Reductions

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

Desiderata

Desiderata. Classify problems according to their computational requirements.

Frustrating news. Huge number of fundamental problems have defied classification for decades.

Desiderata

Desiderata. Classify problems according to their computational requirements.

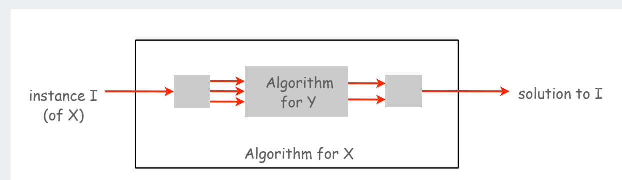
Desiderata'. Suppose we could (couldn't) solve problem X efficiently. What else could (couldn't) we solve efficiently?

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

- Cost of solving X = cost of solving Y + cost of reduction.



Ex. Euclidean MST reduces to Voronoi.
To solve Euclidean MST on N points
- solve Voronoi
- 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 = cost of solving Y + cost of reduction.

Consequences.

- algorithm design: 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.

designing algorithms
proving limits
classifying problems
poly-time reductions
NP-completeness

Linear-time reductions

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.
- Notation: $X \leq_L Y$.

Some familiar examples.

- Median $\leq_L$ sorting.
- Element distinctness $\leq_L$ sorting.
- Closest pair $\leq_L$ Voronoi.
- Euclidean MST $\leq_L$ Voronoi.
- Arbitrage $\leq_L$ Negative cycle detection.
- Linear programming $\leq_L$ Linear programming in std form.

Linear-time reductions for algorithm design

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

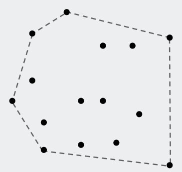
Convex Hull

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 linear reduces to sorting.

Pf. Graham scan algorithm.



convex hull

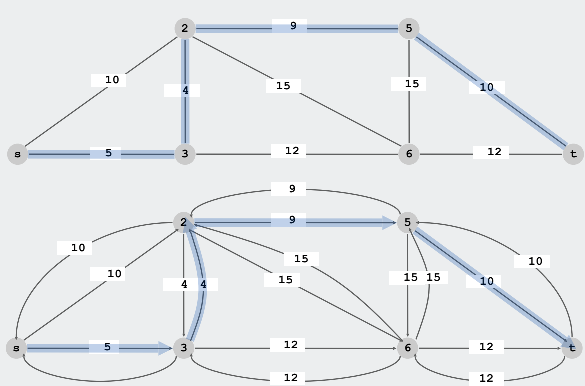
1251432
2861534
3988818
4190745
13546464
89885444

sorting

Shortest Paths on Graphs and Digraphs

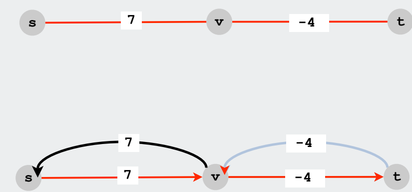
Claim. Undirected shortest path (with nonnegative weights) linearly reduces to directed shortest path.

Pf. Replace each undirected edge by two directed edges.



Shortest Paths with negative weights

Caveat. Reduction invalid in networks with negative weights (even if no negative cycles).



Remark. Can still solve shortest path problem in undirected graphs if no negative cycles, but need more sophisticated techniques.

reduce to weighted non-bipartite matching (!)

designing algorithms

proving limits

classifying problems

poly-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

Proving limits on convex-hull algorithms

Lower bound on sorting: Sorting N integers requires $\Omega(N \log N)$ steps.

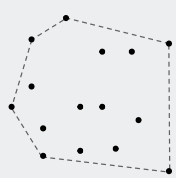
need "quadratic decision tree" model of computation that allows tests of the form $x_i < x_j$ or $(x_j - x_i)(y_k - y_i) - (y_j - y_i)(x_k - x_i) < 0$

Claim. Sorting linear-reduces to convex hull [see next slide].

Theorem.
Any ccw-based convex hull algorithm requires $\Omega(N \log N)$ steps.

1251432
2861534
3988818
4190745
13546464
89885444

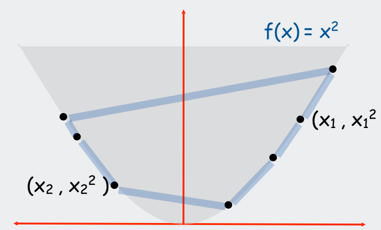
sorting



convex hull

Sorting linear-reduces to convex hull

Sorting instance. $x_1, x_2, \ldots, x_N$
Convex hull instance. $(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_N, x_N^2)$



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.

