# 17. Approximation and Vertex Cover

## CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY
FULLERTON

# Big Idea: Renegotiating Problems

Sometimes we want to solve a problem, but there is an obstacle

- computational complexity: problem is *NP*-hard or undecidable
- ill-posed: don't know how to phrase problem as precise input/output statement

These are insurmountable; progress not possible.

Sometimes we can *negotiate* on the definition of the problem

- adjust input/output def'n to correspond to an easier problem
- more specific input; or more general output
- ideally, computational problem still helps with the business problem
- combines CS hard skills with business soft skills

# Approximation

**Approximation**: output is *nearly-optimal* but not necessarily truly optimal.

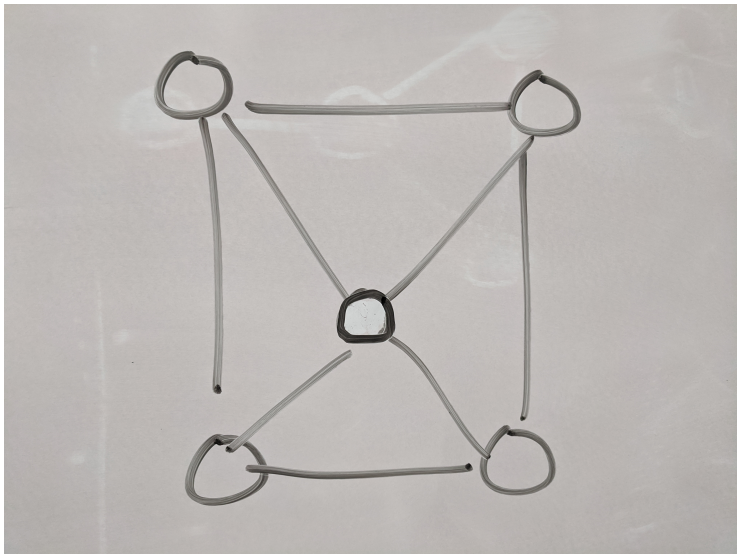
- quality is quantified, **proven**
- "approximation", "approximate" are technical terms; use other words like "decent" for informal ideas about quality
- suitable for use cases where approximate solutions are adequate
- need to rewrite problem definition
- every optimization problem has corresponding approximation problems; but these are distinct problems

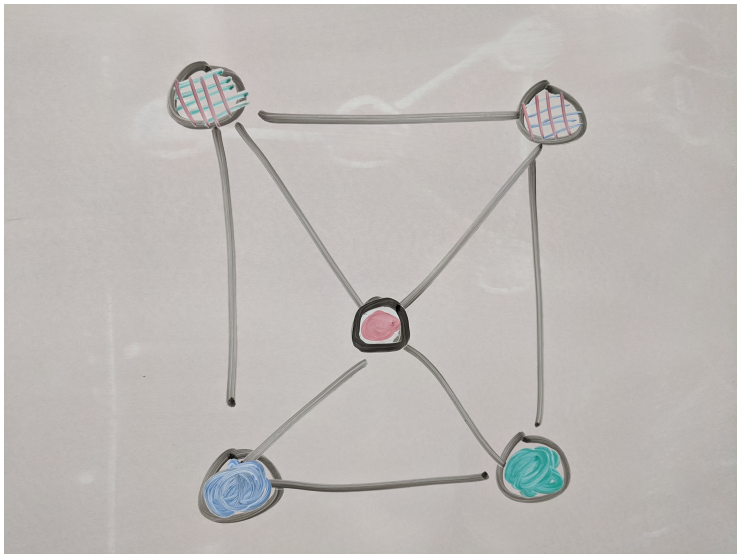
# Example: optimal vs. approximate graph coloring

