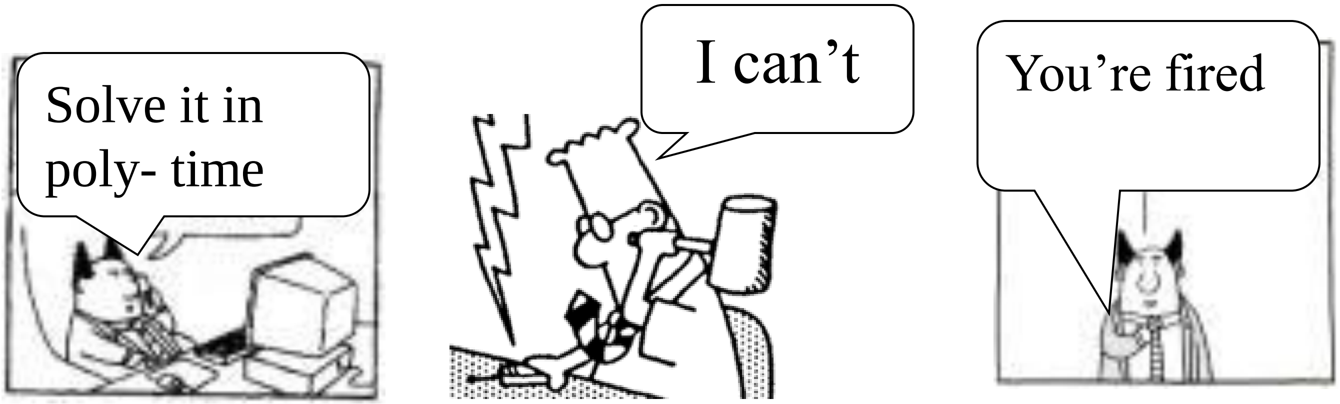
Advanced Algorithms

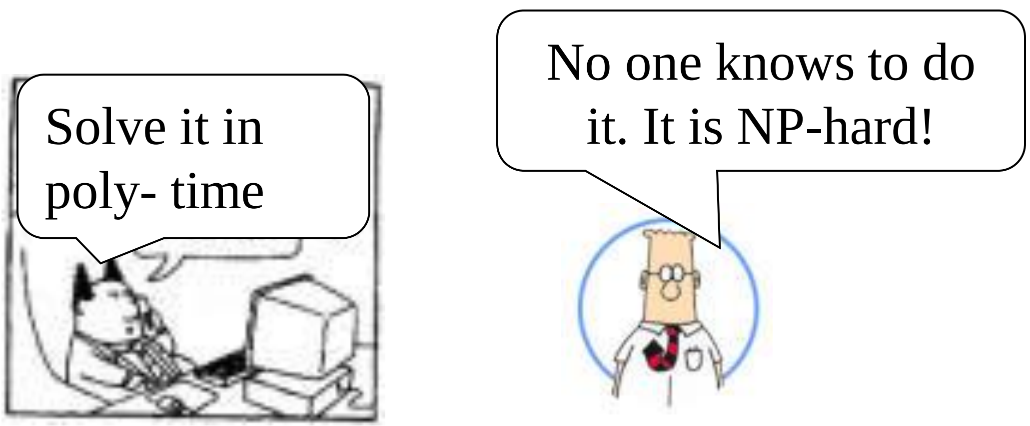
What is Efficient????
NP-Hardness,
Coping with NP-hardness.

NP-Completeness Theory

I.



II.



NP-Completeness Theory

- Explains why some problems are hard and probably not solvable in polynomial time.
- Invented in the early 1970s (Karp, Cook, Levin).
- Talks about the problems, independent of the implementation, the machine, or the algorithm.

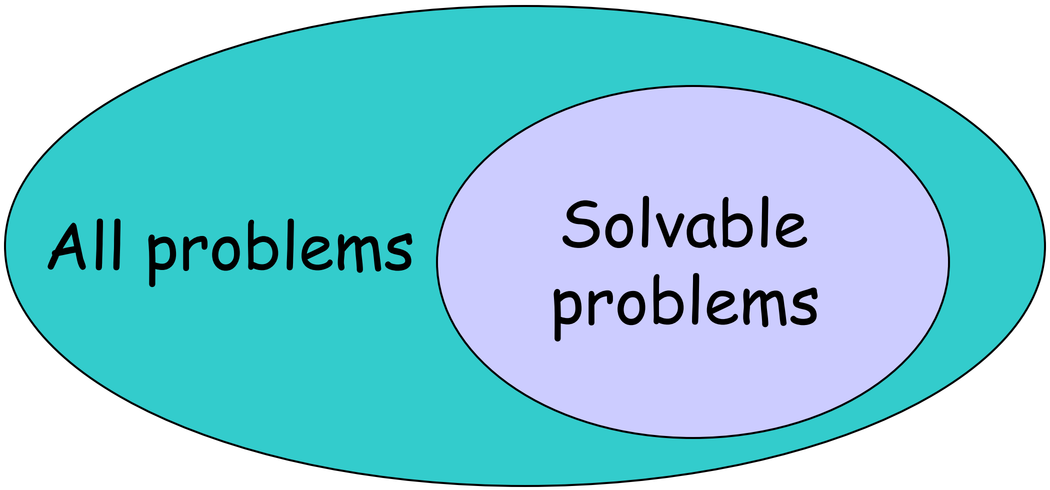
Polynomial-Time Algorithms

- Some problems are intractable: as they grow large, we are unable to solve them in reasonable time.
- What constitutes reasonable time? Standard working definition: polynomial time
  - On an input of size n the worst-case running time is $O(n^k)$ for some constant k
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$
  - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$, $O(n^{\log\log n})$

Polynomial-Time Algorithms

- We define **P** to be the class of problems solvable in polynomial time.
- Are all problems solvable in polynomial time?
  - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
  - Such problems are clearly intractable, not in **P**

So some problems cannot be solved at all



We will explore the 'solvable area', and will distinguish between problems that can be solved efficiently and those that cannot be solved efficiently.

