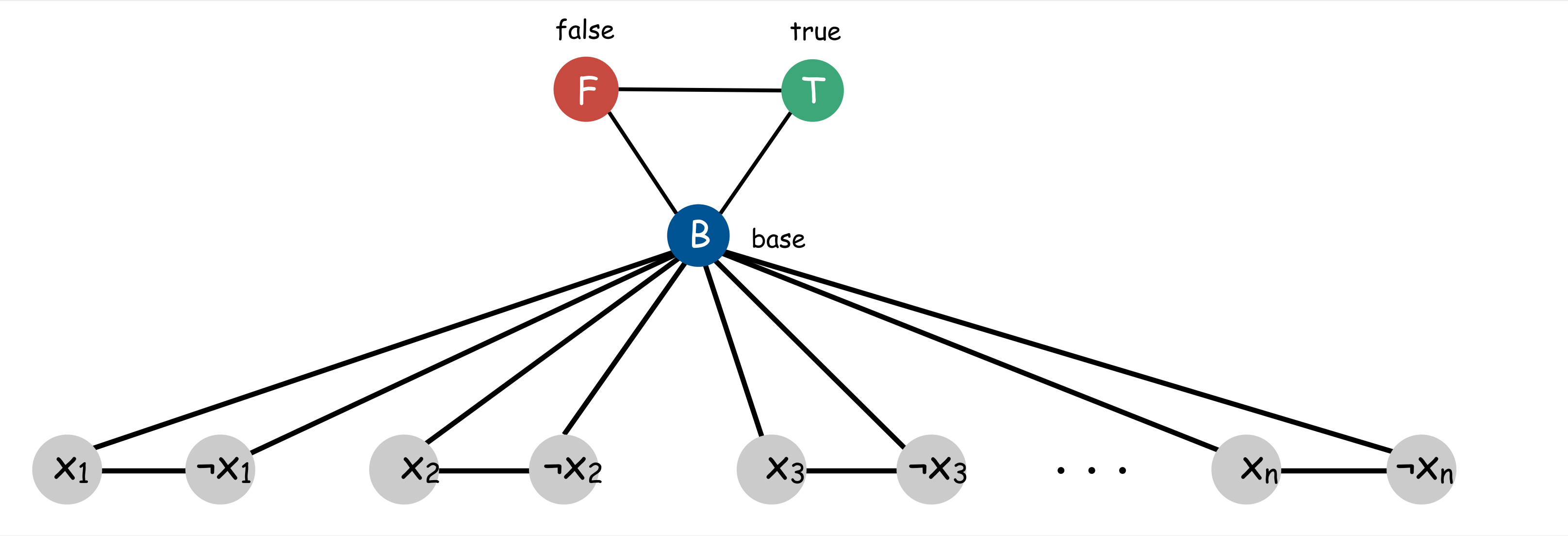
Construction.

(i) Create one vertex for each literal and 3 vertices F T B

(ii) Connect F T B in a triangle and connect each literal to B

(iii) Connect each literal to its negation.

(iv) For each clause, attach a 6-vertex gadget [details to follow].

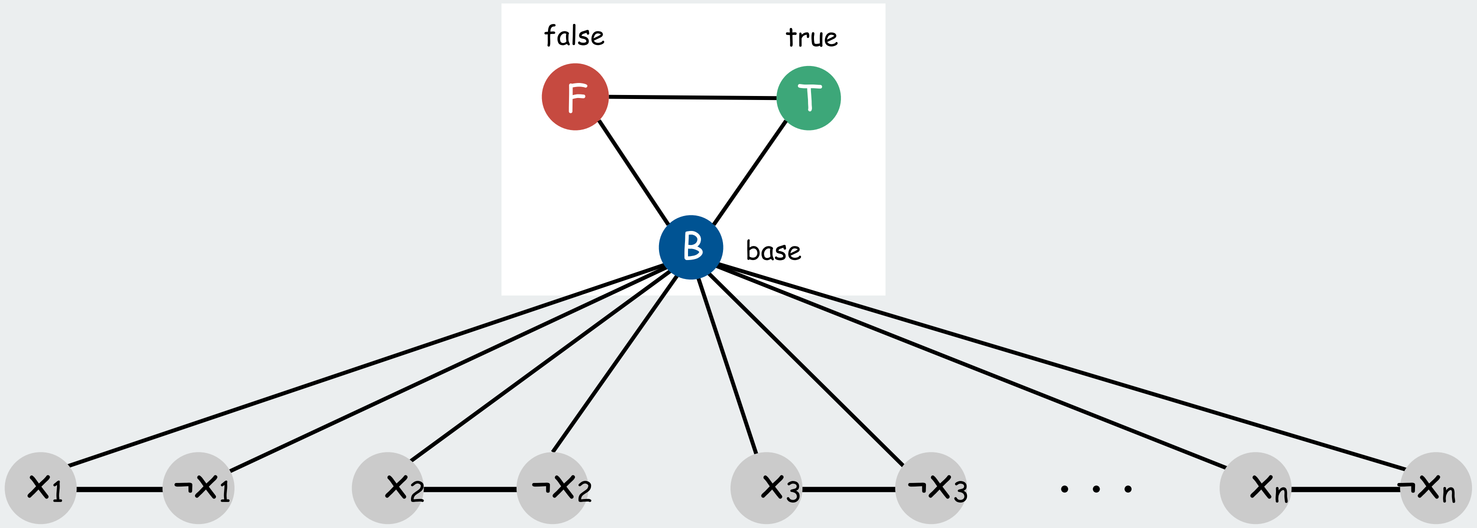


3-satisfiability reduces to graph 3-colorability

Claim. If graph is 3-colorable then Φ is satisfiable..

Pf.

- Consider assignment where F corresponds to false and T to true .
- (ii) [triangle] ensures each literal is true or false.

