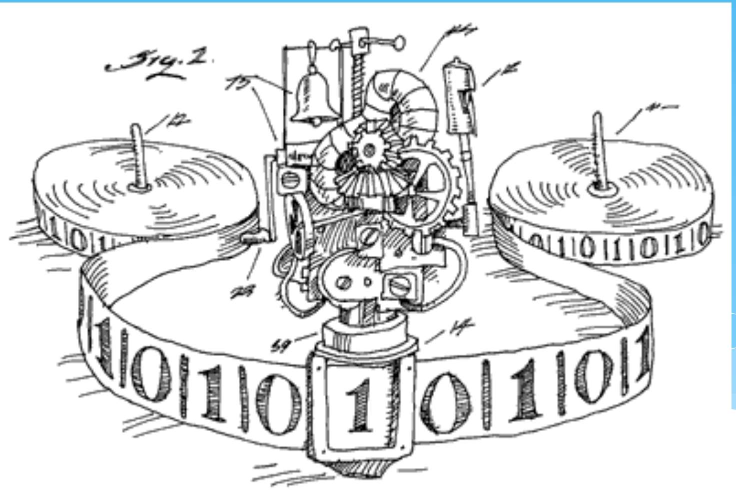
EECS 376: Foundations of Computer Science

Chris Peikert
20 March 2023



Agenda

* Recap poly-time mapping reductions
* Search-to-Decision Reductions
* Dealing with NP-Completeness
  * Approximation algorithms

Recall

* **Definition:** A language B is $\mathbf{NP}$-***Complete*** if:
  1. $B \in \mathbf{NP}$
  2. B is $\mathbf{NP}$-Hard: $A \leq_p B$ for *every* language $A \in \mathbf{NP}$.
     Equivalently: $A \leq_p B$ for *some NP-hard* language A.
* **Definition:** Language A is ***polynomial-time mapping reducible*** to language B, written $A \leq_p B$, if there is a polynomial-time algorithm f such that:
  * $$x \in A \iff f(x) \in B.$$
  * Implies: If $B \in \mathbf{P}$, then $A \in \mathbf{P}$. If $A \notin \mathbf{P}$, then $B \notin \mathbf{P}$.

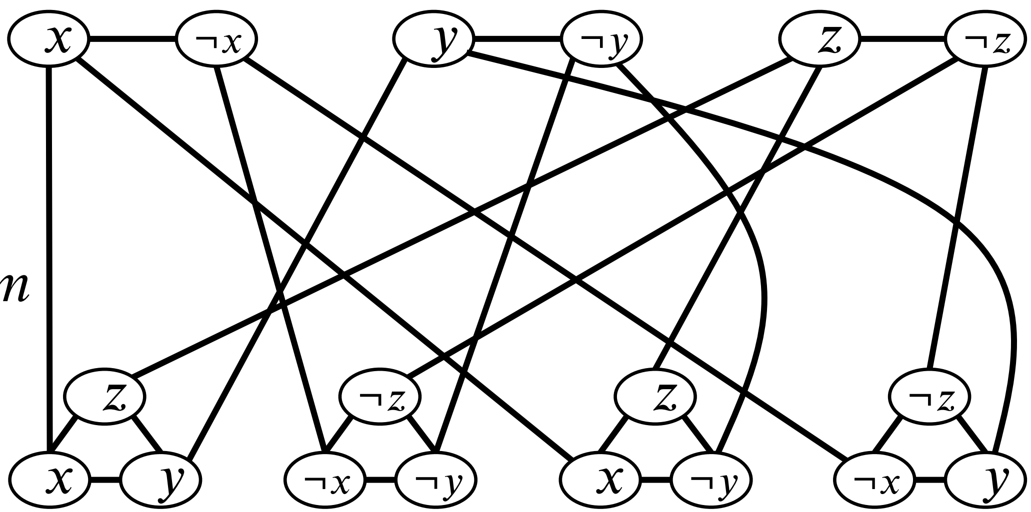
3SAT $\leq_p$ VERTEXCOVER

- **Goal:** efficiently **transform** 3CNF formula ϕ to $f(\phi) = (G, k)$ s.t.
- ϕ is satisfiable $\iff$ G has a VC of size k.
- **Example:**
  $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$

