Approximation Algorithms

Shahin Kamali

Approximation Algorithms
CLRS 35.1-35.5
University of Manitoba
Approaching a Problem

- Assume you are given a problem $X$
- First, check if $X$ belongs to class NP
  - Try to devise a (possibly naive) polynomial algorithm for $X$ (e.g., a $\Theta(n^3)$ algorithm for 3Sum).
Approaching a Problem

- Assume you are given a problem $X$.
- First, check if $X$ belongs to class NP.
  - Try to devise a (possibly naive) polynomial algorithm for $X$ (e.g., a $\Theta(n^3)$ algorithm for 3Sum).
  - In case the problem is in $P$, try to improve its time complexity (e.g., find a $\Theta(n^2)$ algorithm for 3Sum).
Approaching a Problem

- Assume you are given a problem $X$
- First, check if $X$ belongs to class NP
  - Try to devise a (possibly naive) polynomial algorithm for $X$ (e.g., a $\Theta(n^3)$ algorithm for 3Sum).
  - In case the problem is in $P$, try to improve its time complexity (e.g., find a $\Theta(n^2)$ algorithm for 3Sum).
- If could not devise a polynomial algorithm, try to prove $X$ is NP-hard
  - Reduce another NP-hard problem to $X$ in polynomial time
Approaching a Problem

- Assume you are given a problem \( X \)
- First, check if \( X \) belongs to class \( \text{NP} \)
  - Try to devise a (possibly naive) polynomial algorithm for \( X \) (e.g., a \( \Theta(n^3) \) algorithm for 3Sum).
  - In case the problem is in \( P \), try to improve its time complexity (e.g., find a \( \Theta(n^2) \) algorithm for 3Sum).
- If could not devise a polynomial algorithm, try to prove \( X \) is \( \text{NP-hard} \)
  - Reduce another \( \text{NP-hard} \) problem to \( X \) in polynomial time
  - If the problem is \( \text{NP-hard} \), should we stop?
Approaching an NP-hard problem

- solution 1: become depressed and cry (not recommended)
Approaching an NP-hard problem

- solution 1: become depressed and cry (not recommended)
- solution 2: come up with a heuristic (not recommended either)
Approaching an NP-hard problem

- solution 1: become depressed and cry (not recommended)
- solution 2: come up with a heuristic (not recommended either)
  - Heuristics provide no theoretical guarantee on the performance; they might unexpectedly fail.
Approaching an NP-hard problem

- solution 1: become depressed and cry (not recommended)
- solution 2: come up with a heuristic (not recommended either)
  - Heuristics provide no theoretical guarantee on the performance; they might unexpectedly fail.
- Find efficient algorithms for specific instances of the problem
  - E.g., 3-Coloring is NP-hard for graphs in general, but there is a polynomial algorithm for trees (and some other graph families).
Approaching an NP-hard problem

- solution 1: become depressed and cry (not recommended)
- solution 2: come up with a heuristic (not recommended either)
  - Heuristics provide no theoretical guarantee on the performance; they might unexpectedly fail.
- Find efficient algorithms for **specific instances** of the problem
  - E.g., 3-Coloring is NP-hard for graphs in general, but there is a polynomial algorithm for trees (and some other graph families).
- In case of **optimization problems**, design an **approximation algorithm** which approximates the optimal solution.
**Decision vs Optimization**

- Some problems are decision problems by nature
  - E.g., Hamiltonian path is a natural decision problem, either such path exists or not

In an optimization problem, the goal is to minimize a cost or maximize a profit.

E.g., in bin packing problem, the goal is to minimize the number of bins opened to pack a given multi-set of items.

Bin packing is an optimization problem.

When studying complexity of optimization, we consider their decision variant.

E.g., is it possible to pack a multi-set in 2 bins? (we showed this decision problem is NP-hard)

After hardness of an optimization established, we can study approximation algorithms for it.

E.g., First Fit is an approximation algorithm for bin packing which ensures the number of bins is at most 1.7 times the optimal solution.
Decision vs Optimization

- Some problems are decision problems by nature
  - E.g., Hamiltonian path is a natural decision problem, either such path exists or not

- In an optimization problem, the goal is to **minimize a cost** or **maximize a profit**
  - E.g., in bin packing problem, the goal is to **minimize** the number of bins (**cost**) opened to pack a given multi-set of items.
  - Bin packing is an optimization problem

- When studying complexity of optimization, we consider their decision variant
  - E.g., is it possible to pack a multi-set in 2 bins? (we showed this decision problem is NP-hard)
Decision vs Optimization

- Some problems are decision problems by nature
  - E.g., Hamiltonian path is a natural decision problem, either such path exists or not
- In an optimization problem, the goal is to minimize a cost or maximize a profit
  - E.g., in bin packing problem, the goal is to minimize the number of bins (cost) opened to pack a given multi-set of items.
  - Bin packing is an optimization problem
- When studying complexity of optimization, we consider their decision variant
  - E.g., is it possible to pack a multi-set in 2 bins? (we showed this decision problem is NP-hard)
- After hardness of an optimization established, we can study approximation algorithms for it
  - E.g., First Fit is an approximation algorithm for bin packing which ensures the number of bins is at most 1.7 times the optimal solution.
Some Optimization Problems

- **Graph Coloring**: Color vertices of a graph using a minimum number of colors (cost).
  - We saw in the class that decision variant ‘whether a graph can be colored using 3 colors’ is NP-hard.
  - This can be extended to the case of any $k \geq 3$ color.
Some Optimization Problems

- **Graph Coloring:**
  - Color vertices of a graph using a minimum number of colors (cost).
  - We saw in the class that decision variant ‘whether a graph can be colored using 3 colors’ is NP-hard.
  - This can be extended to the case of any $k \geq 3$ color.