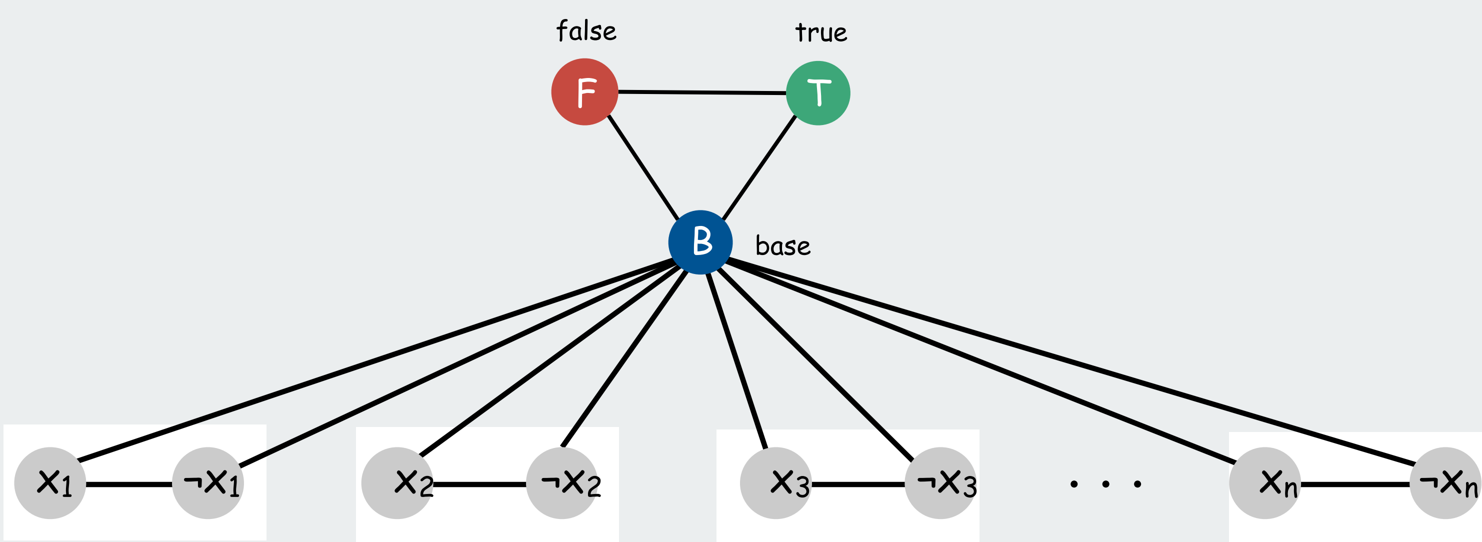


3-satisfiability reduces to graph 3-colorability

Claim. If graph is 3-colorable then Φ is satisfiable..

Pf. $\Rightarrow$ Suppose graph is 3-colorable.

- Consider assignment where F corresponds to false and T to true .
- (ii) [triangle] ensures each literal is true or false.
- (iii) ensures a literal and its negation are opposites.



3-satisfiability reduces to graph 3-colorability

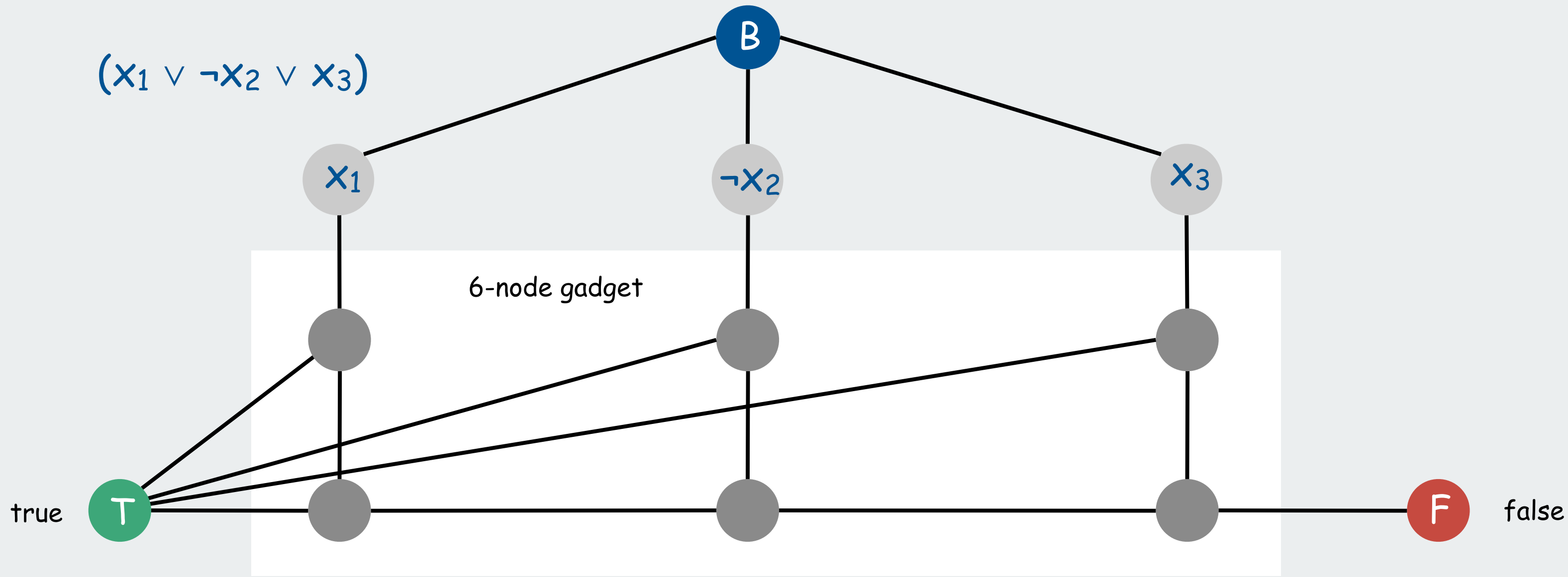
Claim. If graph is 3-colorable then Φ is satisfiable.