n variables

graph G,
$k = n + 2m$

m clauses



3SAT $\leq_p$ VERTEXCOVER

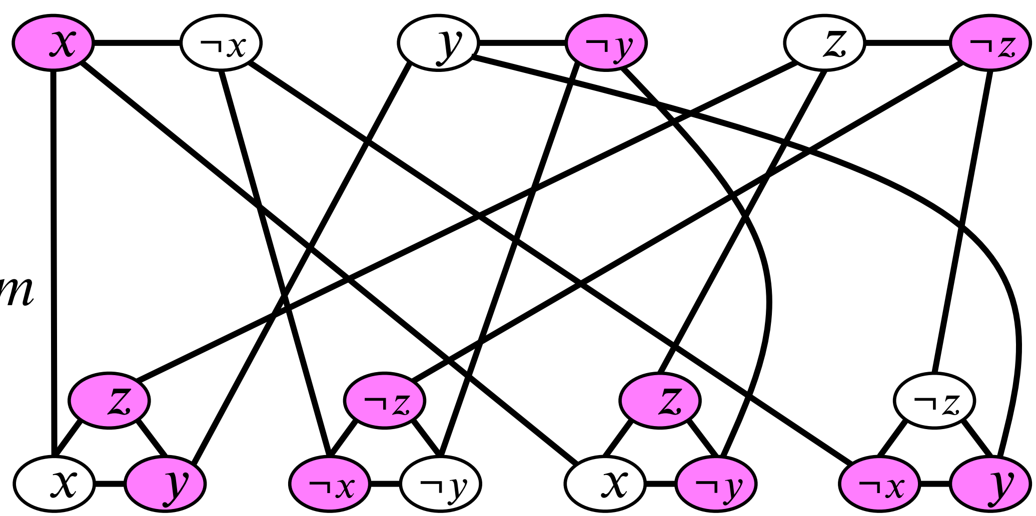
- **If $\phi \in$ 3SAT**: it has a satisfying assignment (e.g., (1,0,0)).
  - We **exhibit a corresponding VC** in G of size $k = n + 2m$.
  - So: $\phi \in$ 3SAT $\implies f(\phi) = (G, k) \in$ VertexCover.
- **Example:**

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

n variables

graph G,
$k = n + 2m$

m clauses



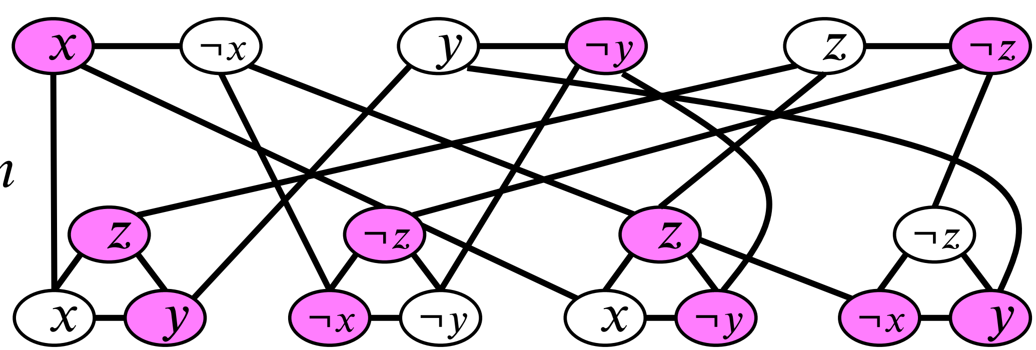
3SAT $\le_p$ VERTEXCOVER

- **If $f(\phi) = (G, k) \in$ VertexCover:** G has some size-k VC.
  - We **exhibit a corresponding satisfying assignment** of ϕ (so $\phi \in$ 3SAT):
  - Any size-k VC *must* include exactly 1 vertex from each variable gadget, and exactly 2 vertices from each clause gadget.
  - In each clause gadget, the *non-selected* vertex's "crossing edge" must be covered by the (selected) vertex of the *same label* in the var gadget.
  - Assign variables so that the *selected vertices* of the variable gadgets have "true" literals. Then, every clause has a true literal!