NP-Complete Problems

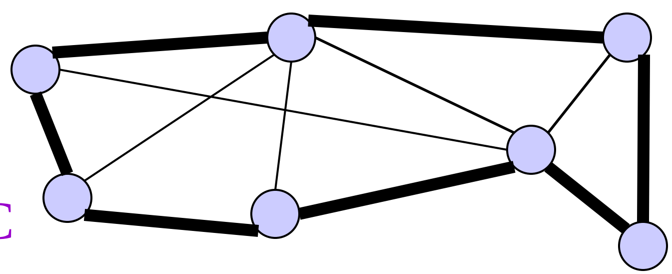
- The *NP-Complete* problems are an interesting class of solvable problems whose status is unknown
  - No polynomial-time algorithm has been discovered for any NP-Complete problem.
  - No super-polynomial lower bound has been proved for any NP-Complete problem, either.
- We call this the *$P = NP$ question*
  - The biggest open problem in CS.

An NP-Complete Problem: Hamiltonian Cycle

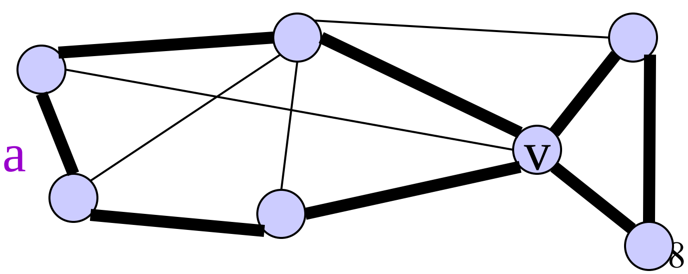
- An example of an NP-Complete problem:
  - A *hamiltonian cycle* in an undirected graph is a simple cycle that visits every vertex.
  - The hamiltonian-cycle problem: given a graph G, does it have a hamiltonian cycle?
  - A naïve algorithm for solving the hamiltonian-cycle problem: check all paths.
  - Running time? Exponential in size of G.

✓

a HC



Not a HC



P and NP

- **P** = problems that can be solved in polynomial time
- **NP** = problems for which a solution can be verified in polynomial time = problems that can be solved in polynomial time by a non-deterministic machine.
- Unknown whether **P = NP** (most suspect not)

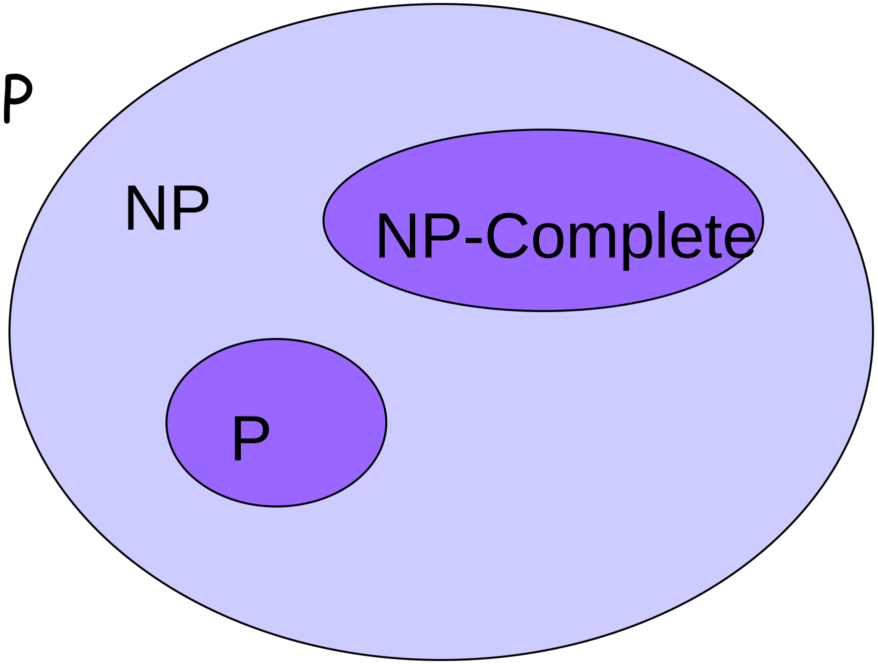
- Hamiltonian-cycle problem is in **NP**:
  - Don't know how to solve in polynomial time.
  - Easy to verify solution in polynomial time.

