# Reductions



## Desiderata

Desiderata. Classify problems according to their computational requirements.

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

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



## Reduction

Def. Problem X reduces to problem Y if given a subroutine for Y, can solve X.

don't confuse with reduces from

- Cost of solving X = cost of solving Y + cost of reduction.
- Ex: X = closest pair, Y = Voronoi.

Consequences.

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



# Linear Time Reductions

## 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.
- Notation: $X \le_L Y$.

Some familiar examples.

- Dedup $\le_L$ sorting.
- Median $\le_L$ sorting.
- Convex hull $\le_L$ Voronoi.
- Closest pair $\le_L$ Voronoi.
- Arbitrage $\le_L$ negative cycle detection.
- Brewer's problem $\le_L$ linear programming.



## 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.

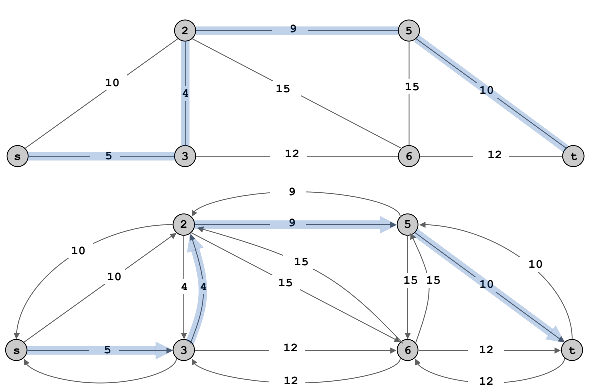
- Design algorithms: given algorithm for Y, can also solve X.
- Establish intractability: if X is hard, then so is Y.
- Classify problems: establish relative difficulty between two problems.



## Shortest Paths

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

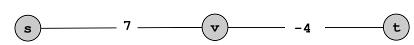
Pf. Replace each undirected edge by two directed edges.

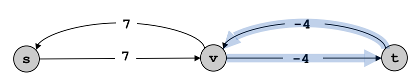





## Shortest Paths with Negative Weights

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.

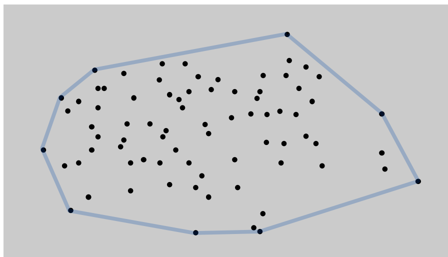
reduces to weighted non-bipartite matching (!)

## Convex Hull and Sorting

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

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

Claim. Convex hull linear reduces to sorting.
Pf. Graham scan algorithm.





## 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.

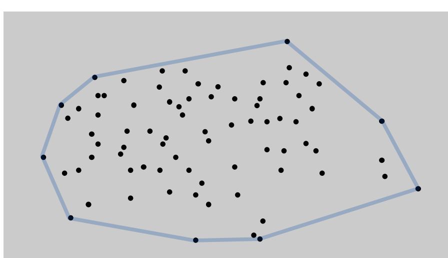


## Sorting and Convex Hull

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

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

Claim. Sorting linear reduces to convex hull.

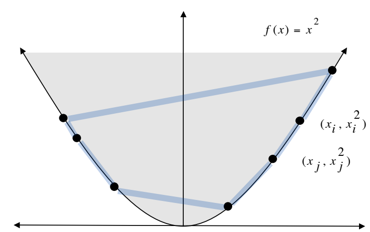




## Sorting Linear Reduces to Convex Hull

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.

## Sorting and Convex Hull: Lower Bound

Theorem. In quadratic decision tree model of computation, sorting N integers requires $\Omega(N \log N)$ steps.

allow tests of the form $x_i < x_j$ or $(x_j - x_i)(y_k - y_i) - (y_j - y_i)(x_k - x_i) < 0$

we just proved this

Claim. Sorting linear reduces to convex hull.

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



## 3-SUM Reduces to 3-COLLINEAR

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

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

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

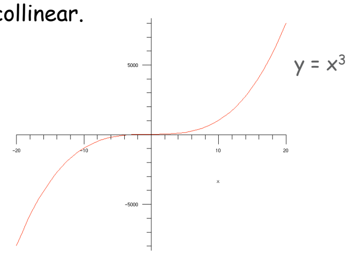
Pf.