- **Bin packing:**
  - Pack a multi-set of items in minimum number of bins (cost).
  - We saw in the class that deciding whether items can be packed into $k = 2$ is NP-hard.
  - This can be extended to any $k \geq 2$. 
Some Optimization Problems

- **Independent Set**
  - Find the **largest** set of vertices in a given graph s.t. no two are adjacent.
  - It is a maximization problem, you want to **maximize profit** in terms of the **size of independent set**.
  - Left figure is an independent set of size 2 and right is a better independent set of size 4.
  - Independent set is not only NP-hard (its decision variant) but also no ‘good’ approximation algorithm exist for it.

\[
\text{is size (benefit)} = 2 \quad \text{is size (benefit)} = 4
\]
Some Optimization Problems

- Minimum Spanning Tree
  - Connect all vertices of a weighted graph with a minimum total cost
Some Optimization Problems

Minimum Spanning Tree

- Connect all vertices of a weighted graph with a minimum total cost.
- This optimization problem belongs to P (note that not all optimization problems are NP-hard).
Some Optimization Problems

- Traveling Salesperson
Some Optimization Problems

- **Traveling Salesperson**
  - Given a list of $n$ cities and the distances between each pair of cities, what is the **shortest** route that visits each city and returns to the origin city?
  
  - Cost is the length of the route $\rightarrow$ minimization problem
Some Optimization Problems

- Traveling Salesperson
  - Given a list of $n$ cities and the distances between each pair of cities, what is the **shortest** route that visits each city and returns to the origin city?
  - Cost is the length of the rout $\rightarrow$ minimization problem
Some Optimization Problems

- Traveling Salesperson
  - Given a list of $n$ cities and the distances between each pair of cities, what is the shortest route that visits each city and returns to the origin city?
  - Cost is the length of the route $\rightarrow$ minimization problem
  - The problem is NP-hard and efficient approximation algorithms
Some Optimization Problems

- **Minimum Enclosing Disk**
  - Position a wireless antenna that minimizes the maximum distance to a given set of sites.
  - In other words, find a circle with *minimum* radius which covers all set of given points.

This optimization problem is in P; in fact, there is a linear algorithm for finding the best circle.
Some Optimization Problems

Minimum Enclosing Disk

- Position a wireless antenna that minimizes the maximum distance to a given set of sites.
- In other words, find a circle with minimum radius which covers all set of given points.
Some Optimization Problems

- **Minimum Enclosing Disk**
  - Position a wireless antenna that minimizes the maximum distance to a given set of sites.
  - In other words, find a circle with minimum radius which covers all set of given points.
  - This optimization problem is in P; in fact, there is a linear algorithm for finding the best circle.
Some Optimization Problems

Unit Disk Cover

- Position the fewest wireless antennae possible to cover a given set of sites.
- In other words, find a minimum number of circles which cover all set of given points.

This optimization problem is NP-hard, and there are 'good' approximation algorithms for it.
Some Optimization Problems

- **Unit Disk Cover**
  - Position the fewest wireless antennae possible to cover a given set of sites.
  - In other words, find a **minimum number** of circles which cover all set of given points.
Some Optimization Problems

- **Unit Disk Cover**
  - Position the fewest wireless antennae possible to cover a given set of sites.
  - In other words, find a minimum number of circles which cover all set of given points.
  - This optimization problem is NP-hard, and there are ‘good’ approximation algorithms for it.
An optimization problem asks either for **minimizing a cost** (e.g., number of bins in bin packing) or **maximizing a profit** (e.g. number of vertices in an independent set).

While some of optimization problems are in P (e.g., Minimum Spanning Tree), many others are NP-hard

- For hardness, we show their decision variant is NP-hard.
An optimization problem asks either for **minimizing a cost** (e.g., number of bins in bin packing) or **maximizing a profit** (e.g., number of vertices in an independent set).

While some of optimization problems are in P (e.g., Minimum Spanning Tree), many others are NP-hard.

- For hardness, we show their decision variant is NP-hard.

In case of NP-hard problem, we try to devise **approximation algorithms**.
Approximation Algorithms

**Definition**

An algorithm guarantees **approximation factor** $\alpha$ if for every instance $i$ of the problem it returns a solution with cost $C_i$, s.t.

$$C_i \leq \alpha \cdot OPT_i$$

where $OPT_i$ is the cost of the optimum solution for instance $i$. 