Pf.

- Consider assignment where F corresponds to false and T to true .
- (ii) [triangle] ensures each literal is true or false.
- (iii) ensures a literal and its negation are opposites.
- (iv) [gadget] ensures at least one literal in each clause is true.

stay tuned

$(x_1 \vee \neg x_2 \vee x_3)$



3-satisfiability reduces to graph 3-colorability

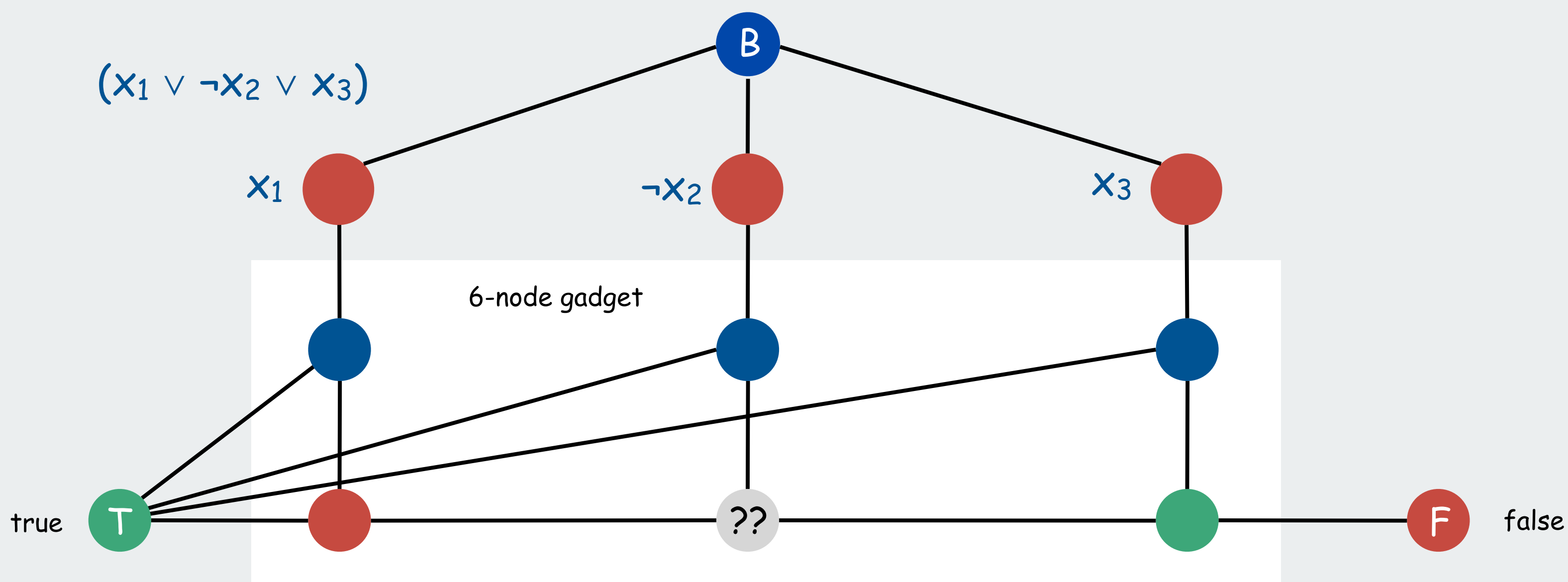
Claim. If graph is 3-colorable then Φ is satisfiable.

Pf.

- Consider assignment where F corresponds to false and F to true .
- (ii) [triangle] ensures each literal is true or false.
- (iii) ensures a literal and its negation are opposites.
- (iv) [gadget] ensures at least one literal in each clause is true.

Therefore, Φ is satisfiable.

$(x_1 \vee \neg x_2 \vee x_3)$



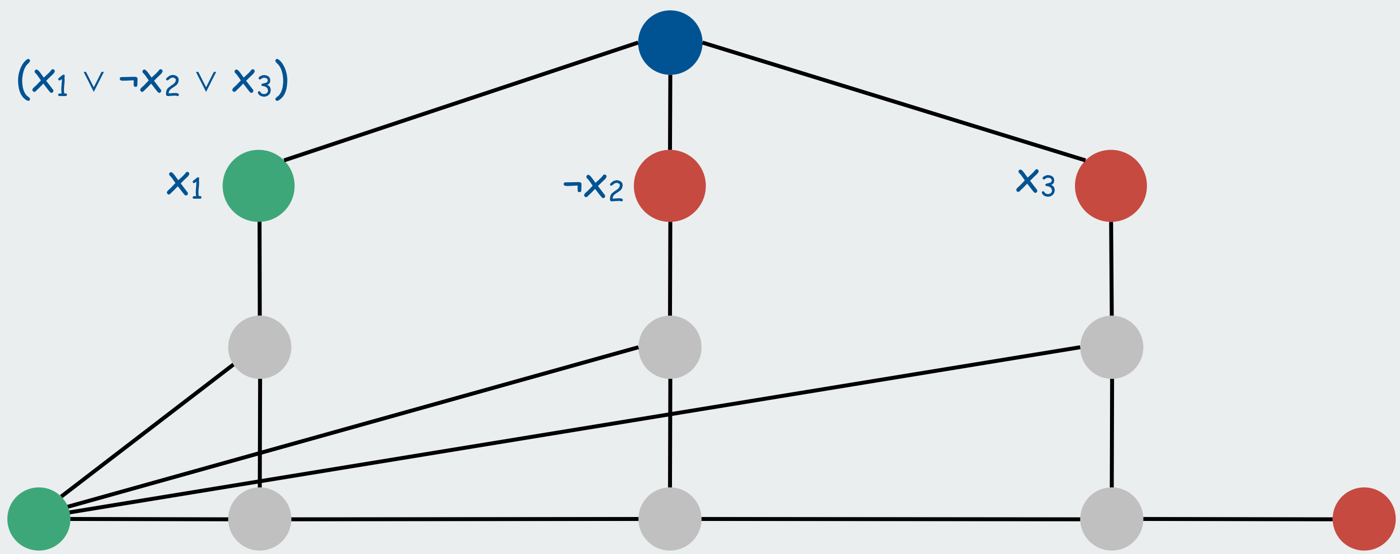
3-satisfiability reduces to graph 3-colorability

Claim. If Φ is satisfiable then graph is 3-colorable.