NP-Complete Problems

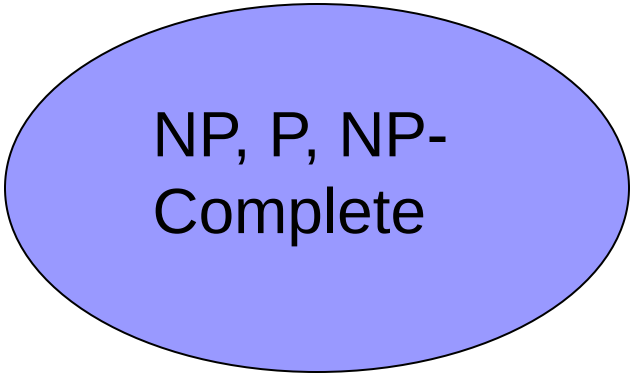
- NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time...
  - ...then *every* problem in NP can be solved in polynomial time (which would show **P = NP**)
  - Thus: solve hamiltonian-cycle in $O(n^{100})$ time, you've proved that **P = NP**. Retire rich & famous.

NP Problems

For sure P $\subseteq$NP



But maybe P=NP ??



Why Prove NP-completeness?

- Though nobody has proven that **P != NP**, if you prove a problem is NP-Complete, most people accept that it is probably intractable.
- Therefore it can be important to prove that a problem is NP-Complete
  - Don't bother coming up with an efficient algorithm.
  - Can instead work on *approximation algorithms*.
  - Or try other ways to circumvent the problem

NP-Hard and NP-Complete Problems

- important concept - reduction
- P is *polynomial-time reducible* to Q ($P \leq_p Q$) if given a black box that solves Q in polynomial time, it is possible solve P in polynomial time.
- P is NP-complete if:
  - $P \in NP$ and
  - Every problem R in NP is reducible to P
    $R \leq_p P, \forall R \in NP$

  NP-Hard

- Exercise: prove:
  If $P \leq_p Q$ and P is NP-hard then Q is NP-hard.

Using Reductions

- Given one NP-Complete problem, we can prove that many interesting problems are NP-Complete. This includes:
  - Graph coloring
  - Hamiltonian path/cycle
  - Knapsack problem
  - Traveling salesman
  - Job scheduling
  - Many, many, many more

Graph Coloring

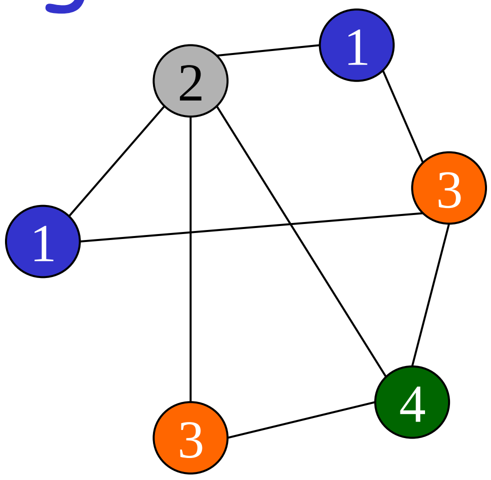
A problem that has lots of applications:

- Resource Allocation
- VLSI design
- Parallel computing

Definition: A coloring of a graph $G(V,E)$ is a function $c:V \rightarrow N$ such that for any edge $(u,v) \in E$, $c(v) \neq c(u)$

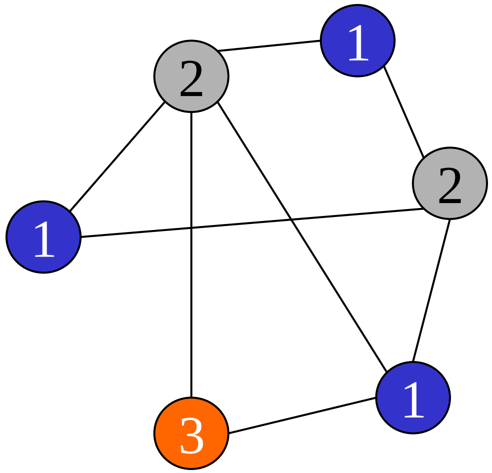
Graph Coloring

Example: coloring with 4 colors.



Problem: Given a graph G, color G using the minimal number of colors.

Example: same graph, 3 colors.



Definition: The chromatic number of a graph (denoted $\chi(G)$) is the minimal number of colors needed to color G.

Optimization v.s. Decision

To simplify things, we will worry only about *decision problems* with a yes/no answer

- Many problems are *optimization/search problems*, but we can often re-cast them as decision problems

Example: Graph coloring.

- Optimization problem: what is the minimal number of colors needed to color *G*?
- Search problem: Can *G* be colored using k colors? If so, find a legal k-coloring.
- Decision problem: Can *G* be colored using k colors?

Proving NP-Completeness

- How do we prove a problem P is NP-Complete?
  - Pick a known NP-Complete problem A
  - Reduce A to B (show $A \leq_p B$, use B to solve A)
    - Describe a transformation that maps instances of A to instances of B, s.t. "yes" for $A \Leftrightarrow$ "yes" for B
    - Prove the transformation works
    - Prove it runs in polynomial time
  - and yeah, prove $B \in NP$
- We need at least one problem for which NP-hardness is known. Once we have one, we can start reducing it to many problem.