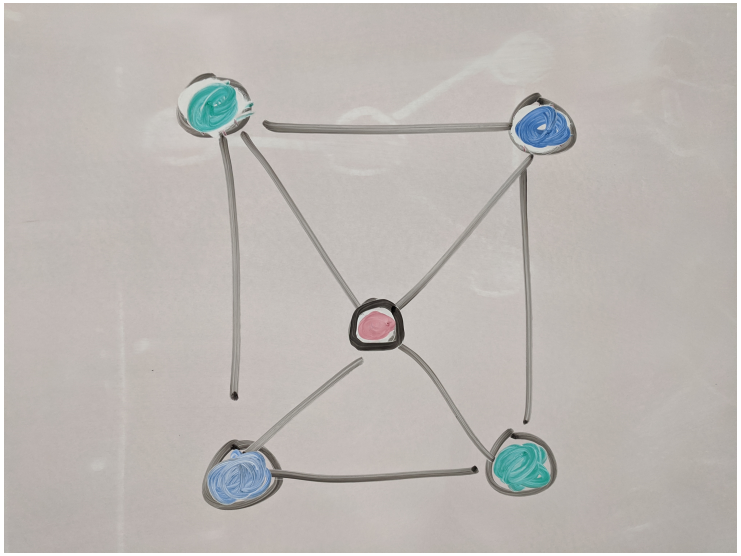
*graph coloring*
**input**: connected graph $G = (V, E)$
**output**: coloring $c$ using $k$ colors, where each vertex $v \in V$ is assigned color $c(v) \in \{1, \ldots k\}$, no pair of adjacent vertices are assigned the same color, and the number of colors $k$ is minimal

*3-approximate graph coloring*
**input**: connected graph $G = (V, E)$
**output**: coloring $c$ using $k$ colors, where each vertex $v \in V$ is assigned color $c(v) \in \{1, \ldots k\}$, no pair of adjacent vertices are assigned the same color, **and the number of colors $k$ satisfied $k \leq 3k^\star$, where $k^\star$ is the fewest colors possible for $G$**

# Graph Coloring Example







# Approximation vs. Other Renegotiation Approaches

Other ways of dealing with hard problems:

- say "no"
- when $n$ is small enough, just use exponential-time algorithm
- no *proof* of solution quality, but nonetheless sometimes good enough:
  - machine learning algorithms (also, in M.L. humans don't need to precisely define what counts as "correct")
  - fast heuristic algorithms
  - Monte Carlo algorithms

Approximation

- pros: *provable* solution quality, often fast
- con: human needs to design and analyze algorithm for each specific problem

# Performance Ratios

**Approximation ratio** $\rho(n)$: ratio between quality of algorithm's output and optimal output; smaller is better

- for **maximization** problem: if optimal quality is $C^\star$ and alg. produces quality $C$, by definition $C^\star \geq C$, and define

$$\rho(n) = \frac{C^\star}{C}$$

- for **minimization** problem: if optimal quality is $C^\star$ and alg. produces quality $C$, by definition $C^\star \leq C$ and define

$$\rho(n) = \frac{C}{C^\star}$$

Recall 3-approx. vertex cover: output # colors $\leq 3k^\star$

# Fixed Approximation Ratios

Some approxmation algorithms have a fixed approximation ratio that is "baked in" to the design of the algorithm.

Ex.: algorithm that solves 3-approx. vertex cover would have fixed $\rho(n) = 3$

In general, better (smaller) ratios require slower algorithms. (note 1-approximation algorithms produce optimal solutions.)

Deriving a different quality-vs.-time trade-off requires designing an entirely different algorithm.

# Approximation Schemes

**approximation scheme**: family of related algorithms, such that, for any parameter $\epsilon > 0$, scheme defines a $(1+\epsilon)$-approximate algorithm

- think of $\epsilon$ as being a **const** variable
- time-performance trade-off is fully tuneable at compile time

**Polynomial Time Approximation Scheme (PTAS)**: approx. scheme where runtime is polynomial in $n$; nothing said of relationship to $\epsilon$
e.g. $O(2^{1/\epsilon} n \log n)$