E.g., approximation ratio of First Fit for bin packing is 1.7 since the number of bins that First Fit opens for any multi-set is no more than 1.7 times that of an optimal packing for that multi-set.

For analyzing approximation algorithms which run in polynomial-time, instead of focusing on their running time, we analyze their approximation factor. An algorithm with time $\Theta(n^3)$ and approximation ratio 3 is preferred over an algorithm with time $\Theta(n^2)$ and approximation ratio 4.

The above definition is for minimization problems; similar definition exists for maximization problems.
Approximation Algorithms

Definition

An algorithm guarantees **approximation factor** \( \alpha \) if for every instance \( i \) of the problem it returns a solution with cost \( C_i \), s.t.

\[
C_i \leq \alpha OPT_i
\]

where \( OPT_i \) is the cost of the optimum solution for instance \( i \).

- E.g., approximation ratio of First Fit for bin packing is 1.7 since the number of bins that First Fit opens for any multi-set is no more than 1.7 times that of an optimal packing for that multi-set.
Approximation Algorithms

Definition

An algorithm guarantees **approximation factor** $\alpha$ if for every instance $i$ of the problem it returns a solution with cost $C_i$, s.t.

$$C_i \leq \alpha OPT_i$$

where $OPT_i$ is the cost of the optimum solution for instance $i$.

- E.g., approximation ratio of First Fit for bin packing is 1.7 since the number of bins that First Fit opens for any multi-set is no more than 1.7 times that of an optimal packing for that multi-set.

- For analyzing approximation algorithms which run in polynomial-time, instead of focusing on their running time, we analyze their approximation factor

  - An algorithm with time $\Theta(n^3)$ and approximation ratio 3 is preferred over an algorithm with time $\Theta(n^2)$ and approximation ratio 4.
Definition

An algorithm guarantees **approximation factor** $\alpha$ if for every instance $i$ of the problem it returns a solution with cost $C_i$, s.t.

$$C_i \leq \alpha OPT_i$$

where $OPT_i$ is the cost of the optimum solution for instance $i$.

- E.g., approximation ratio of First Fit for bin packing is 1.7 since the number of bins that First Fit opens for any multi-set is no more than 1.7 times that of an optimal packing for that multi-set.

- For analyzing approximation algorithms which run in polynomial-time, instead of focusing on their running time, we analyze their approximation factor

  - An algorithm with time $\Theta(n^3)$ and approximation ratio 3 is preferred over an algorithm with time $\Theta(n^2)$ and approximation ratio 4.

- The above definition is for minimization problems; similar definition exists for maximization problems.
Vertex Cover Problem

Given a graph $G$, find a minimum-cardinality subset $V_0$ of the vertices such that every edge in $G$ has at least one endpoint in $V$. 
Vertex Cover Problem

- Given a graph $G$, find a minimum-cardinality subset $V_0$ of the vertices such that every edge in $G$ has at least one endpoint in $V$. 

Approximation Algorithms

Vertex Cover Problem

- Given a graph $G$, find a minimum-cardinality subset $V_0$ of the vertices such that every edge in $G$ has at least one endpoint in $V$. 

![Graph example](image-url)
Vertex Cover Problem

Given a graph $G$, find a minimum-cardinality subset $V_0$ of the vertices such that every edge in $G$ has at least one endpoint in $V$. Vertex cover is a famous NP-hard problem. We provide a simple algorithm for approximation factor 2, i.e., the number of vertices in our solution is never more than the number of optimal solutions.
Vertex Cover Problem

Given a graph $G$, find a minimum-cardinality subset $V_0$ of the vertices such that every edge in $G$ has at least one endpoint in $V$.

Vertex cover is a famous NP-hard problem.
Vertex Cover Problem

- Given a graph $G$, find a minimum-cardinality subset $V_0$ of the vertices such that every edge in $G$ has at least one endpoint in $V$.
- Vertex cover is a famous NP-hard problem.
- We provide a simple algorithm for approximation factor 2, i.e., the number of vertices in our solution is never more than the number of optimal solutions.
Approximation Algorithm

- Initially all edges are unmarked and output set $S$ of vertices is empty
- Repeat following until all edges are marked

![Graph Example]
Approximation Algorithm

- Initially all edges are unmarked and output set $S$ of vertices is empty
- Repeat following until all edges are marked
  - Select an arbitrary unmarked edge $(u, v)$
  - Add two endpoints $u$ and $v$ of the edge to the set $S$
  - Mark all edges adjacent to $u$ and $v$.

\[ S = \{ 1, 4 \} \]

$Opt$
Approximation Algorithm

- Initially all edges are unmarked and output set $S$ of vertices is empty
- Repeat following until all edges are marked
  - **Select** an arbitrary unmarked edge $(u, v)$
  - Add two endpoints $u$ and $v$ of the edge to the set $S$
  - Mark all edges adjacent to $u$ and $v$.

$$S = \{ 1, 4, 2, 5 \}$$

*Opt*
Approximation Algorithm

- Initially all edges are unmarked and output set $S$ of vertices is empty
- Repeat following until all edges are marked
  - **Select** an arbitrary unmarked edge $(u, v)$
  - Add two endpoints $u$ and $v$ of the edge to the set $S$
  - Mark all edges adjacent to $u$ and $v$.

$$S = \{1, 4, 2, 5, 3, 6\}$$
Approximation Algorithm

- Let $x$ denote the number of selected edges (red edges)

$S = \{1, 4, 2, 5, 3, 6\}$

Opt
Approximation Algorithm

Let $x$ denote the number of selected edges (red edges).