The SAT Problem

- The first problems to be proved NP-Complete was *satisfiability* (SAT):
  - Given a Boolean expression on *n* variables, can we assign values such that the expression is TRUE?
  - Ex: $((x_1 \wedge x_2) \vee \neg((\neg x_1 \wedge x_3) \vee x_4)) \wedge \neg x_2$
  - **The Cook-Levin Theorem:** SAT is NP-Complete
    - Note: Argue from first principles, not reduction
    - Proof: not here
      (any computation can be described using SAT expressions)

Conjunctive Normal Form

- Even if the form of the Boolean expression is simplified, the problem may be NP-Complete
  - *Literal*: an occurrence of a Boolean or its negation
  - A Boolean formula is in *conjunctive normal form*, or *CNF*, if it is an AND of clauses, each of which is an OR of literals
    - Ex: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5)$

3-CNF: each clause has exactly 3 distinct literals

- Ex: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5 \vee x_3 \vee x_4)$
- Note: true if at least one literal in each clause is true

The 3-CNF Problem

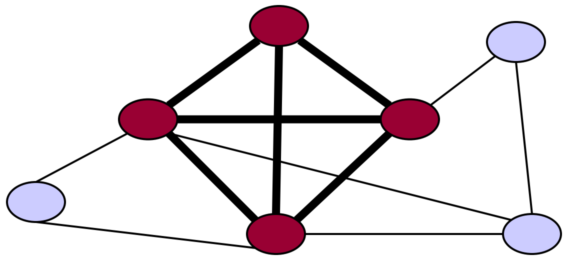
- Theorem: Satisfiability of Boolean formulas in 3-CNF form (the *3-CNF Problem*) is NP-Complete
  - Proof: not here
- The reason we care about the 3-CNF problem is that it is relatively easy to reduce to others.
  - Thus, knowing that 3-CNF is NP-Complete we can prove many seemingly unrelated problems are NP-Complete.
- Remark: 2-CNF is in P

The k-clique Problem

- *A clique in a graph G is* a subset of vertices fully connected to each other, i.e. a complete subgraph of G.
- The *clique problem*: how large is the maximum-size clique in a graph?
- *Can we turn this into a decision problem?*
- A: Yes, we call this the *k-clique problem*
- *Is the k-clique problem within **NP**?*

  Yes: Given a set of vertices, it is easy to verify that it is a clique of size k.

4-clique:



3-CNF $\leq_p$ Clique

- How can we prove that k-clique is NP-hard?
- We need to show that if we can solve k-clique then we can solve a problem which is known to be NP-hard.
- We will do it for 3-CNF:
- Given a 3-CNF formula, we will transform it to an instance of k-clique (a graph and a number k), for which a k-clique exists iff the 3-CNF formula is satisfiable.

3-CNF $\leq_p$ Clique

- The reduction:
  - Let $F = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a 3-CNF formula with k clauses, each of which has 3 distinct literals.
  - For each clause, put three vertices in the graph, one for each literal.
  - Put an edge between two vertices if they are in different triples and their literals are consistent (i.e., not each other's negation).

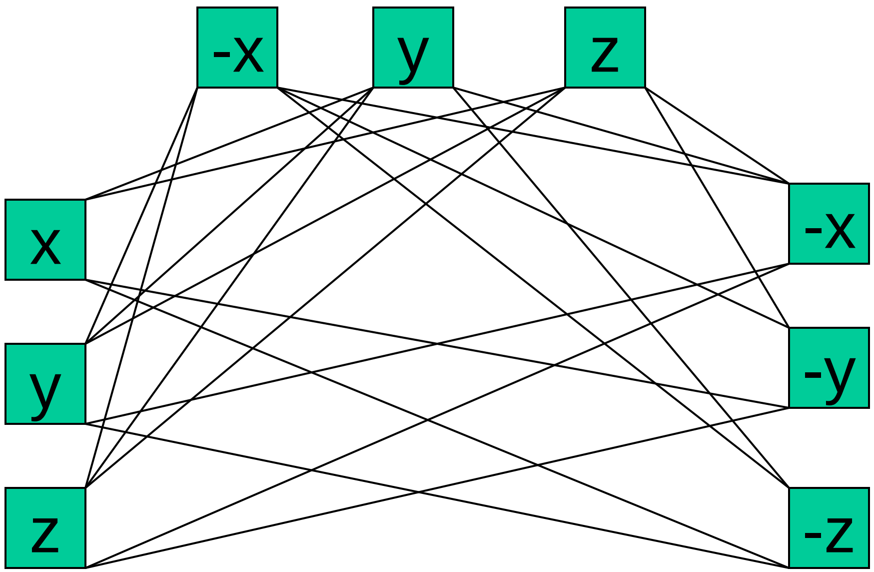
Construction by Example

$$F = (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge \underline{(\neg x \vee \neg y \vee \neg z)}$$