Pf.

at least one in each clause

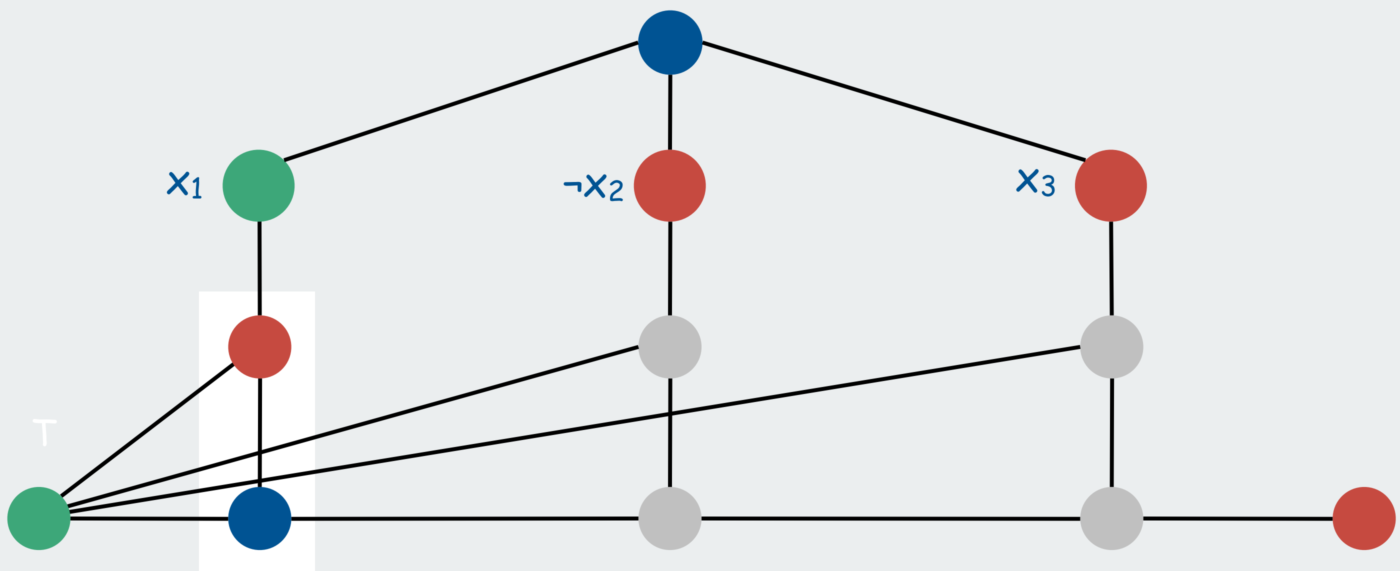
- Color nodes corresponding to false literals and to true literals .



Claim. If Φ is satisfiable then graph is 3-colorable.

Pf.

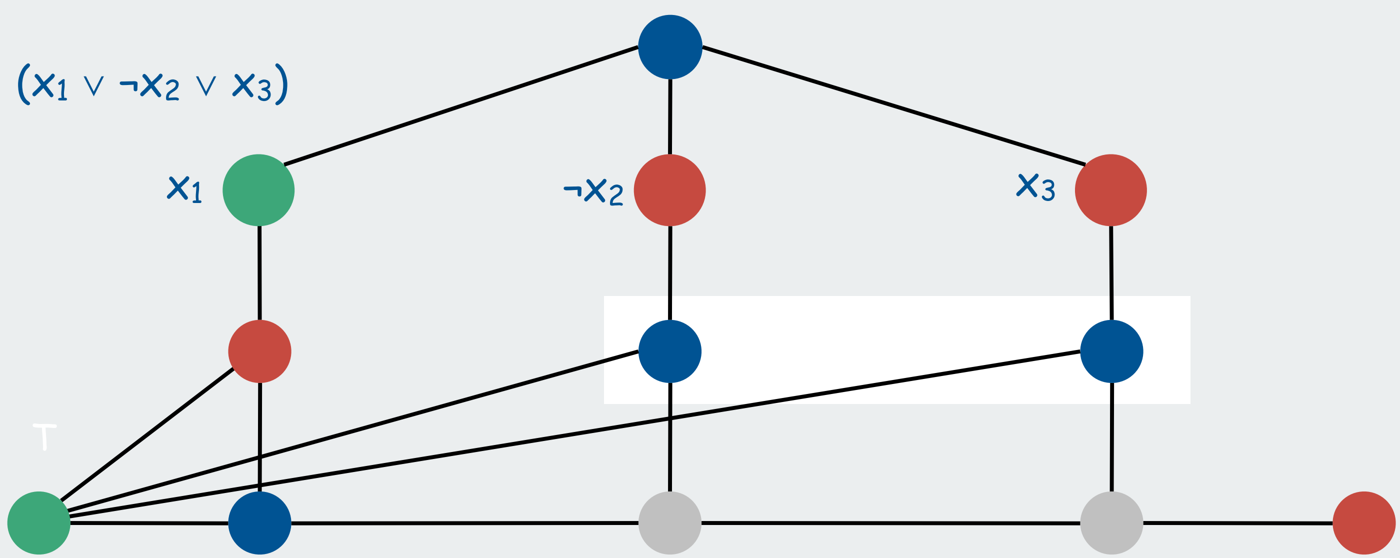
- Color nodes corresponding to false literals and to true literals .
- Color vertex below one vertex , and vertex below that .



Claim. If Φ is satisfiable then graph is 3-colorable.

Pf.