- Cost of the algorithm (size of $S$) is $2x$, i.e. $\text{Alg}_G = 2x$.
Approximation Algorithm

Let $x$ denote the number of selected edges (red edges)

- Cost of the algorithm (size of $S$) is $2x$, i.e. $\text{Alg}_G = 2x$
- No two selected edge $e_1$ and $e_2$ share an endpoint; since if they do, $e_2$ was marked and hence not selected when $e_1$ was selected.

$S = \{1, 4, 2, 5, 3, 6\}$

$\text{Opt}$
Approximation Algorithm

Let $x$ denote the number of selected edges (red edges)

- Cost of the algorithm (size of $S$) is $2x$, i.e. $\text{Alg}_G = 2x$
- No two selected edge $e_1$ and $e_2$ share an endpoint; since if they do, $e_2$ was marked and hence not selected when $e_1$ was selected.
- $\Rightarrow$ the same vertex cannot be used to cover two of the red edges

$S = \{1, 4, 2, 5, 3, 6\}$

$\text{Opt}$
Approximation Algorithm

- Let $x$ denote the number of selected edges (red edges)
  - Cost of the algorithm (size of $S$) is $2x$, i.e., $\text{Alg}_G = 2x$
  - No two selected edge $e_1$ and $e_2$ share an endpoint; since if they do, $e_2$ was marked and hence not selected when $e_1$ was selected.
  - → the same vertex cannot be used to cover two of the red edges
  - Hence, in any solution (in particular optimal solution), the number of selected vertices should be at least equal to the number of red edges, i.e., $\text{Opt}_G \geq x$. 

$$S = \{1, 4, 2, 5, 3, 6\}$$

[Graph showing $S$ and Opt]
Let $x$ denote the number of selected edges (red edges)

- Cost of the algorithm (size of $S$) is $2x$, i.e. $\text{Alg}_G = 2x$
- No two selected edge $e_1$ and $e_2$ share an endpoint; since if they do, $e_2$ was marked and hence not selected when $e_1$ was selected.
- $\rightarrow$ the same vertex cannot be used to cover two of the red edges
- Hence, in any solution (in particular optimal solution), the number of selected vertices should be at least equal to the number of red edges, i.e., $\text{Opt}_G \geq x$.

The cost of the algorithm ($2x$) is no more than twice the cost of optimal solution (which is at least $x$), i.e., the approximation factor of the algorithm is at most 2.

$S = \{1, 4, 2, 5, 3, 6\}$

Opt
Vertex Cover Summary

- The problem is NP-complete
- We showed there is an approximation algorithm with approximation factor 2.
Vertex Cover Summary

- The problem is NP-complete.
- We showed there is an approximation algorithm with approximation factor 2.
- The best existing approximation ratio is \(2 - \frac{\ln \ln m}{\ln m} = 2 - o(1)\).
- Unless \(P = NP\), no algorithm can give an approximation ratio better than \(\frac{7}{6}\).
- If the unique game conjecture is correct, we cannot approximate vertex cover with a constant ratio better than 2.
The problem is NP-complete

We showed there is an approximation algorithm with approximation factor 2.

The best existing approximation ratio is $2 - \frac{\ln \ln m}{\ln m} = 2 - o(1)$.

Unless $P = NP$, no algorithm can give an approximation ratio better than $7/6$. 
The problem is NP-complete

We showed there is an approximation algorithm with approximation factor 2.

The best existing approximation ratio is \(2 - \frac{\ln \ln m}{\ln m} = 2 - o(1)\).

Unless \(P = NP\), no algorithm can give an approximation ratio better than 7/6.

If unique game conjecture is correct, we cannot approximate vertex cover with a constant ratio better than 2.
Geometric $k$-center

- Given a set of $n$ points (clients) in the plane and a positive integer $k$, find a set of $k$ ‘centers’ (facilities) s.t. the maximum distance of any point (client) to its closest facility is minimized.
Geometric $k$-center

- Given a set of $n$ points (clients) in the plane and a positive integer $k$, find a set of $k$ ‘centers’ (facilities) such that the maximum distance of any point (client) to its closest facility is minimized.