literal

clause

G



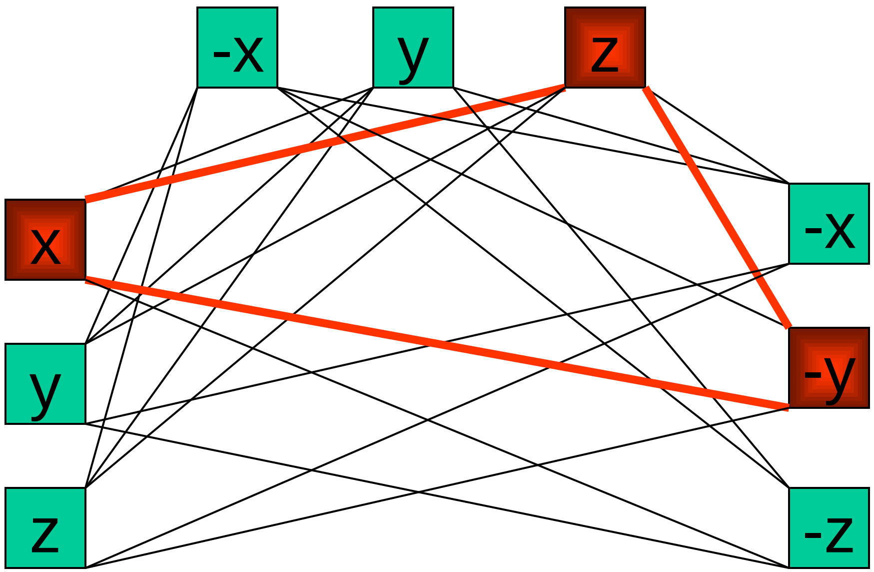
An edge means 'these two literals do not contradict each other'.

Construction by Example

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

$$x = 1, y = 0, z = 1$$

G



Any clique of size k must include exactly one literal from each clause.

General Construction

$$F = \bigcap_{i=1}^{k} \bigcup_{j=1}^{3} a_{ij} \quad \text{where } a_{ij} \in \{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$$

literals

$$G = (V, E) \quad \text{where}$$

$$V = \{a_{ij} : 1 \leq i \leq k, 1 \leq j \leq 3\}$$

$$E = \{\{a_{ij}, a_{i',j'}\} : i \neq i' \text{ and } a_{ij} \neq \neg a_{i'j'}\}$$

k is the number of clauses

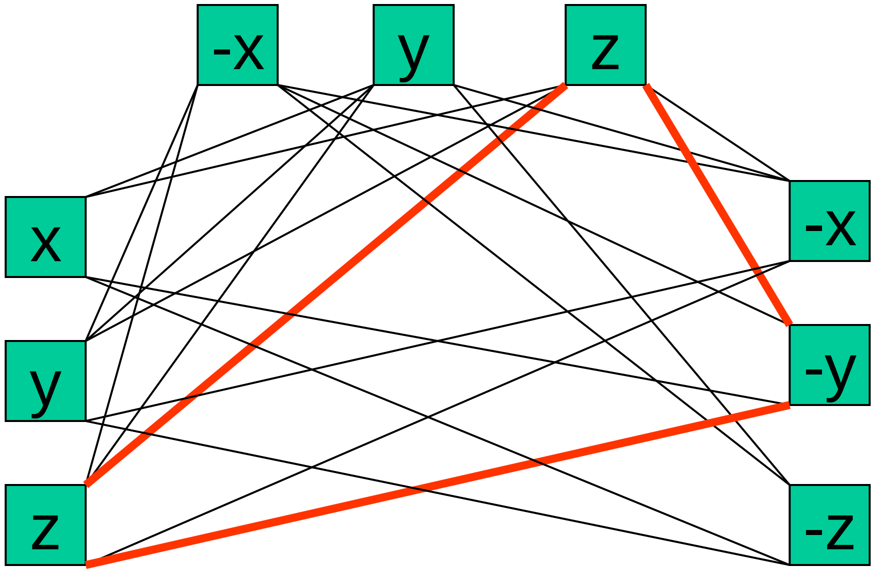
The Reduction Argument

- We need to show
  - F satisfiable implies G has a clique of size k.
    - Given a satisfying assignment for F, for each clause pick a literal that is satisfied. Those literals in the graph G form a k-clique.
  - G has a clique of size k implies F is satisfiable.
    - Given a k-clique in G, assign TRUE to each literal in the clique. This yields a satisfying assignment to F (why?).