- Color nodes corresponding to false literals and to true literals .
- Color vertex below one vertex , and vertex below that .
- Color remaining middle row vertices .



3-satisfiability reduces to graph 3-colorability

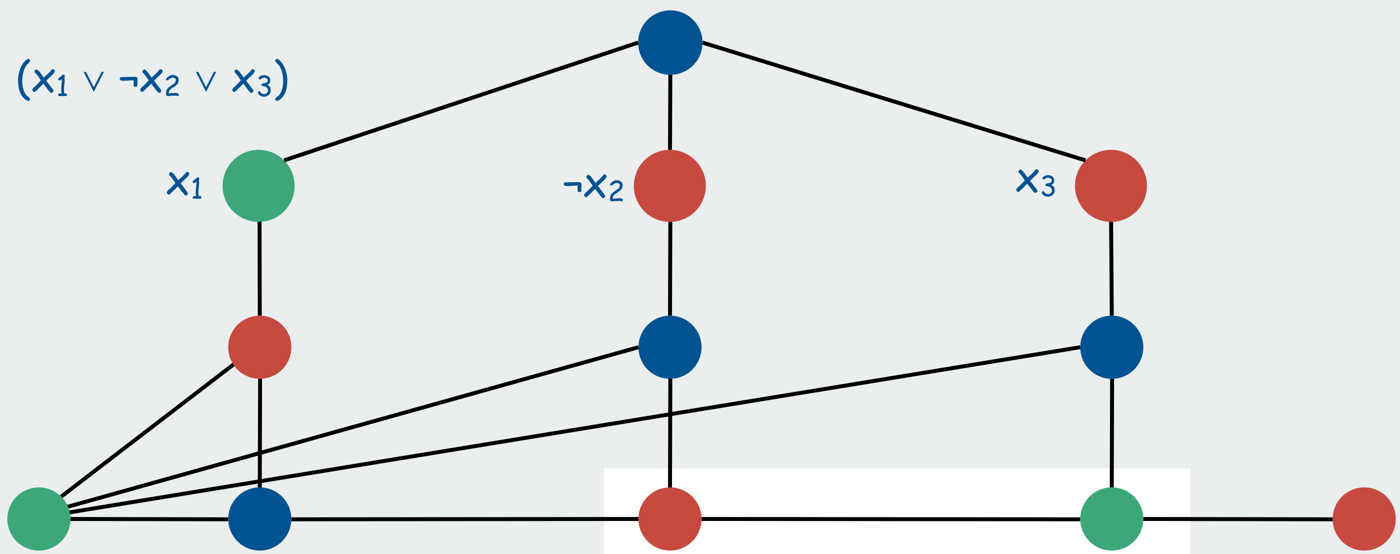
Claim. If Φ is satisfiable then graph is 3-colorable.

Pf.

- Color nodes corresponding to false literals [red] and to true literals [green].
- Color vertex below one [green] vertex [red], and vertex below that [blue].
- Color remaining middle row vertices [blue].
- Color remaining bottom vertices [green] or [red] as forced.

Works for all gadgets, so graph is 3-colorable. ▪

$(x_1 \vee \neg x_2 \vee x_3)$



3-satisfiability reduces to graph 3-colorability

Claim. 3-SAT $\leq_P$ 3-COLOR.

Pf. Given 3-SAT instance Φ, we construct an instance of 3-COLOR that is 3-colorable if and only if Φ is satisfiable.

Construction.

(i) Create one vertex for each literal.

(ii) Create 3 new vertices T, F, and B; connect them in a triangle, and connect each literal to B.

(iii) Connect each literal to its negation.

(iv) For each clause, attach a gadget of 6 vertices and 13 edges

Conjecture: No polynomial-time algorithm for 3-SAT

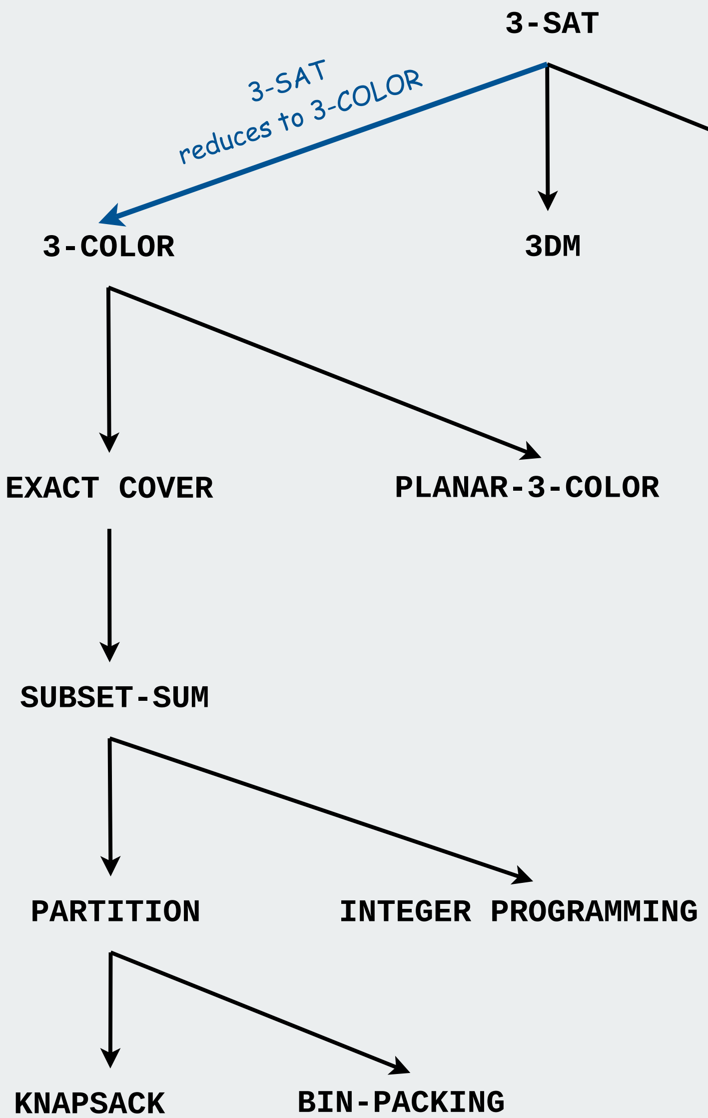
Implication: No polynomial-time algorithm for 3-COLOR.