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



## 3-SUM Reduces to 3-COLLINEAR

Claim. 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:

$$\begin{aligned}
\frac{a^3-b^3}{a-b} = \frac{b^3-c^3}{b-c} &\Leftrightarrow \frac{(a-b)(a^2+ab+b^2)}{a-b} = \frac{(b-c)(b^2+bc+c^2)}{b-c} \\
&\Leftrightarrow c^2 + bc - a^2 - ab = 0 \\
&\Leftrightarrow (c-a)(c+a+b) = 0 \\
&\Leftrightarrow c = a \quad \text{or} \quad a+b+c = 0
\end{aligned}$$



## 3-SUM and 3-COLLINEAR

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

we just proved this

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

Corollary. If no sub-quadratic algorithm for 3-SUM, then no sub-quadratic algorithm for 3-COLLINEAR.

## 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 decimal), is x prime?
COMPOSITE. Given an integer x, does x have a nontrivial factor?

Claim. PRIME $\leq_L$ COMPOSITE.

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



## Primality and Compositeness

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

Claim. COMPOSITE $\leq_L$ PRIME.

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



## Reduction Gone Wrong

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

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

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

# Polynomial-Time Reductions



## 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, plus
- One call to subroutine for Y.

Notation. $X \leq_P Y$.

Ex. Assignment problem $\leq_P$ LP.
Ex. 3-SAT $\leq_P$ 3-COLOR.



## Poly-Time Reductions

Goal. Classify and separate problems according to relative difficulty.

- Those that can be solved in polynomial time.
- Those that (probably) require exponential time.

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

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

Useful property. If $X \leq_P Y$ and $Y \leq_P Z$ then $X \leq_P Z$.



## Assignment Problem

Assignment problem. Assign n jobs to n machines to minimize total cost, where $c_{ij}$ = cost of assignment 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 PU students to writing seminars.

## Assignment Problem Reduces to Linear Programming

LP formulation. $x_{ij}$ = 1 if job j assigned to machine i.

$$
\begin{array}{lrcll}
\min & \sum_{1 \le i \le n} \sum_{1 \le j \le n} c_{ij}\, x_{ij} & & & \\
\text{s. t.} & \sum_{1 \le j \le n} x_{ij} & = & 1 & 1 \le i \le n \\
 & \sum_{1 \le i \le n} x_{ij} & = & 1 & 1 \le j \le n \\
 & x_{ij} & \ge & 0 & 1 \le i, j \le n
\end{array}
$$

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

Corollary. Assignment problem reduces to LP; can solve in poly-time.

we assume LP returns an extreme point solution



## 3-Satisfiability

Literal: A Boolean variable or its negation. $x_i$ or $\overline{x_i}$

Clause. A disjunction of 3 distinct literals. $C_j = x_1 \lor \overline{x_2} \lor x_3$

Conjunctive normal form. A propositional formula $\Phi$ that is the conjunction of clauses. $\Phi = C_1 \land C_2 \land C_3 \land C_4$

SAT. Given CNF formula $\Phi$, does it have a satisfying truth assignment?

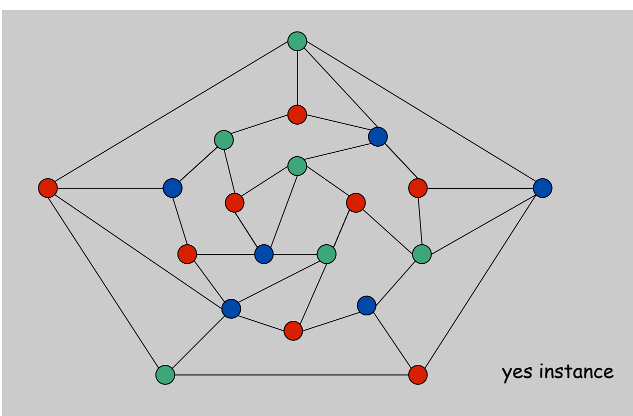
Ex. $\left(\overline{x_1} \lor x_2 \lor x_3\right) \land \left(x_1 \lor \overline{x_2} \lor x_3\right) \land \left(\overline{x_1} \lor \overline{x_2} \lor \overline{x_3}\right)$