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 $\leq_L$ 3-COLLINEAR.

see next two slides

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

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

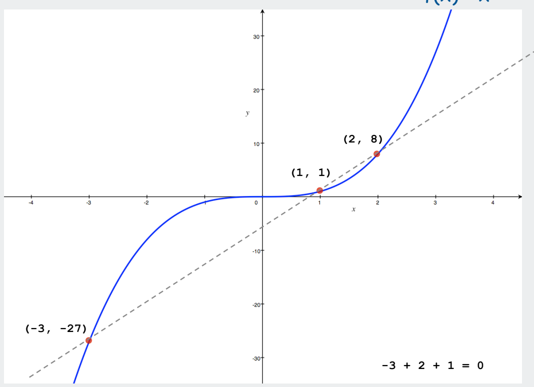
your $N^2 \log N$ algorithm from Assignment 2 was pretty good

3-SUM Reduces to 3-COLLINEAR

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

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

$f(x) = x^3$



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

3-SUM Reduces to 3-COLLINEAR

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

Linear Time Reductions

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.

Consequences.

- 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.

Primality and Compositeness

PRIME. Given an integer x (represented in binary), is x prime?
COMPOSITE. Given an integer x, does x have a nontrivial factor?

Claim. PRIME $\leq_L$ COMPOSITE.

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


Primality and Compositeness

PRIME. Given an integer x (represented in binary), is x prime?
COMPOSITE. Given an integer x, does x have a nontrivial factor?

Claim. COMPOSITE $\leq_L$ PRIME.

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

Conclusion. COMPOSITE and PRIME have same complexity.

Reduction Gone Wrong

Caveat.
- System designer specs the interfaces for project.
- One programmer might implement `isComposite()` using `isPrime()`.
- Other programmer might implement `isPrime()` using `isComposite()`.
- Be careful to avoid infinite reduction loops in practice.

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

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


Poly-Time Reduction

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.

Notation. $X \leq_P Y$.

Ex. Assignment problem $\leq_P$ LP ← last lecture
Ex. 3-SAT $\leq_P$ 3-COLOR. ← stay tuned
Ex. Any linear reduction.

designing algorithms
proving limits
classifying problems
poly-time reductions
NP-completeness

Poly-time reductions

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

Establish tractability. If $X \le_P Y$ and Y can be solved in poly-time, then X can be solved in poly-time.

Establish intractability. If $Y \le_P X$ and Y cannot be solved in poly-time, then X cannot be solved in poly-time.

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

Assignment Problem

Assignment problem. Assign n jobs to n machines to minimize total cost, where c_{ij} = cost of assigning job j to machine i.

	1'	2'	3'	4'	5'
1	3	8	9	15	10
2	4	10	7	16	14
3	9	13	11	19	10
4	8	13	12	20	13
5	1	7	5	11	9

cost = 3 + 10 + 11 + 20 + 9 =53

	1'	2'	3'	4'	5'
1	3	8	9	15	10
2	4	10	7	16	14
3	9	13	11	19	10
4	8	13	12	20	13
5	1	7	5	11	9

cost = 8 + 7 + 20 + 8 + 11 = 44

Applications. Match jobs to machines, match personnel to tasks, match Princeton students to writing seminars.

Assignment problem reduces to LP

N^2 variables one corresponding to each cell

2N equations one per row one per column

maximize

$$c_{11}x_{11} + c_{12}x_{12} + c_{13}x_{13} + c_{14}x_{14} + c_{15}x_{15} +$$
$$c_{21}x_{21} + c_{22}x_{22} + c_{23}x_{23} + c_{24}x_{24} + c_{25}x_{25} +$$
$$c_{31}x_{31} + c_{32}x_{32} + c_{33}x_{33} + c_{34}x_{34} + c_{35}x_{35} +$$
$$c_{41}x_{41} + c_{42}x_{42} + c_{43}x_{43} + c_{44}x_{44} + c_{45}x_{45} +$$
$$c_{51}x_{51} + c_{52}x_{52} + c_{53}x_{53} + c_{54}x_{54} + c_{55}x_{55}$$

subject to the constraints

$$x_{11} + x_{12} + x_{13} + x_{14} + x_{15} = 1$$
$$\ldots$$
$$x_{51} + x_{52} + x_{53} + x_{54} + x_{55} = 1$$
$$x_{11} + x_{21} + x_{31} + x_{41} + x_{51} = 1$$
$$\ldots$$
$$x_{51} + x_{52} + x_{53} + x_{54} + x_{55} = 1$$
$$x_{11}, \ldots, x_{55} \ge 0$$

Interpretation: if $x_{ij} = 1$, then assign job j to machine i

Theorem. [Birkhoff 1946, von Neumann 1953] All extreme points of the above polyhedron are {0-1}-valued.