The problem is NP-hard. Unless $P = NP$, no algorithm can give an approximation ratio better than 2. We provide a simple algorithm with an approximation factor of 2.
Geometric $k$-center

- Given a set of $n$ points (clients) in the plane and a positive integer $k$, find a set of $k$ ‘centers’ (facilities) s.t. the maximum distance of any point (client) to its closest facility is minimized.
- The problem is NP-hard
Geometric $k$-center

- Given a set of $n$ points (clients) in the plane and a positive integer $k$, find a set of $k$ ‘centers’ (facilities) s.t. the maximum distance of any point (client) to its closest facility is minimized.

- The problem is NP-hard

- Unless $P = NP$, no algorithm can give an approximation ratio better than 2.
Geometric $k$-center

- Given a set of $n$ points (clients) in the plane and a positive integer $k$, find a set of $k$ ‘centers’ (facilities) s.t. the maximum distance of any point (client) to its closest facility is minimized.

- The problem is NP-hard

- Unless $P = NP$, no algorithm can give an approximation ratio better than 2.

- We provide a simple algorithm with approximation factor of 2.
Approximation Algorithm for $k$-center

- Select an arbitrary point (client) and declare it as the first center $c_1$.
- For $i = 2, 3, \ldots k$, repeat the following:
  - Select the point (client) $x$ with maximum distance $d_i$ to its nearest center among declared centers.
  - Declare $x$ as the next center $c_i$. 

![Diagram showing the algorithm process]
Approximation Algorithm for $k$-center

- Select an arbitrary point (client) and declare it as the first center $c_1$.
- For $i = 2, 3, \ldots k$, repeat the following:
  - Select the point (client) $x$ with maximum distance $d_i$ to its nearest center among declared centers.
  - Declare $x$ as the next center $c_i$. 

![Diagram](image.png)
Approximation Algorithm for \( k \)-center

- Select an arbitrary point (client) and declare it as the first center \( c_1 \).
- For \( i = 2, 3, \ldots k \), repeat the following:
  - Select the point (client) \( x \) with maximum distance \( d_i \) to its nearest center among declared centers.
  - Declare \( x \) as the next center \( c_i \).
Approximation Algorithm for $k$-center

- Select an arbitrary point (client) and declare it as the first center $c_1$.
- For $i = 2, 3, \ldots k$, repeat the following:
  - Select the point (client) $x$ with maximum distance $d_i$ to its nearest center among declared centers.
  - Declare $x$ as the next center $c_i$. 

![Diagram showing the algorithm process](image)
Analysis of Algorithm

Let \( p \) the point with maximum distance \( m \) to its closest center; so the cost of the algorithm is \( C_{\text{Alg}} = m \).
Let $p$ the point with maximum distance $m$ to its closest center; so
the cost of the algorithm is $C_{Alg} = m$.

We claim $d_2 \geq d_3 \geq \ldots \geq d_k \geq c_{Alg}$
Let $p$ the point with maximum distance $m$ to its closest center; so the cost of the algorithm is $C_{Alg} = m$.

We claim $d_2 \geq d_3 \geq \ldots \geq d_k \geq c_{Alg}$

- At the time $c_2$ was selected, its distance $d_2$ to $c_1$ was more than distance of any other point, in particular $c_3$ to $c_1$.
- $d_2 = \text{dist}(c_2, c_1) \geq \text{dist}(c_3, c_1) \geq \min(\text{dist}(c_3, c_1), \text{dist}(c_3, c_2)) = d_3$
Let $p$ the point with maximum distance $m$ to its closest center; so the cost of the algorithm is $C_{\text{Alg}} = m$.

We claim $d_2 \geq d_3 \geq \ldots \geq d_k \geq c_{\text{Alg}}$

- At the time $c_2$ was selected, its distance $d_2$ to $c_1$ was more than distance of any other point, in particular $c_3$ to $c_1$.
- $d_2 = \text{dist}(c_2, c_1) \geq \text{dist}(c_3, c_1) \geq \min(\text{dist}(c_3, c_1), \text{dist}(c_3, c_2)) = d_3$
- Using similar argument, we can show $d_i \geq d_{i-1}$.
- $d_i = \min(\text{dist}(c_i, c_1), \text{dist}(c_i, c_2), \ldots, \text{dist}(c_i, c_{i-1})) \geq \min(\text{dist}(c_{i+1}, c_1), \text{dist}(c_{i+1}, c_2), \ldots, \text{dist}(c_{i+1}, c_{i-1})) \geq \min(\text{dist}(c_{i+1}, c_1), \text{dist}(c_{i+1}, c_2), \ldots, \text{dist}(c_{i+1}, c_{i-1}, \text{dist}(c_{i+1}, c_i)) = d_{i+1}$
Analysis of Algorithm

- We have $d_2 \geq d_3 \geq \ldots \geq d_k \geq c_{\text{Alg}}$
- It means that pairwise distance of any two points in set $\{c_1, c_2, \ldots, c_k, p\}$ is at least $m$.
- There are $k + 1$ points of pairwise distance $m$.
  - In the optimal solution, at least two of them share their closest center
  - If two points of distance $m$ share a center, the distance of the center to one of them is at least $m/2$
  - The cost of $\text{OPT}$ is at least $m/2$. 
The cost of the algorithm is $m$.