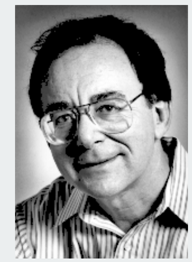
Reminder

Construction is not intended for use, just to prove 3-COLOR difficult

- designing algorithms
- proving limits
- classifying problems
- polynomial-time reductions
- **NP-completeness**

More Poly-Time Reductions





Dick Karp
'85 Turing award

Conjecture: no poly-time algorithm for 3-SAT.
(and hence none of these problems)

Cook's Theorem

NP: set of problems solvable in polynomial time
by a nondeterministic Turing machine

THM. Any problem in NP $\leq_P$ 3-SAT.

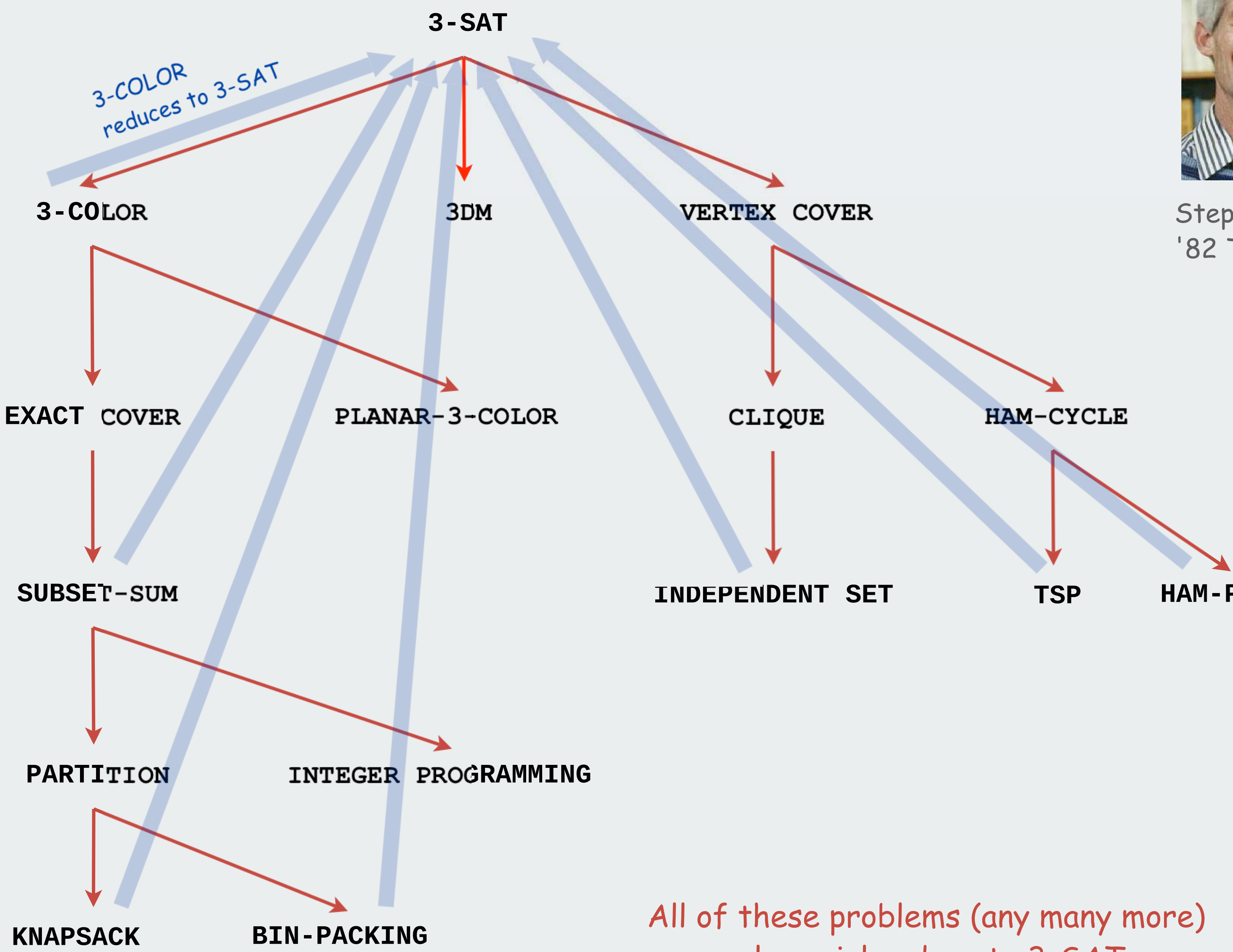
Pf sketch.

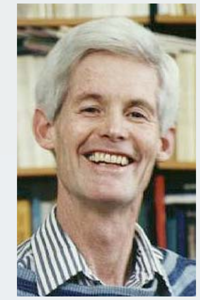
Each problem P in NP corresponds to a TM M that accepts or rejects
any input in time polynomial in its size
Given M and a problem instance I, construct an instance of 3-SAT
that is satisfiable iff the machine accepts I.

Construction.