Corollary. Can solve assignment problem by solving LP since LP algorithms return an optimal solution that is an extreme point.

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?

Ex:

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

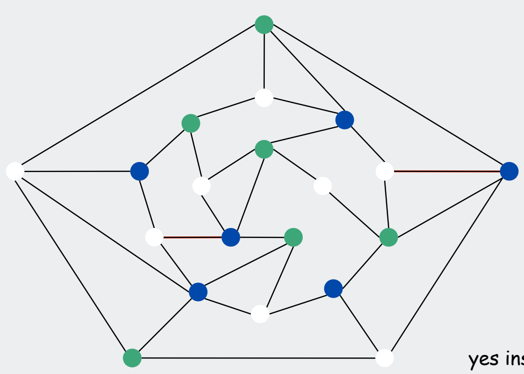
	x_1	x_2	x_3	x_4
solution	T	T	F	T

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

Key application. Electronic design automation (EDA).

Graph 3-Colorability

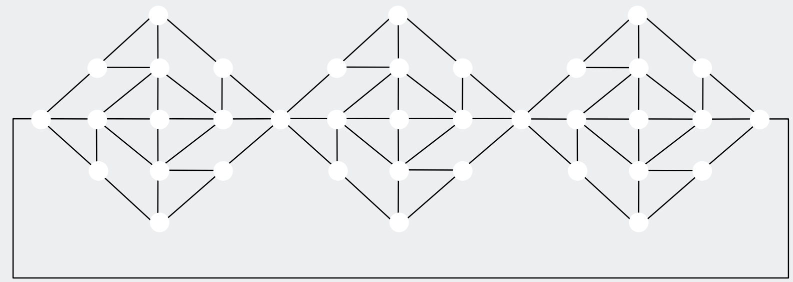
3-COLOR. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?



yes instance

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?



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 iff 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.

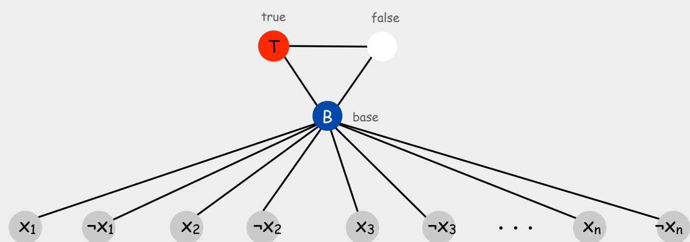
to be described next

Graph 3-Colorability

Claim. Graph is 3-colorable iff Φ is satisfiable.

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

- Consider assignment that sets all T literals to true.
- (ii) [triangle] ensures each literal is T or F.

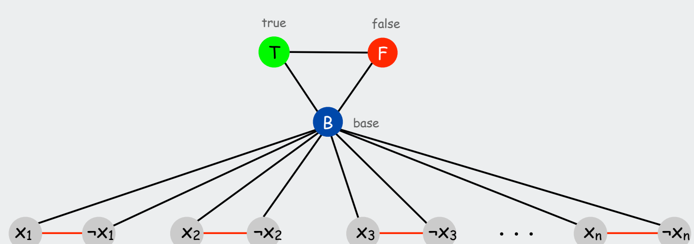


Graph 3-Colorability

Claim. Graph is 3-colorable iff Φ is satisfiable.

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

- Consider assignment that sets all T literals to true.
- (ii) ensures each literal is T or F.
- (iii) ensures a literal and its negation are opposites.