**Fully PTAS**: runtime is polynomial in $n$ and $1/\epsilon$
e.g. $O((1/\epsilon)^2 n^3)$

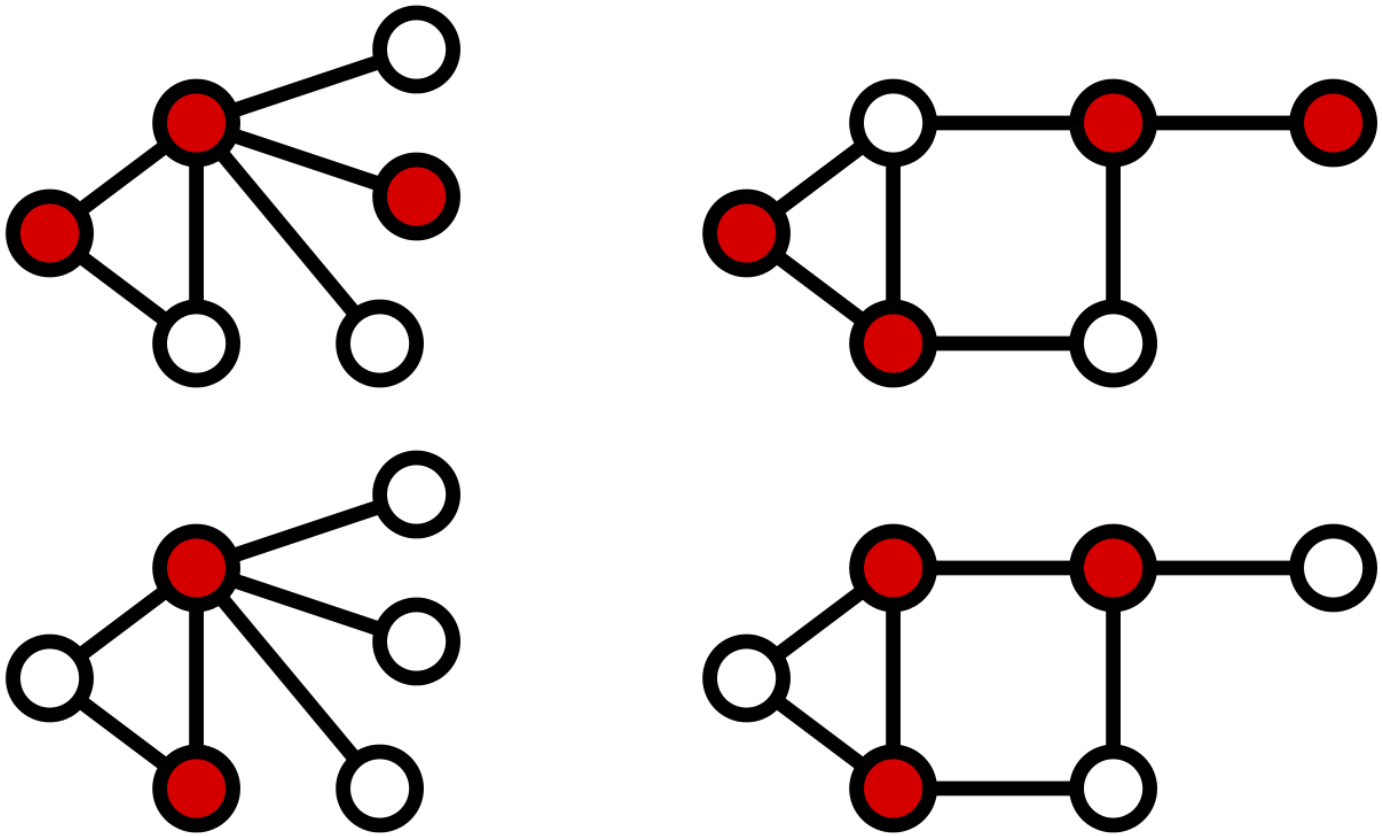
# Vertex Cover Problem

*vertex cover problem*
**input**: undirected graph $G = (V, E)$
**output**: set of vertices $C \subseteq V$, of minimal size $|C|$, such that every edge in $E$ is incident on at least one vertex in $C$