The cost of $\text{OPT}$ is at least $m/2$.

Hence the approximation factor is at most 2.

Unless $P=NP$, it is not possible to improve this approximation factor.
PTAS

- Some NP-hard problems can be approximated closely with a PTAS.
- **Polynomial-Time Approximation Scheme (PTAS)**
  - A PTAS for a given problem $A$ is an algorithm that returns a solution to $A$ with approximation factor at most $(1 + \epsilon)$ in polynomial time for every $\epsilon > 0$. 

- For an NP-hard problem which has a PTAS:
  - An optimal solution cannot be found (unless P=NP).
  - However, any degree of approximation can be achieved (arbitrarily close to 1). For example, we can get solutions whose cost is $1.000001$ times the cost of $Opt$ (here $\epsilon = 0.000001$).
  - A trade-off between time complexity and quality of approximation: for better approximations (approximation factor close to 1, i.e., smaller $\epsilon$), the time complexity increases, but remains polynomial for any constant value of $\epsilon$.
    - E.g., the running time of a PTAS can be $\Theta(n^{1/\epsilon})$.
    - For approximation ratio of 1.01, the time complexity will be $\Theta(n^{100})$ and for ratio 1.001, the time complexity will be $\Theta(n^{1000})$. 

PTAS

- Some NP-hard problems can be approximated closely with a PTAS.

**Polynomial-Time Approximation Scheme (PTAS)**

- A PTAS for a given problem $A$ is an algorithm that returns a solution to $A$ with approximation factor at most $(1 + \epsilon)$ in polynomial time for every $\epsilon > 0$.

- For an NP-hard problem which has a PTAS:
  - An optimal solution cannot be found (unless $P=NP$).
Some NP-hard problems can be approximated closely with a PTAS.

**Polynomial-Time Approximation Scheme (PTAS)**

A PTAS for a given problem $A$ is an algorithm that returns a solution to $A$ with approximation factor at most $(1 + \epsilon)$ in polynomial time for every $\epsilon > 0$.

For an NP-hard problem which has a PTAS:

- An optimal solution cannot be found (unless $P=NP$).
- However, any degree of approximation can be achieved (arbitrary close to 1). For example, we can get solutions whose cost is $1.000001$ times the cost of $OPT$ (here $\epsilon = 0.000001$).
Some NP-hard problems can be approximated closely with a PTAS.

**Polynomial-Time Approximation Scheme (PTAS)**

- A PTAS for a given problem $A$ is an algorithm that returns a solution to $A$ with approximation factor at most $(1 + \epsilon)$ in polynomial time for every $\epsilon > 0$.

For an NP-hard problem which has a PTAS:

- An optimal solution cannot be found (unless P=NP).
- However, any degree of approximation can be achieved (arbitrary close to 1). For example, we can get solutions whose cost is 1.000001 times the cost of $OPT$ (here $\epsilon = 0.000001$).
- A trade-off between time complexity and quality of approximation: for better approximations (approximation factor close to 1, i.e., smaller $\epsilon$), the time complexity increases, but remains polynomial for any constant value of $\epsilon$. 


**PTAS**

- Some NP-hard problems can be approximated closely with a PTAS.

**Polynomial-Time Approximation Scheme (PTAS)**

- A PTAS for a given problem \( A \) is an algorithm that returns a solution to \( A \) with approximation factor at most \((1 + \epsilon)\) in polynomial time for every \( \epsilon > 0 \).

- For an NP-hard problem which has a PTAS:
  - An optimal solution cannot be found (unless P=NP).
  - However, any degree of approximation can be achieved (arbitrary close to 1). For example, we can get solutions whose cost is 1.000001 times the cost of \( \text{OPT} \) (here \( \epsilon = 0.000001 \)).
  - A trade-off between time complexity and quality of approximation: for better approximations (approximation factor close to 1, i.e., smaller \( \epsilon \)), the time complexity increases, but remains polynomial for any constant value of \( \epsilon \).

- E.g., the running time of a PTAS can be \( \Theta(n^{1/\epsilon}) \).
  - For approximation ratio of 1.01, the time complexity will be \( \Theta(n^{100}) \) and for ratio 1.001, the time complexity will be \( \Theta(n^{1000}) \).
APX-hard Problems

For many NP-hard problems, known as APX-hard problems, no PTAS is known.
For many NP-hard problems, known as **APX-hard problems**, no PTAS is known.

Suppose you search for a PTAS for a given NP-hard problem, but fail to find one.

P $\neq$ NP implies that no PTAS exists for any APX-hard problem.
For many NP-hard problems, known as **APX-hard problems**, no PTAS is known.

Suppose you search for a PTAS for a given NP-hard problem, but fail to find one.

How can you determine whether you should keep searching or whether no PTAS is possible?

Use an **approximation-preserving reduction** to show that your problem is at least as hard another known APX-hard problem.

\( P \neq NP \) implies that no PTAS exists for any APX-hard problem.
Approximation-preserving Reduction

Unlike NP-hardness reduction where we reduce one decision problem to another, here we reduce one optimization problem to another.