Clique to Assignment

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

G

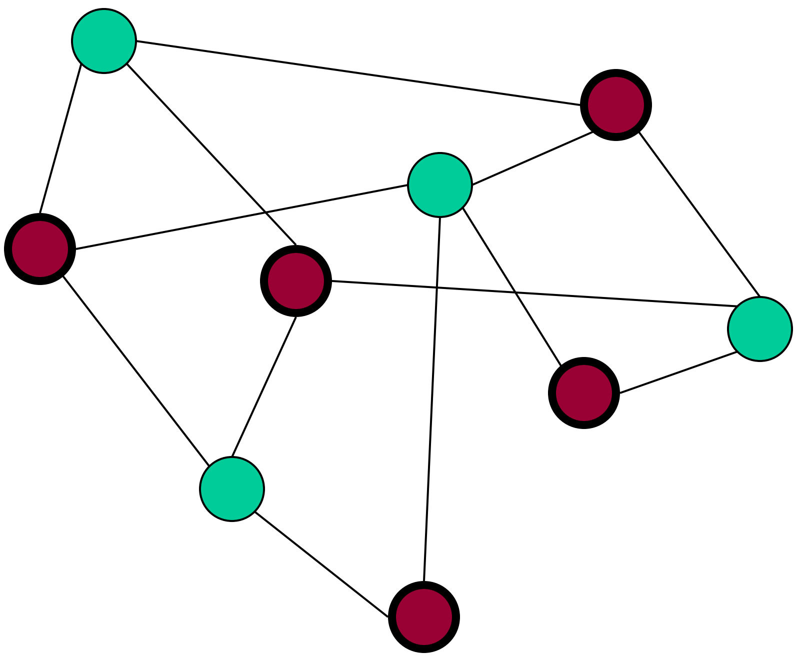


$$y = 0, z = 1$$

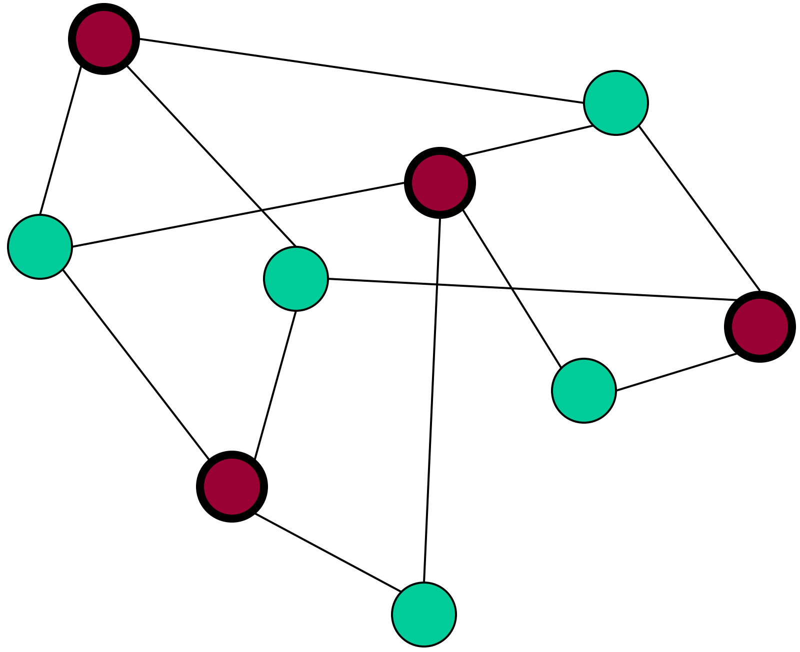
The Vertex Cover Problem

- A vertex cover for a graph G is a set of vertices incident to every edge in G
- The vertex cover problem: what is the minimum size vertex cover in G?
- Restated as a decision problem: does a vertex cover of size k exist in G?
- Theorem: vertex cover is NP-Complete

Vertex Cover (Example)



A vertex cover of size 5

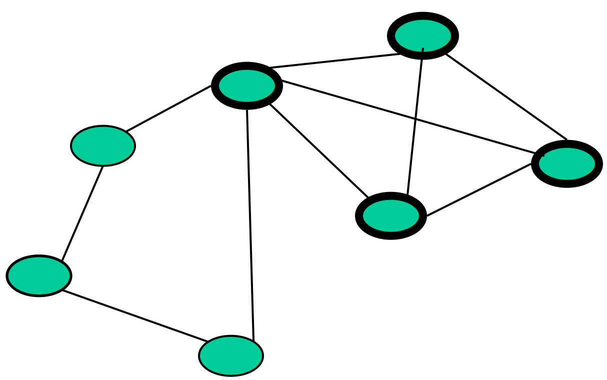


A vertex cover of size 4

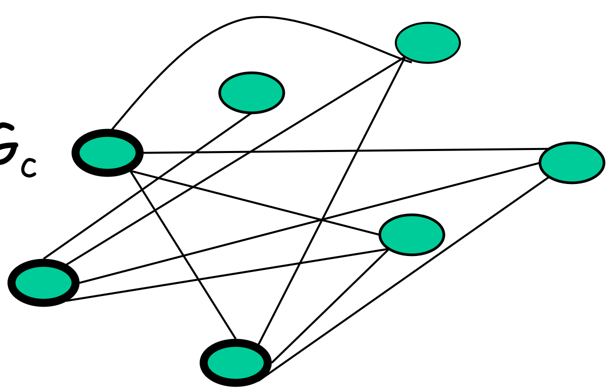
Vertex Cover is NP-Complete

- First, show vertex cover in NP (How?)
- Next, reduce k-clique to vertex cover:
  - The complement G_C of a graph G contains exactly those edges not in G
  - Given (G,k), in input for the clique problem
  - Compute G_C in polynomial time

G



G_c

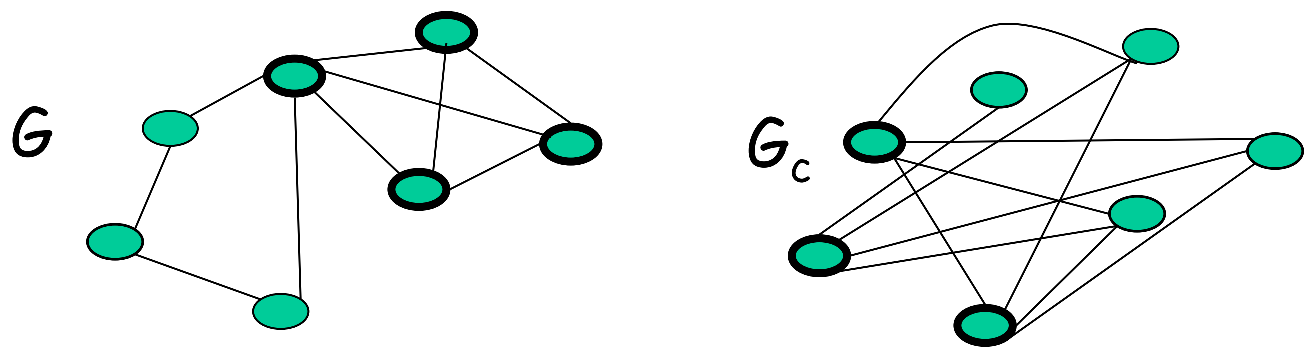


Clique $\leq_p$ Vertex Cover

Claim 1: If G has a clique of size k, then G_C has a vertex cover of size $|V| - k$