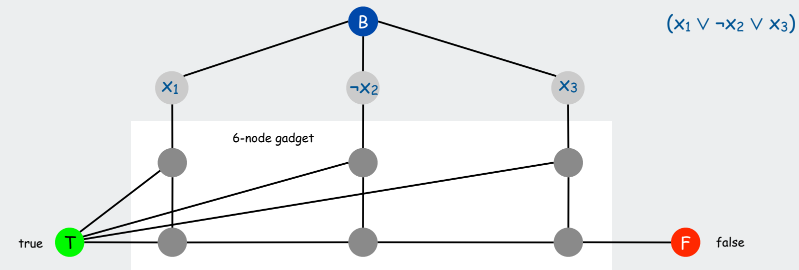


Graph 3-Colorability

Claim. Graph is 3-colorable iff Φ is satisfiable.

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

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



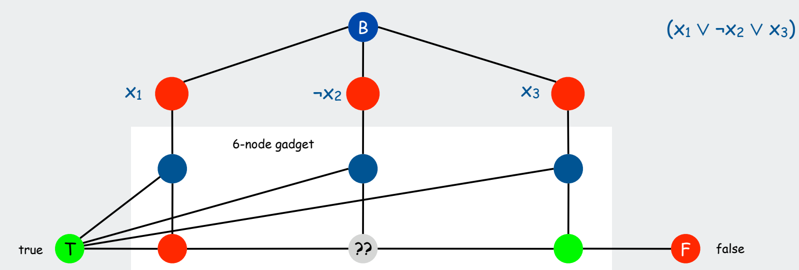
Graph 3-Colorability

Claim. Graph is 3-colorable iff Φ is satisfiable.

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

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

Therefore, Φ is satisfiable.



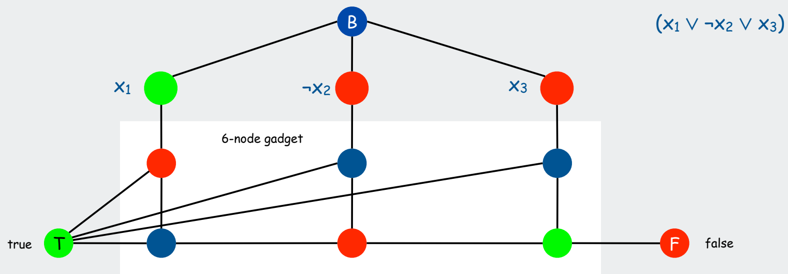
Graph 3-Colorability

Claim. Graph is 3-colorable iff Φ is satisfiable.

Pf. $\Leftarrow$ Suppose 3-SAT formula Φ is satisfiable.

- Color all true literals T and false literals F.
- Color vertex below one green vertex F, and vertex below that B.
- Color remaining middle row vertices B.
- Color remaining bottom vertices T or F as forced.

Therefore, graph is 3-colorable. ▪



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 iff 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

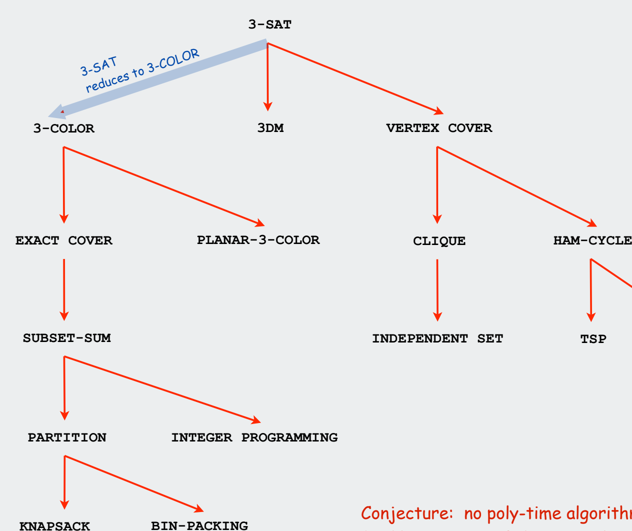
Conjecture: No polynomial-time algorithm for 3-SAT

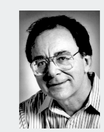
Implication: No polynomial-time algorithm for 3-COLOR.

Note: Construction is not intended for use, just for proof.

designing algorithms
proving limits
classifying problems
poly-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.