n variables

$k = n + 2m$

m clauses



General Mapping Reduction: How to Prove It

- To prove that $A \leq_p B$ for **NP-languages** A, B:

1. Give an **efficient transformation** f from A-instances to B-instances. (Note: typically can't decide A efficiently!)
2. Show that $x \in A \iff f(x) \in B$.
   (a) Show that **any A-witness** for x yields a **corresponding B-witness** for $f(x)$…
   (b) And **vice-versa**.

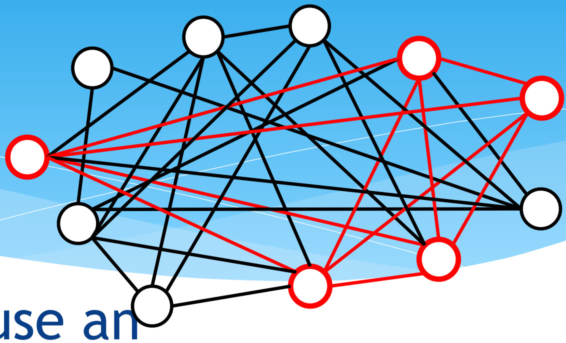
Search Problems

* Real problems often have more than "yes/no" answers.
* Some examples of such ***search*** problems:
  * Given an array of integers, output the sorted array.
  * Given a Boolean formula, output a satisfying assignment.
  * Given a graph, return a max clique / min vertex-cover.
* We've seen ***decision*** versions of these problems.
* **"Theorem:"** An "**NP**-Hard" **search** problem has an efficient algorithm *if and only if* its **decision** variant has an efficient algorithm.

Search vs. Decision

* **Types of Search Problems:**
  * **Maximization:** Maximum Clique, Knapsack
  * **Minimization:** Minimum Vertex Cover, TSP
  * **Exact:** Satisfying assignment, Hamiltonian path/cycle
* **Decision Versions:**
  * Does G have a clique of size k?
  * Does G have a vertex cover of size k?
  * Is ϕ satisfiable?
* **Q:** Given an efficient solver for the decision version, how can we efficiently solve the search problem?

Step 1: Get Size of Optimal Solution



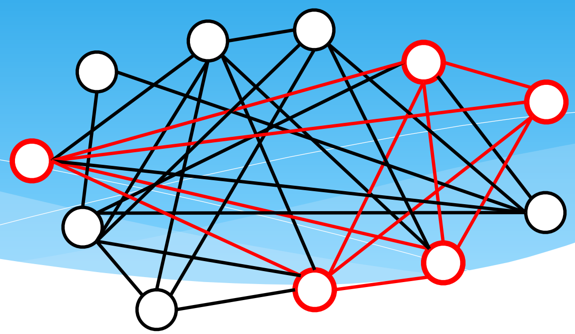
- For optimization problems, we can first use an efficient decider to find the *size* of an optimal solution.
- **Example:** Given a graph G, find a maximum clique in G.
  - Suppose **hasClique**(G, k) *efficiently* solves the decision problem $\text{CLIQUE} = \{(G, k) : G$ is a graph with a clique of size $k\}$.
  - We search over k to efficiently find the maximum clique size.

max-clique-size(G):
1. $k \leftarrow 0$
2. **while hasClique**$(G, k+1)$: $k \leftarrow k + 1$
3. **return** k

  - **Caution:** If k can have more than polynomially many possible values, we need to do a *binary search*.