Assume we want to reduce optimization problem A to problem B

Given an instance of \( i \), create an instance \( f(i) \) of B in polynomial time (similar to NP-hardness reduction).

From a solution \( y \) for \( f(i) \) (instance of B), we should be able to get a solution \( g(y) \) for \( i \) (instance of A).

Show that \( g(y) \) is a \( (1 + \epsilon \cdot h(\epsilon)) \)-approximate solution to the instance \( i \) of A iff \( y \) is a \( (1 + \epsilon) \)-approximate solution to the instance \( f(i) \) of Problem B.

Hence if \( B \in \text{PTAS} \rightarrow A \in \text{PTAS} \) as well.

More importantly, if \( A \not\in \text{PTAS} \rightarrow B \not\in \text{PTAS} \)

If we know \( A \) is APX-hard, then \( B \) is also APX-hard!
Approximation-preserving Reduction

Unlike NP-hardness reduction where we reduce one decision problem to another, here we reduce one optimization problem to another.

Assume we want to reduce optimization problem A to problem B

- Given an instance of $i$, create an instance $f(i)$ of $B$ in polynomial time (similar to NP-hardness reduction).

Show that $g(y)$ is a $(1 + \epsilon)$-approximate solution to the instance $i$ of A iff $y$ is a $(1 + \epsilon)$-approximate solution to the instance $f(i)$ of problem B.

Hence if $B \in \text{PTAS} \rightarrow A \in \text{PTAS}$ as well.

More importantly, if $A \not\in \text{PTAS} \rightarrow B \not\in \text{PTAS}$ If we know A is APX-hard, then B is also APX-hard!
Approximation-preserving Reduction

Unlike NP-hardness reduction where we reduce one decision problem to another, here we reduce one optimization problem to another.

Assume we want to reduce optimization problem A to problem B

- Given an instance of $i$, create an instance $f(i)$ of $B$ in polynomial time (similar to NP-hardness reduction).
- From a solution $y$ for $f(i)$ (instance of $B$), we should be able to get a solution $g(y)$ for $i$ (instance of $A$).
- Show that $g(y)$ is a $(1 + h(\epsilon))$-approximate solution to the instance $i$ of $A$ iff $y$ is a $(1 + \epsilon)$-approximate solution to the instance $f(i)$ of Problem $B$. 

Hence if $B \in \text{PTAS}$ $\rightarrow$ $A \in \text{PTAS}$ as well.

More importantly, if $A \not\in \text{PTAS}$ $\rightarrow$ $B \not\in \text{PTAS}$ if we know $A$ is APX-hard, then $B$ is also APX-hard!
Approximation-preserving Reduction

Unlike NP-hardness reduction where we reduce one decision problem to another, here we reduce one optimization problem to another.

Assume we want to reduce optimization problem A to problem B

- Given an instance of $i$, create an instance $f(i)$ of B in polynomial time (similar to NP-hardness reduction).
- From a solution $y$ for $f(i)$ (instance of B), we should be able to get a solution $g(y)$ for $i$ (instance of A).
- Show that $g(y)$ is a $(1 + h(\epsilon))$-approximate solution to the instance $i$ of A iff $y$ is a $(1 + \epsilon)$-approximate solution to the instance $f(i)$ of Problem B.

Hence if $B \in PTAS \rightarrow A \in PTAS$ as well.

More importantly, if $A \notin PTAS \rightarrow B \notin PTAS$

- If we know A is APX-hard, then B is also APX-hard!
A problem is **APX-complete** if it is APX-hard, and it has a constant-factor polynomial-time approximation algorithm.

- It is ‘easy’ in the sense that there is a constant-factor approximation algorithm for it.
- It is ‘hard’ in the sense that no PTAS exists for it (assuming $P \neq NP$)
Problems by Approximation Complexity

- PTAS problems: we can approximate by any constant factor close to 1.
  - E.g., Euclidean Travelling Salesperson, Knapsack problem

- APX-complete problems: there is no PTAS (assuming P \neq NP) but there is a constant-factor approximation algorithm.
  - E.g., Vertex-cover, Euclidean k-center, Independent set when vertices have constant degree, Dominating set.

- For some APX-hard problems, no constant-factor optimization is known (i.e., they are not APX-complete). Instead, there are approximation algorithms which are not as good:
  - Art-Gallery problem: there is a $O(\log n)$-approximation but no constant-factor approximation is known.

- There are problems for which any approximation algorithm has factor $\Omega(n^{1-\epsilon})$.