Yes. $x_1 = \text{true},\ x_2 = \text{true},\ x_3 = \text{false}$



## Graph 3-Colorability

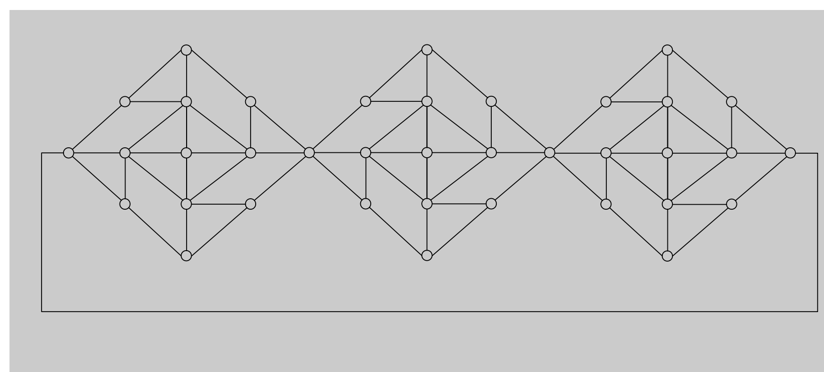
3-COLOR. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?






## Graph 3-Colorability

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 $\Phi$, we construct an instance of 3-COLOR that is 3-colorable iff $\Phi$ 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

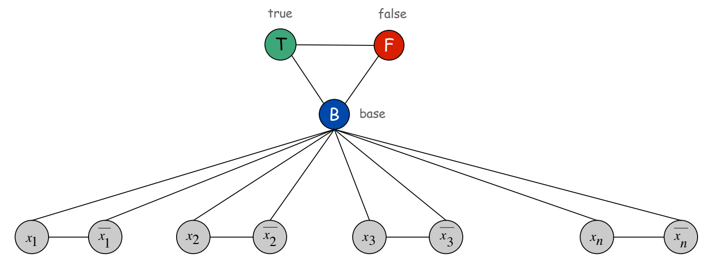


## Graph 3-Colorability

Claim. Graph is 3-colorable iff $\Phi$ 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.



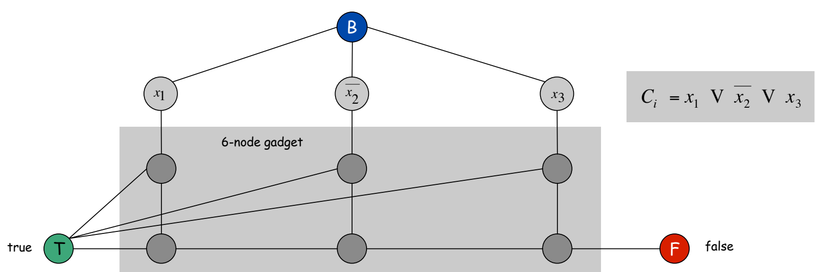



## Graph 3-Colorability

Claim. Graph is 3-colorable iff $\Phi$ 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) ensures at least one literal in each clause is T.



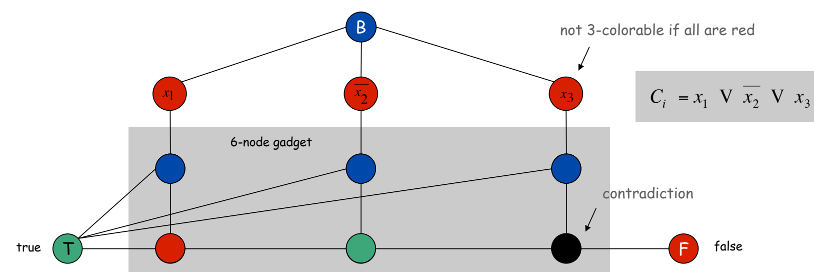



## Graph 3-Colorability

Claim. Graph is 3-colorable iff $\Phi$ 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) ensures at least one literal in each clause is T.

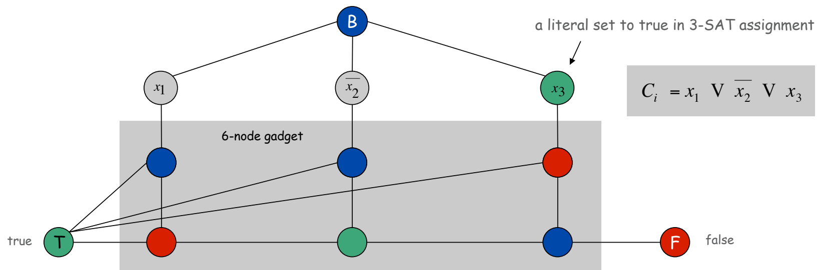


## Graph 3-Colorability

Claim. Graph is 3-colorable iff $\Phi$ is satisfiable.