Step 2: Find an Optimal Solution

max-clique(G):
1. $k \leftarrow$ **max-clique-size**(G)
2. **return find-clique**(G, k)

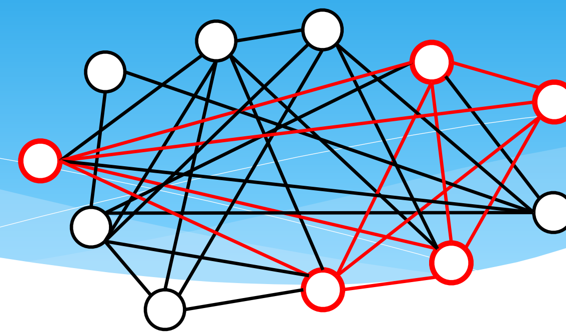


- Once we know the optimal size, we can use the efficient decider to find an optimal solution.
- **Common Strategy #1:** Throw away unneeded parts of the instance until all that is left is a solution.

find-clique(G, k): // precondition: G has a k-clique
1. **for each** vertex $v \in G$:
2. $\quad G' \leftarrow G - v$ // delete vertex v from G to obtain G'
3. $\quad$ **if hasClique**(G', k): // G' still has a k-clique
4. $\quad\quad G \leftarrow G'$ // continue without v, since it's unnecessary
5. **return** $V(G)$ // return the remaining vertices in G

Step 2: Find an Optimal Solution

max-clique(graph G):
1. $k \leftarrow$ **max-clique-size**(G)
2. **return find-clique**(G, k)



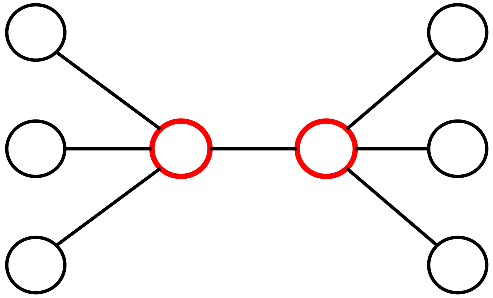
- Once we know the optimal size, we can use the efficient decider to find an optimal solution.
- **Common Strategy #2:** Build up a solution piece-by-piece, guessing the individual pieces.

find-clique(G, k): // precondition: G has a k-clique
1. **if** $k = 0$: **return** $\varnothing$
2. **for each** vertex $v \in G$:
3. $\quad G' \leftarrow$ **neighborhood**(G, v) // v's neighbors and edges among them
4. $\quad$ **if hasClique**$(G', k-1)$: // v's neighbors have a $(k-1)$-clique
5. $\qquad$ **return** $\{v\} \cup$ **find-clique**$(G', k-1)$

Minimum Vertex Cover

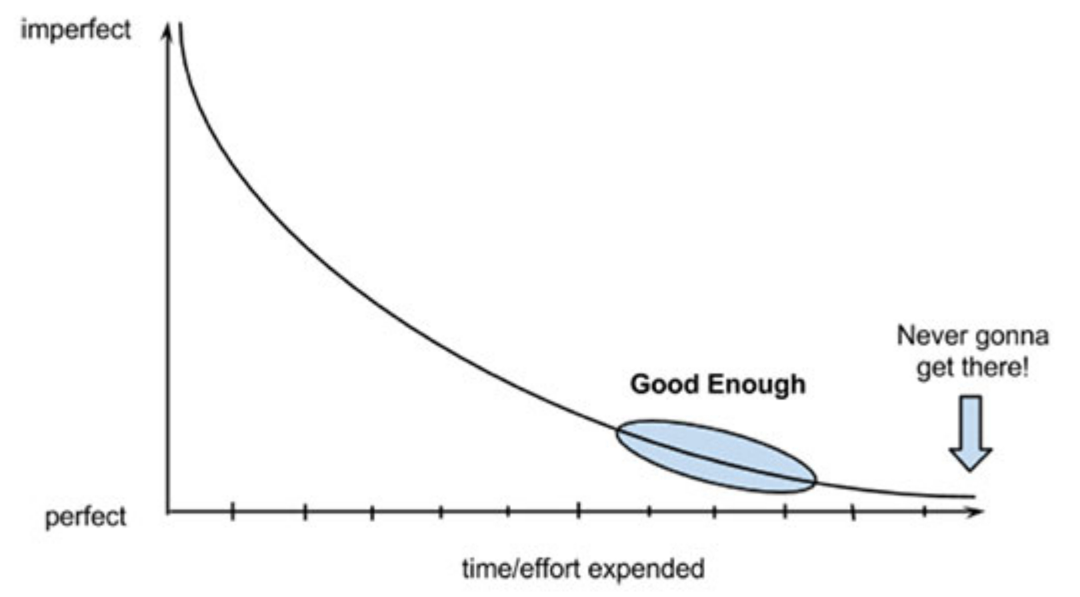
find-VC(G, k): // precondition: G has size-k VC
1. **if** $k = 0$: **return** $\varnothing$
2. **for each** vertex $v \in G$:
3. $\quad G' \leftarrow G - v$ // delete v and all its incident edges
4. $\quad$ **if hasVC**$(G', k-1)$:
5. $\quad\quad$ **return** $\{v\} \cup$ **find-VC**$(G', k-1)$