- Variables for every tape cell, head position, and state at every step.
- Clauses corresponding to each transition.
- [many details omitted]

Implications of Cook's theorem



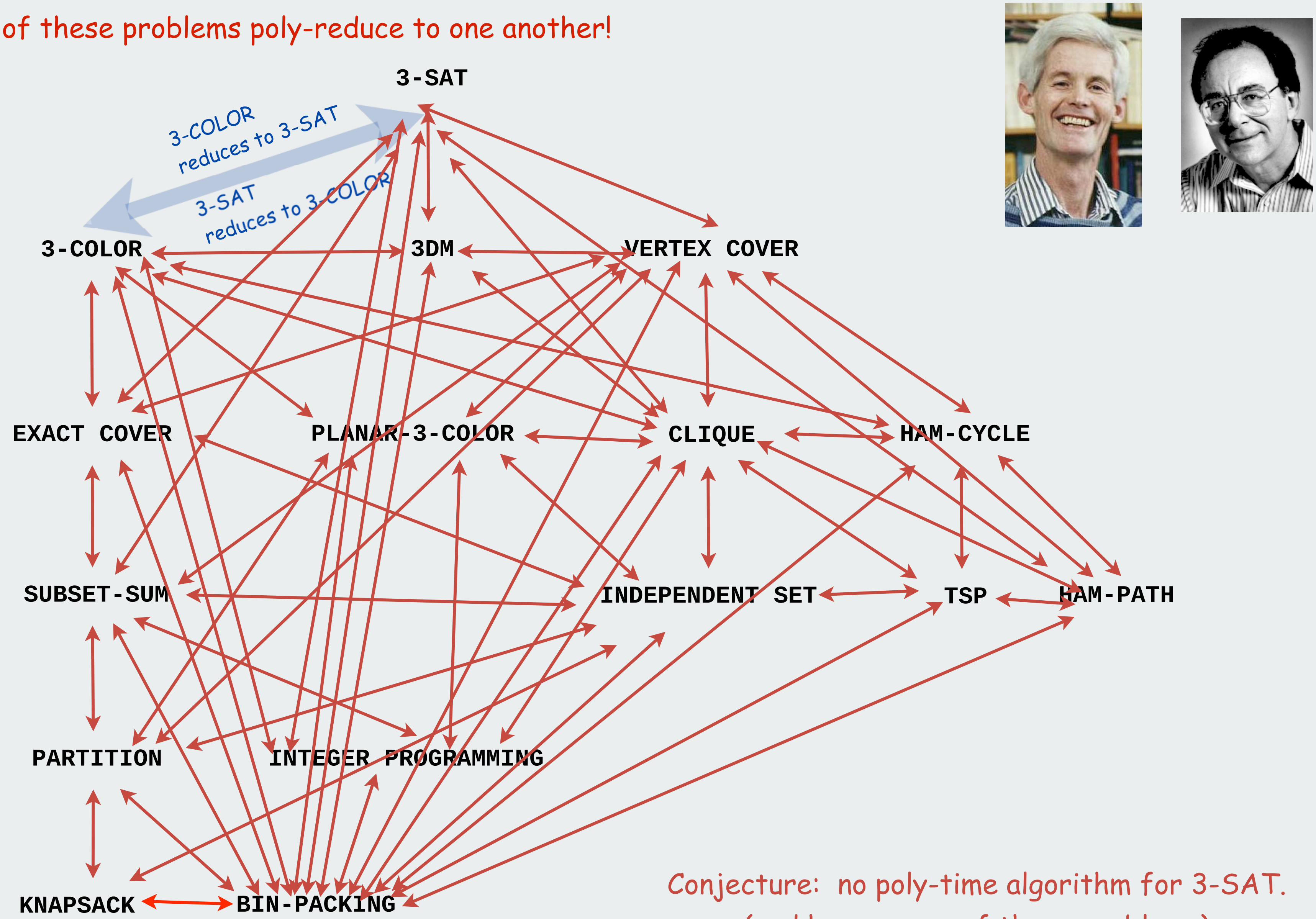


Stephen Cook
'82 Turing award

All of these problems (any many more) polynomial reduce to 3-SAT.

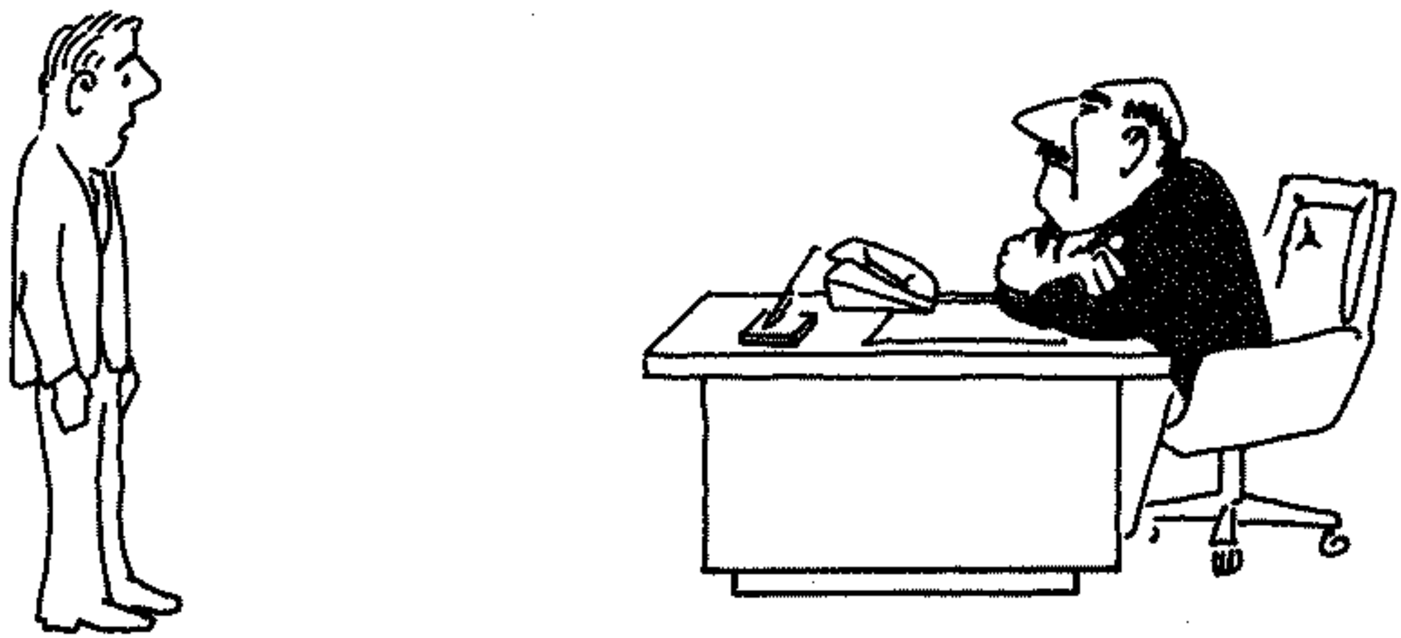
Implications of Karp + Cook

All of these problems poly-reduce to one another!

