

12. Approximation

CPSC 535 ~ Spring 2019

Kevin A. Wortman



May 6, 2019



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.

- ▶ proximity to optimality is quantified, **proven**
- ▶ “approximation”, “approximate” are technical terms; use other words like “decent” for informal ideas about near-optimality
- ▶ suitable for business scenarios 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 $c(v) \in \{1, \dots, k\}$, 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 $c(v) \in \{1, \dots, k\}$, no pair of adjacent vertices are assigned the same color, **and the number of colors k satisfied $k \leq 3k^*$, where k^* is the fewest colors possible for G**

Approximation vs. Other Approaches

Other ways of dealing with unsolvable 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
 - ▶ other AI 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^* and alg. produces quality C , by definition $C^* \geq C$, and define

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

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

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

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

Fixed Approximation Ratios

Some approximation 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 $\rho(n)$ vs. time trade-offs 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 ϵ 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 ϵ
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 $|C| \leq 2|C^*|$ where C^* is a minimal vertex cover for G

See Wiki page:

https://en.wikipedia.org/wiki/Vertex_cover

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:    $C = \emptyset$ 
3:    $T = E$ 
4:   while  $T \neq \emptyset$  do
5:     Let  $e = (u, v)$  be an arbitrary edge in  $T$ 
6:      $C = C \cup \{u, v\}$ 
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: $O(m + n)$, using proper data structures

Vertex Cover Performance Ratio

Lemma: APPROX-VERTEX-COVER is a 2-approximation algorithm.

Need: for any G , $|C| \leq 2|C^*|$

Proof sketch:

- ▶ Let A be the set of edges chosen inside the **while** loop
- ▶ will bound $|C|, |C^*|$ both in terms of $|A|$
- ▶ **(1)** $|C^*|$ vs. $|A|$
- ▶ C^* is a vertex cover, so for every edge $(u, v) \in A$, we must have $u \in C^*$ and/or $v \in C^*$
- ▶ 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 $x \in C^*$ covers *exactly* one edge in A
- ▶ $\Rightarrow |C^*| \geq |A|$

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)**

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

- ▶ QED

Commentary on this Proof

- ▶ note that us analysts do not know concretely which vertices are in C^*
- ▶ the algorithm certainly doesn't know what C^* is, either
- ▶ all we do know is that, due to the definition of vertex cover, and the logic of our algorithm,
$$\# \text{ vertices in optimal cover} \geq \# \text{ iterations } \mathbf{while} \text{ loop}$$
- ▶ and, due to algorithm logic,
$$\# \text{ iterations } \mathbf{while} \text{ loop} = \# \text{ vertices chosen for approx. cover}$$
- ▶ 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

TSP

traveling salesperon problem (TSP)

input: a complete undirected graph $G = (V, E)$ where each edge has weight $w(e) \geq 0$

output: a sequence of vertices H forming a Hamiltonian cycle, minimizing total edge weight

Recall:

- ▶ *Cycle*: path that starts and ends at same vertex
- ▶ *Hamiltonian*: visits each vertex exactly once
- ▶ every complete graph contains some Hamiltonian cycle

Bad news:

- ▶ TSP is *NP*-complete; if $P \neq NP$, no polynomial-time optimization algorithm
- ▶ TSP is also *APX*-complete; if $P \neq NP$, no PTAS

Triangle Inequality

Triangle inequality in general: for distance function d and sites a, b, c ,

$$d(a, c) \leq d(a, b) + d(b, c)$$

\Rightarrow direct path $a \rightarrow b$ always cheaper than two-step path $a \rightarrow b \rightarrow c$ (or tied)

Triangle inequality in a complete graph: for vertices x, y, z and edge weights w ,

$$w(x, z) \leq w(x, y) + w(y, z)$$

\Rightarrow same intuition; adding an intermediate step is never a shortcut

\Rightarrow automatically holds for Euclidean graphs

TSP with Triangle Inequality (TSPTI)

input: a complete undirected graph $G = (V, E)$ where each edge has weight $w(e) \geq 0$; and for any $x, y, z \in V$,
 $w(x, z) \leq w(x, y) + w(y, z)$

output: (same)

- ▶ **renegotiating** TSP
- ▶ different problem; *NP*-completeness and *APX*-completeness proofs may not apply
- ▶ less-general problem
- ▶ probably still relevant to practical applications of TSP

TSPTI Approximation Algorithm Idea

- ▶ need a structure that can lower-bound an optimal cycle H^* and upper-bound our approximate cycle H
- ▶ *minimum spanning tree* features
 - ▶ minimizes weight of chosen edges
 - ▶ connects all vertices
 - ▶ can be computed fast
- ▶ but an MST is not a Hamiltonian cycle; MST is acyclic, for one thing
- ▶ *Euler tour*: cycle around a tree; preorder, inorder, postorder
- ▶ build an MST; perform preorder traversal; treat that vertex order as Hamiltonian cycle

Approximate TSPTI Pseudocode

```
1: function APPROX-TSPTI( $G = (V, E), w$ )
2:    $T = \text{PRIM} - \text{MST}(G, w)$ 
3:    $H =$  empty sequence of vertices
4:   for vertex  $v$  in preorder traversal of tree  $T$  do
5:      $H.\text{ADDBACK}(v)$ 
6:   end for
7:   return  $H$ 
8: end function
```

Analysis: Prim's algorithm takes $O(m + n \log n)$ (w/ Fibonacci heap), traversal takes $O(m + n)$, total $O(m + n \log n)$ time

TSPTI Performance

Lemma: APPROX-TSPTI is a 2-approximation algorithm

Proof Sketch:

- ▶ let H^* be an optimal Hamiltonian cycle for G
- ▶ (1) every spanning tree is one edge short of a cycle; and weights are nonnegative; so the weight of our tree T obeys $w(T) \leq w(H^*)$
- ▶ (2) a *full tour* W is the sequence of vertices in both a preorder and postorder tour, and has weight $w(W) = 2w(T)$
- ▶ (3) combining (1) and (2), $w(W) \leq 2w(H^*)$
- ▶ (4) our H is like W with some vertices removed, so $w(H) \leq w(W)$
- ▶ combining (3) and (4),

$$w(H) \leq w(W) \leq 2w(H^*)$$

Summary

There is a 2-approximation algorithm for vertex cover that takes $O(m + n)$ time.

There is a 2-approximation algorithm for TSP with the triangle inequality that takes $O(m + n \log n)$ time.