- **Problem:** Given graph G, return a *smallest* vertex cover of G.
- Suppose we have an *efficient* decider
  hasVC(G, k) = *true* if G has a size-k VC; *false* otherwise
- **Q1:** How can we use **hasVC** to *find the size* of a smallest VC in G?
- **Q2:** Once we know the size, how to use **hasVC** to *find a smallest VC*?
- **Use Strategy #2:** Guess a vertex in the cover, remove it and its edges, check if the remaining graph has a cover of size one smaller.



“Although this may seem a paradox, all exact science is based on the idea of approximation. If a man tells you he knows a thing exactly, then you can be safe in inferring that you are speaking to an inexact man.” – Bertrand Russell

Coping with NP-Completeness



So Your Problem is NP-Hard… Now What?

* Don’t expect an efficient algorithm anytime soon!

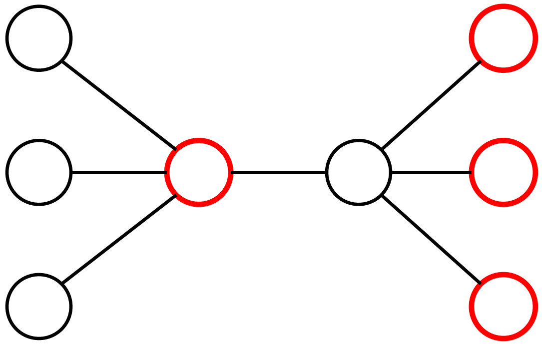
1. Restrict to **special-case inputs**
   * May have efficient algorithms (e.g., planar max-cut)
2. Use **heuristics**: good for “most” “real-world” inputs
   * SAT solvers often do very well in practice!
3. Use an **inefficient** algorithm on **“small” inputs**
   * OK if not too often, and you can afford to wait
4. Devise an efficient **approximation algorithm**
   * Yields an output that is “close” to optimal

Approximation Algorithms

* There are many real-life examples of problems whose decision versions are **NP**-Complete.
  * Max-clique (friends), min-vertex cover (Starbucks), min-set cover (project management), optimal knapsack (robbery), traveling salesperson, …
* While efficiently finding an **optimal** solution seems unattainable for such problems, we might be able to find a "good enough" one: an approximation

Approximating Min Vertex-Cover

- **Starbucks Executive:** "I'm ok with building *at most twice* as many stores as is optimal."
- A vertex cover S is an ***α-approximation*** if S contains *at most* **α** times as many vertices as a smallest one: $|S| \leq \alpha \cdot |C|$ for any VC C.
  - α is called the ***approximation ratio*** (smaller is better here)

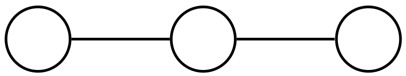


A 2-approximate min-VC (optimum = 2)

Attempt #1: Single Cover

- *Arbitrarily* choose vertices to add to cover.
- **Q:** How large can the approximation ratio be?

cover-and-remove(G):
1. $C \leftarrow \varnothing$
2. **while** G has an edge:
3. choose a vertex v covering *at least one edge*
4. $G \leftarrow G - v$; $C \leftarrow C \cup \{v\}$ // delete/add it to cover
5. **return** C



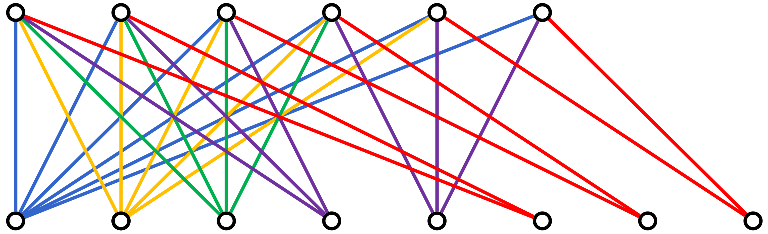
Approx ratio here is 2… can it be worse in other cases?