*2-approximate vertex cover problem*
**input**: undirected graph $G = (V, E)$
**output**: set of vertices $C \subseteq V$, such that every edge in $E$ is incident on at least one vertex in $C$, and $|C| \leq 2|C^\star|$ where $C^\star$ is a minimal vertex cover for $G$

See Wiki page:
https://en.wikipedia.org/wiki/Vertex_cover

# Vertex Cover Example



Images credit: Wikipedia user Miym, CC BY-SA 3.0,

https://commons.wikimedia.org/wiki/File:Vertex-cover.svg,

https://commons.wikimedia.org/wiki/File:Minimum-vertex-cover.svg

# Vertex Cover Hardness

- vertex cover is $NP$-complete
- baseline algorithm:
  - exhaustive search
  - for each subset $C$ of vertices, check whether every edge has an endpoint in $C$
  - return the smallest $C$ that is a valid cover
  - $\Theta(2^n m)$ time, exponential, slow
- goal of a 2-approximate vertex cover algorithm:
  - get a decent (though imperfect) cover much faster

# A Greedy Approximation Algorithm

Idea:

- every edge $e = (u, v)$ needs both $u \in C$ and $v \in C$
- so grab an edge $e = (u, v)$ and include $u$ and $v$ in $C$
- every other edge touching $u$ or $v$ is now covered, so eliminate them
- continue until every edge is either grabbed or eliminated
- good: definitely finds a correct cover $C$
- bad: depending on the order of the "grabs", heuristic can get tricked into picking sub-optimal vertices

# 2-Approximate Vertex Cover Pseudocode

```
1: function APPROX-VERTEX-COVER(G = (V, E))
2:     C = ∅
3:     T = E                          ▷ T = edges that may be taken
4:     while T ≠ ∅ do
5:         Let e = (u, v) be an arbitrary edge in T
6:         C = C ∪ {u, v}
7:         Remove from T any edge f that is incident on u or v
8:     end while
9:     return C
10: end function
```

**Efficiency Analysis**: $O(m+n)$ time (assuming efficient data structures for $G$, $T$, and $C$)

# Vertex Cover Performance Ratio

**Lemma**: APPROX-VERTEX-COVER is a 2-approximation algorithm.
**Need:** for any $G, |C| \leq 2|C^\star|$
**Proof sketch**:

- Let $A$ be the set of edges chosen inside the **while** loop
- will bound $|C|, |C^\star|$ both in terms of $|A|$

# Vertex Cover Performance Ratio (cont'd)

- **(1)** $|C^\star|$ vs. $|A|$
- $C^\star$ is a vertex cover, so for every edge $(u, v) \in A$, we must have $u \in C^\star$ and/or $v \in C^\star$
- the "Remove from $T$" step guarantees that, after $(u, v)$ is chosen, no other edge incident on $u$ or $v$ will be chosen and added to $A$
- $\Rightarrow$ each vertex $x \in C^\star$ covers *exactly* one edge in $A$
- $\Rightarrow |C^\star| \geq |A|$

# Vertex Cover Performance Ratio (cont'd)

- **(2)** $|C|$ vs. $|A|$
- the $C = C \cup \{u, v\}$ step inserts 2 vertices into $C$
- due to the same "Remove from $T$" logic, neither $u$ nor $v$ was already in $C$
- $\Rightarrow |C| = 2|A|$ (note exact equality)
- **combining (1) and (2)**

$$|C| = 2|A| \leq 2|C^\star|$$

- QED

# Commentary on this Proof

- note that us analysts do not know concretely which vertices are in $C^\star$
- the algorithm certainly doesn't know what $C^\star$ is, either
- all we do know is that, due to the definition of vertex cover, and the logic of our algorithm,

  $$\#\text{ vertices in optimal cover} \geq \#\text{ iterations } \textbf{while} \text{ loop}$$

- and, due to algorithm logic,

  $$\#\text{ iterations } \textbf{while} \text{ loop} = \#\text{ vertices chosen for approx. cover}$$

- in general, to prove an approx. ratio, need
  1. to bound quality of arbitrary, opaque optimal solution; and
  2. bound quality of approx. solution the same way