Claim 2: If G_C has a vertex cover of size $|V|-k$, then G has a clique of size k

Proofs: easy (Complexity course)



The Traveling Salesman Problem:

- A well-known optimization problem:
  - Optimization variant: a salesman must travel to n cities, visiting each city exactly once and finishing where he begins. How to minimize travel time?
  - Model as complete graph with cost c(i,j) to go from city i to city j
- How would we turn this into a decision problem?
  - Answer: ask if there exists a path with cost < k

The Traveling Salesman Problem:

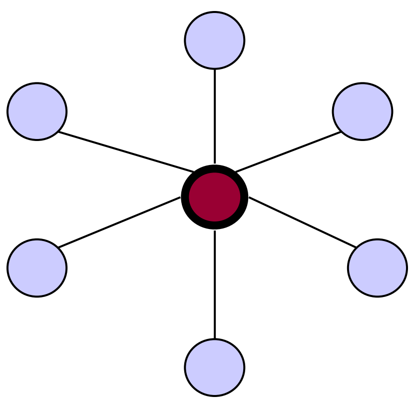
- Asides:
  - TSPs (and variants) have enormous practical importance
    - E.g., for shipping and freighting companies
    - Lots of research into good approximation algorithms
  - Made famous as a DNA computing problem
    - Adleman used DNA to solve a 7-city instance [1994]

Other NP-Complete Problems

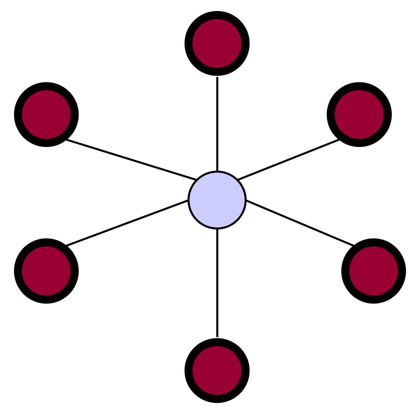
- *Partition:* Given a set of integers, whose total sum is 2S, can we partition them into two sets, each adds up to S?
- *Subset-sum*: Given a set of integers, does there exist a subset that adds up to some desired target T?

Independent Set

- Input: A graph $G=(V,E)$, k
- Problem: Is there a subset S of V of size at least k such that no pair of vertices in S has an edge between them.
- Maximum independent set problem: find a maximum size independent set of vertices.



Maxim**al** independent set



Maxim**um** independent set

Steiner Tree

- Input: A graph $G=(V,E)$, a subset T of the vertices V, and a bound B
- Problem: Is there a tree connecting all the vertices of T of total weight at most B?

- Application: Network design and wiring layout.
- The case $T=V$ is polynomially solvable (this is the MST problem).

Exact Cover

- Input: A set $U = \{u_1, u_2, \ldots u_n\}$ and subsets

$$S_1, S_2, \ldots, S_m \subseteq U$$

- Output: Determine if there is a set of disjoint sets that union to U, that is, a set X such that:

$$X \subseteq \{1, 2, \ldots, m\}$$

$$i, j \in X \text{ and } i \neq j \text{ implies } S_i \cap S_j = \emptyset$$

$$\bigcup_{i \in X} S_i = U$$

Example of Exact Cover

$$U = \{a, b, c, d, e, f, g, h, i\}$$

$$\{a, c, e\}, \{a, f, g\}, \{b, d\}, \{b, f, h\}, \{e, h, i\}, \{f, h, i\}, \{d, g, i\}$$

Exact Cover:

$$\{a, c, e\}, \{b, f, h\}, \{d, g, i\}$$

Bin Packing

- Input: A set of numbers $A = \{a_1, a_2, \ldots, a_m\}$ and numbers B (capacity) and K (number of bins).
- Output: Determine if A can be partitioned into $S_1, S_2, \ldots, S_K$ such that for all i

$$\sum_{j \in S_i} a_j \leq B.$$

Bin Packing Example

- A = {2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5}
- B = 10, K = 4
- Bin Packing:
  - 3, 3, 4
  - 2, 3, 5
  - 5, 5
  - 2, 4, 4

Perfect fit!

Coping with NP-hardness

- O.K, I know that a problem is NP-hard.
  What should I do next?
- First, stop looking for an efficient algorithm.
- Next, you might insist on finding an optimal solution (knowing that this might take a lot of time), or you can look for solutions that are satisfactory but not optimal.