Result of Single Cover

Attempt #2: Greedy Cover

* Choose vertices that cover the most edges.
* **Fact:** Approximation ratio could be $\Omega(\log n)$!

greedy-cover-and-remove(G):
1. $C \leftarrow \emptyset$
2. **while** G has an edge:
3. choose a vertex v covering <ins>*the most edges*</ins>
4. $G \leftarrow G - v$; $C \leftarrow C \cup \{v\}$ // delete/add it to cover
5. **return** C



Q: What's a smallest vertex cover here?

Q: What's the approx ratio?

Attempt #3: Double Cover

* **Weird Idea:** Choose *edges* and delete *both* endpoints!

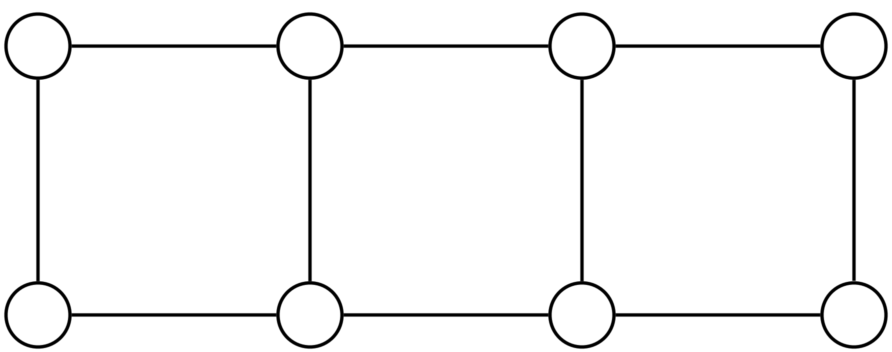
double-cover(G):
1. $C \leftarrow \varnothing$
2. **while** G has an edge:
3. choose any edge $e = (u, v)$
4. $G \leftarrow G - \{u, v\}$; $C \leftarrow C \cup \{u, v\}$ // delete/add **both** endpoints
5. **return** C

Theorem: double-cover obtains a 2-approx to min-vertex-cover.

Example and Key Fact

double-cover(G):
1. $C \leftarrow \varnothing$
2. **while** G has an edge:
3. choose any edge $e = (u, v)$
4. $G \leftarrow G - \{u, v\}$; $C \leftarrow C \cup \{u, v\}$ // delete/add both endpoints
5. **return** C

- **Key Fact:** chosen edges are (vertex-)*disjoint*; output cover has $2 \bullet$ **(# chosen edges)** vertices.
- **Q:** How many vertices are needed to cover a set of *disjoint* edges?
- **Observe:** *Any* cover C^* has *at least* **(# chosen edges)** vertices.



Proof of 2-Approx

double-cover(G):
$C \leftarrow \varnothing$
while G has an edge:
    choose any edge $e = (u, v)$
    $G \leftarrow G - \{u, v\}$; $C \leftarrow C \cup \{u, v\}$ // delete/add both endpoints
return C

* **Theorem:** **double-cover** outputs a 2-approx of min-vertex-cover.
  * Let M be the set of chosen edges and C be the set of vertices of M (i.e., output cover). Then $|C| = 2|M|$.
  * Consider an arbitrary vertex cover C^*.
  * Since M is disjoint and C^* covers it, $|M| \le |C^*|$.
  * Therefore, $|C| = 2|M| \le 2|C^*|$.
* **Claim:** The **double-cover** algorithm is efficient.
  * **Exercise:** Do the analysis to show this is the case.