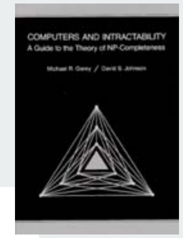


Conjecture: no poly-time algorithm for 3-SAT.
(and hence none of these problems)

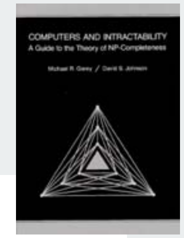
Poly-Time Reductions: Implications



“I can’t find an efficient algorithm, I guess I’m just too dumb.”

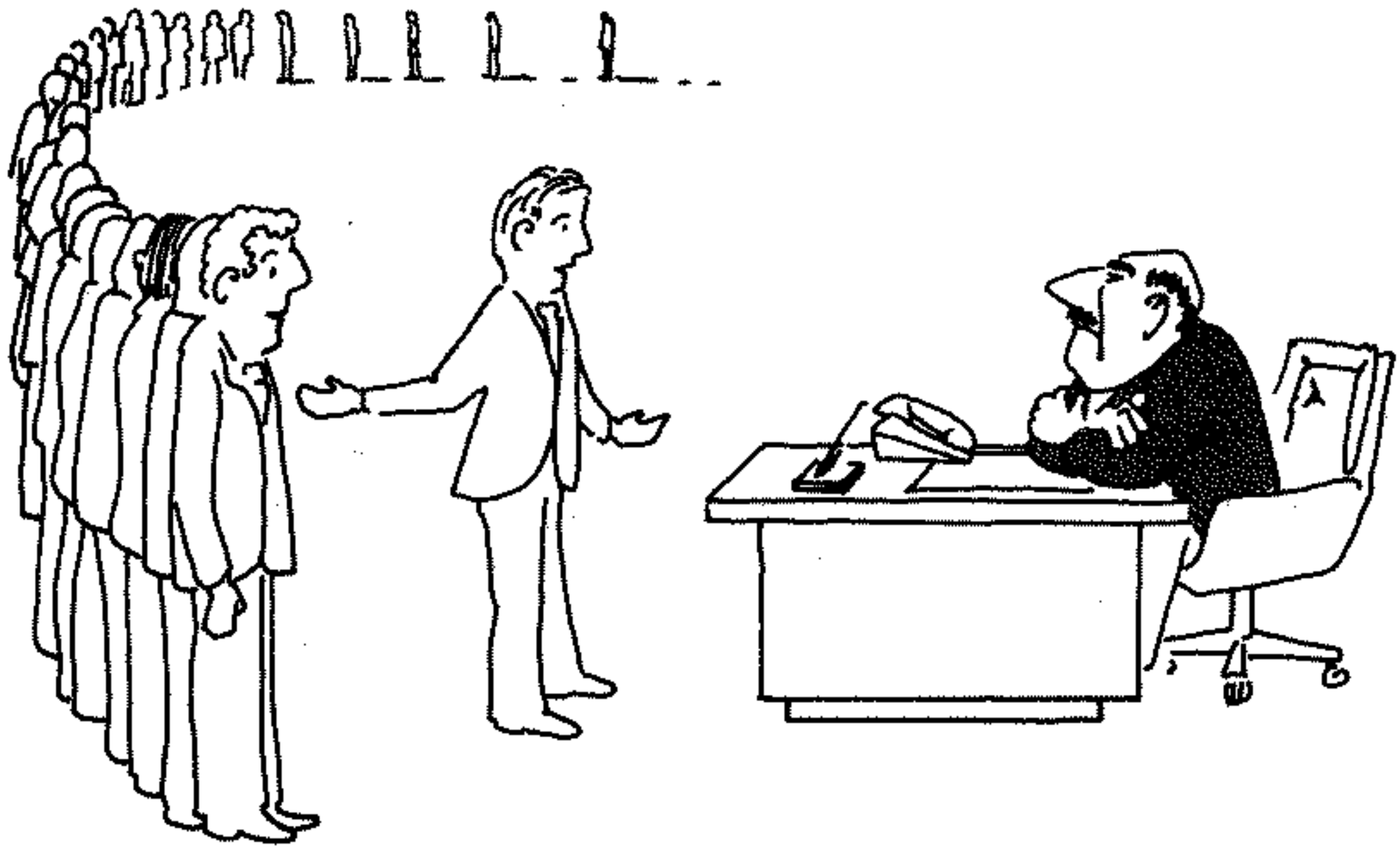


Poly-Time Reductions: Implications

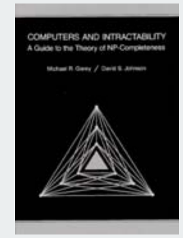


"I can't find an efficient algorithm, because no such algorithm is possible!"

Poly-Time Reductions: Implications



"I can't find an efficient algorithm, but neither can all these famous people."



Summary

Reductions are important in theory to:

- Establish tractability.
- Establish intractability.
- Classify problems according to their computational requirements.

Reductions are important in practice to:

- Design algorithms.
- Design reusable software modules.
  - stack, queue, sorting, priority queue, symbol table, set, graph
  - shortest path, regular expressions, linear programming
- Determine difficulty of your problem and choose the right tool.
  - use exact algorithm for tractable problems
  - use heuristics for intractable problems