Techniques for Dealing with NP-complete Problems

- Exactly
  - backtracking, branch and bound, dynamic programming.
- Approximately
  - approximation algorithms with performance guarantees.
  - heuristics with good average results.
- Change the problem (impose more structure on instances / solutions)

Advanced Algorithms

Approximation Algorithms

Approximation Algorithms

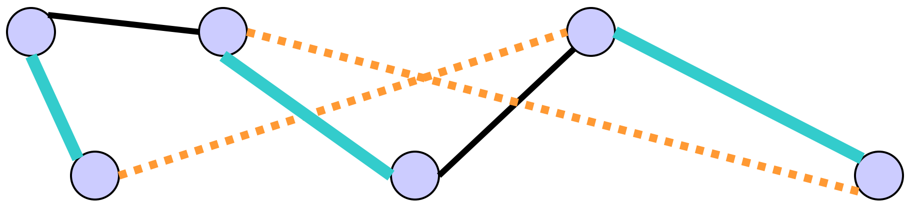
- The fact that a problem is NP-complete doesn't mean that we cannot find an approximate solution efficiently.
- We would like to have some guarantee on the performance – how far are we from the optimal?
- What is the best we can hope for (assuming $P \neq NP$)?

Approximation Algorithms with Additive Error.

- For few NP-hard problems, there are approximation algorithms that produce an **almost optimal** solution – one that is far only by an additive constant from the optimal.
- Minimization problems: $Alg(I) \leq opt(I) + c$
- Maximization problems: $Alg(I) \geq opt(I) - c$
- Example: Edge coloring.

Edge Coloring

- An Edge-coloring of a graph $G=(V,E)$ is an assignment, c, of integers to the edges such that if e_1 and e_2 share an endpoint then $c(e_1) \neq c(e_2)$.



- Let Δ be the maximal degree of some vertex in G.
- It is known that for any graph the minimal number of colors required to edge-color G is Δ or $\Delta+1$.
- It is NP-hard to distinguish between these two cases.
- There exists a poly-time algorithm that colors any graph G with at most $\Delta+1$ colors.
- For this algorithm $Alg(I) \leq OPT(I) + 1$.

r-approximation Algorithms

- Approximations with guaranteed additive error are rare.
- All approximation algs we are going to see are factor-r approximations:
  - Vertex cover
  - Traveling salesman
  - Bin packing
  - Knapsack
- An algorithm *Alg* is an r-approximation if, for any input, the solution that *Alg* outputs is within factor r from the optimal. ($r \geq 1$)

Approximation Algorithms (minimization)

- In minimization problems: *Alg* is r-approximation if $Alg(I) \leq r \cdot OPT(I)$ for any instance I.

Example 1: Traveling Salesman is a minimization problem (the goal is to find a tour with minimal cost). If we have an algorithm, *A*, that finds, *for any graph*, a tour whose cost is at most 5 times the optimal, then *A* is 5-approximation to TSP.

Example 2: Minimum Spanning Tree is a minimization problem (the goal is to find an ST with minimal cost). The optimal algorithms we know are 1-approximate.

Approximation Algorithms (maximization)

- In maximization problems: *Alg* is r-approximation if $Alg(I) \geq (1/r) \cdot opt(I)$ for any instance I.

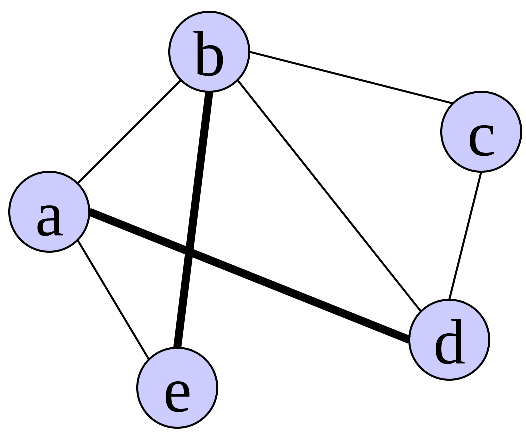
Example: Maximum clique is a maximization problem (the goal is to find a clique with maximum size). If we have an algorithm, *A*, that finds, for any graph, a clique whose size is at least $(\log n)^2/n$ times the optimal, then *A* is $n/(\log n)^2$-approximation to clique.

(remark: best known ratio for clique is $n (\log \log n)^2/(\log n)^3$)

Reminder: Matching (to be used soon)

- Definition: a matching in a graph G is a subset M of E such that the degree of each vertex in $G'=(V',M)$ is 0 or 1.
- Example: $M=\{(a,d),(b,e)\}$ is a matching.

  $S=\{(a,d), (c,d)\}$ is not a matching.



Example 1: Vertex Cover

- Given $G=(V,E)$, find a minimum sized subset W of V such that for every (v,u) in E, at least one of v or u is in W.
- Vertex Cover is NP-Hard.
- We are willing to end up with a vertex cover W which is not of minimum size. But, we don't want it to be too large and we want to be able to find it in polynomial time.

Approximating Vertex Cover

VertexCover($G=(V,E)$):

while ($E \neq \emptyset$)

1. select an arbitrary edge (u,v)
2. add both u and v to the cover
3. delete all edges incident to either u or v

1. This is a legal cover (why?)
2. This is a 2-approximation; its size is at most 2 times OPT (the size of a minimum vertex cover).

Proof: Let c be the number of iterations. The VC has size $2c$. The edges selected in step 1 form a matching of size c (why?). Even if we only need to cover these edges we need at least c vertices.

Approximating Vertex Cover

A more natural algorithm: select in each iteration a vertex with maximum degree, add it to the cover and remove all its adjacent edges.

Looks promising!

However, the approximation ratio of this approach is not bounded: for any r there exists a graph for which the VC chosen by the algorithm is r-times larger than the optimal VC

Proof: In Class