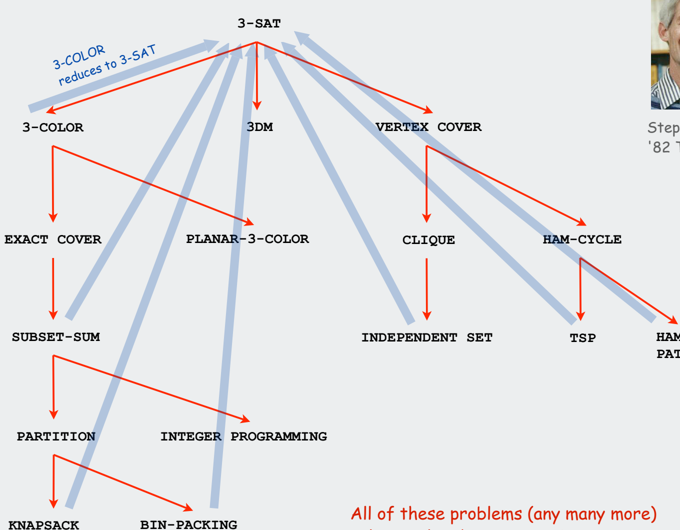
Pf sketch.

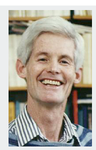
Each problem P in NP corresponds to a TM M that accepts or rejects any input in time polynomial in its size

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



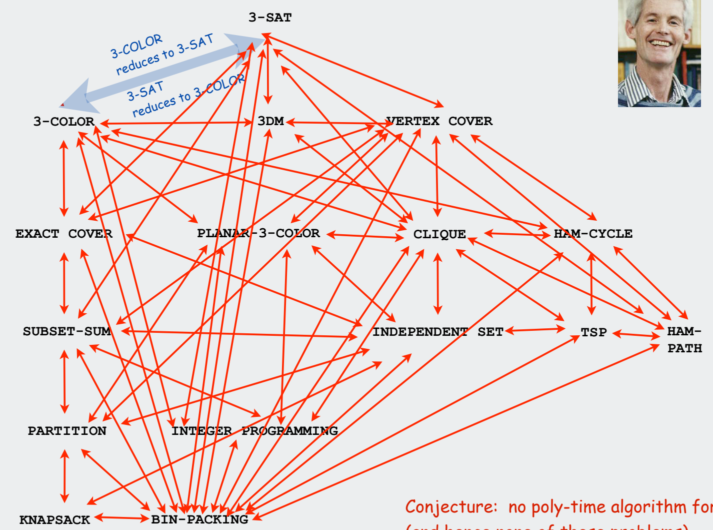


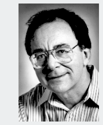
Stephen Cook
'82 Turing award

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

Implications of Karp + Cook

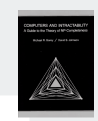
All of these problems poly-reduce to one another!

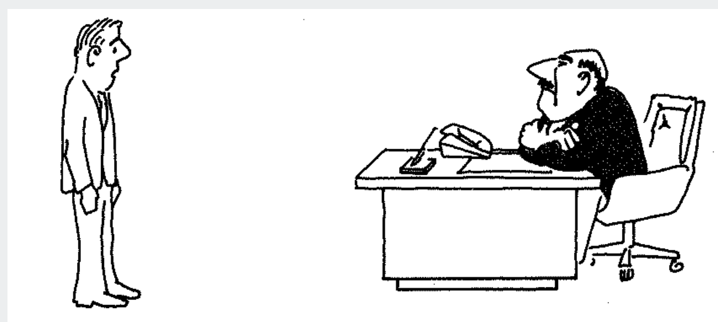




Conjecture: no poly-time algorithm for 3-SAT.
(and hence none of these problems)

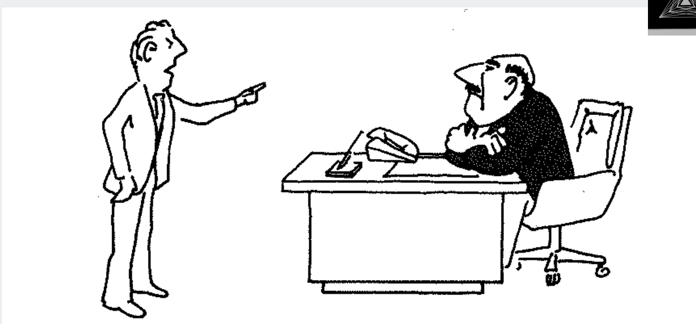
Poly-Time Reductions: Implications

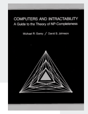




"I can't find an efficient algorithm, I guess I'm just too dumb."

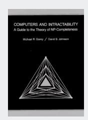
Poly-Time Reductions: Implications





"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