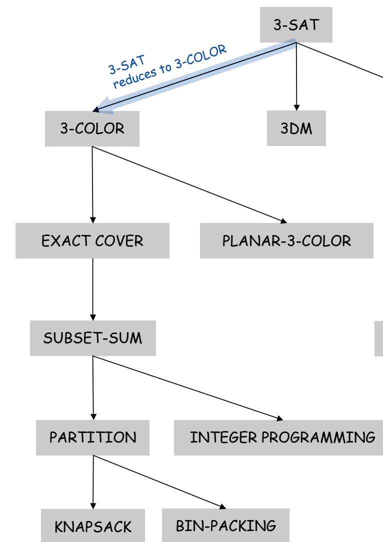
Pf. $\Leftarrow$ Suppose 3-SAT formula $\Phi$ is satisfiable.

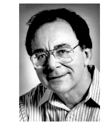
- Color all true literals T.
- Color node below green node F, and node below that B.
- Color remaining middle row nodes B.
- Color remaining bottom nodes T or F as forced. ▪






## More Poly-Time Reductions



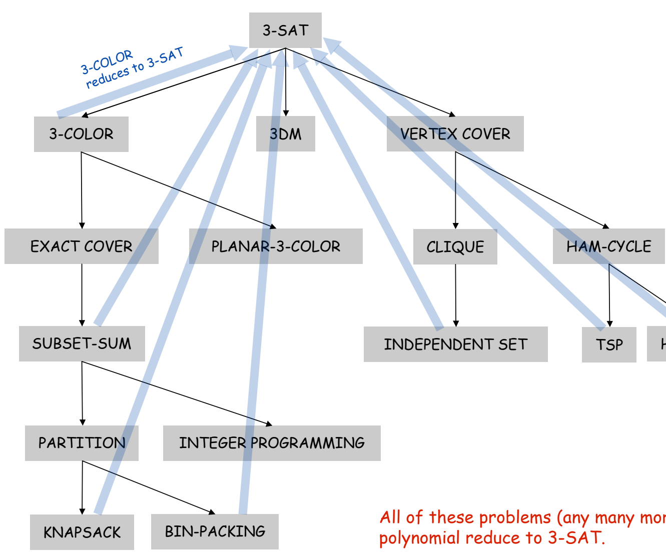



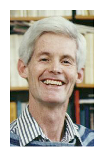
Dick Karp
Turing award (1985)

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



## Cook's Theorem



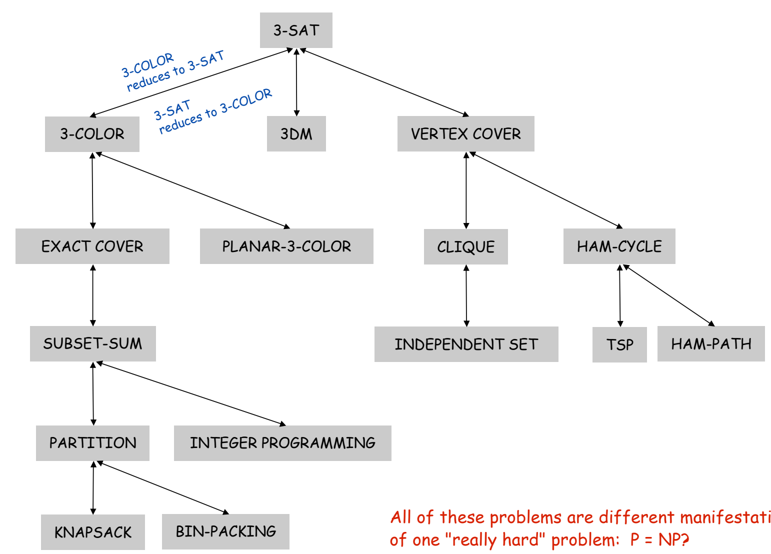



Stephen Cook
Turing award (1982)

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



## Cook + Karp




All of these problems are different manifestations of one "really hard" problem: P = NP?

## Summary

Reductions are important in theory to:

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

Reductions are important in practice to:

- Design algorithms.
- Design reusable software modules.
  - stack, queue, sorting, priority queue, symbol table
    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 NP-hard problems

e.g., bin packing