- Independent set and Vertex Coloring: There are algorithms with approximation factors $O(n \log c_n)$. 
Problems by Approximation Complexity

- **PTAS problems**: we can approximate by any constant factor close to 1.
  - E.g., Euclidean Travelling Salesperson, Knapsack problem
- **APX-complete problems**: there is no PTAS (assuming $P \neq NP$) but there is a constant-factor approximation algorithm
  - E.g., Vertex-cover, Euclidean $k$-center, Independent set when vertices have constant degree, Dominating set.
Problems by Approximation Complexity

- PTAS problems: we can approximate by any constant factor close to 1.
  - E.g., Euclidean Travelling Salesperson, Knapsack problem
- APX-complete problems: there is no PTAS (assuming $P \neq NP$) but there is a constant-factor approximation algorithm
  - E.g., Vertex-cover, Euclidean $k$-center, Independent set when vertices have constant degree, Dominating set.
- For some APX-hard problems, no constant-factor optimization is known (i.e., they are not APX-complete). Instead, there are approximation algorithm which are not as good:
  - Art-Gallery problem: there is a $O(\log n)$-approximation but no constant-factor approximation is known.
Problems by Approximation Complexity

- **PTAS problems**: we can approximate by any constant factor close to 1.
  - E.g., Euclidean Travelling Salesperson, Knapsack problem
- **APX-complete problems**: there is no PTAS (assuming $P \neq NP$) but there is a constant-factor approximation algorithm
  - E.g., Vertex-cover, Euclidean $k$-center, Independent set when vertices have constant degree, Dominating set.
- For some APX-hard problems, no constant-factor optimization is known (i.e., they are not APX-complete). Instead, there are approximation algorithm which are not as good:
  - Art-Gallery problem: there is a $O(\log n)$-approximation but no constant-factor approximation is known.
- There are problems for which any approximation algorithm has factor $\Omega(n^{1-\epsilon})$
  - Independent set and Vertex Coloring
  - There are algorithms with approximation factors $O(n \log^c n)$.
Approximations & Linear Programs

Assume we want to provide an approximation algorithm for vertex cover with input graph $G = (V, E)$.

Define a variable $a_i \in \{0, 1\}$ for each vertex $v_i$.

- $a_i = 0 \rightarrow v_i$ is included in the vertex cover.
- $a_i = 1 \rightarrow v_i$ is excluded in the vertex cover.

An algorithm should set values for $a_i$'s so that each edge is covered by at least one vertex, i.e.,

$$\forall e = (v_i, v_j) \in E, a_i + a_j \geq 1,$$

i.e., at least one of the endpoints of $e$ is in the vertex cover.

This can be written as the following integer program

$$\text{Minimize } \sum_{v_i \in V} a_i \text{ s.t. } \forall e = (v_i, v_j) \in E, a_i + a_j \geq 1$$

For all $a_i$'s we have $a_i \geq 0$ and $a_i \leq 1$.

The above formulation was a linear program if we allowed variables ($a_i$'s) to be real values instead of integers.
Assume we want to provide an approximation algorithm for vertex cover with input graph $G = (V, E)$.

Define a variable $a_i \in \{0, 1\}$ for each vertex $v_i$.

- $a_i = 0 \rightarrow v_i$ is included in the vertex cover.
- $a_i = 1 \rightarrow v_i$ is excluded in the vertex cover.

An algorithm should set values for $a_i$’s so that each edge is covered by at least one vertex, i.e., $\forall e = (v_i, v_j) \in E, a_i + a_j \geq 1$, i.e., at least one of the endpoints of $e$ is in the vertex cover.
Approximations & Linear Programs

- Assume we want to provide an approximation algorithm for vertex cover with input graph $G = (V, E)$.

- Define a variable $a_i \in \{0, 1\}$ for each vertex $v_i$
  - $a_i = 0 \rightarrow v_i$ is included in the vertex cover.
  - $a_i = 1 \rightarrow v_i$ is excluded in the vertex cover.

- An algorithm should set values for $a_i$’s so that each edge is covered by at least one vertex, i.e., $\forall e = (v_i, v_j) \in E, a_i + a_j \geq 1$, i.e., at least one of the endpoints of $e$ is in the vertex cover. This can be written as the following integer program

$$\text{Minimize } \sum_{v_i \in V} a_i \text{ s.t. }$$
$$\forall e = (v_i, v_j) \in E, a_i + a_j \geq 1$$
$$\text{For all } a_i \text{s’ we have } a_i \geq 0 \text{ and } a_i \leq 1.$$
Approximations & Linear Programs

- Assume we want to provide an approximation algorithm for vertex cover with input graph \( G = (V, E) \).
- Define a variable \( a_i \in \{0, 1\} \) for each vertex \( v_i \)
  - \( a_i = 0 \) → \( v_i \) is included in the vertex cover.
  - \( a_i = 1 \) → \( v_i \) is excluded in the vertex cover.
- An algorithm should set values for \( a_i \)'s so that each edge is covered by at least one vertex, i.e., \( \forall e = (v_i, v_j) \in E, a_i + a_j \geq 1 \), i.e., at least one of the endpoints of \( e \) is in the vertex cover. This can be written as the following integer program
  
  Minimize \( \sum_{v_i \in V} a_i \) s.t.
  \( \forall e = (v_i, v_j) \in E, a_i + a_j \geq 1 \)
  For all \( a_i \)'s' we have \( a_i \geq 0 \) and \( a_i \leq 1 \).
- The above formulation was a linear program if we allowed variables (\( a_i \)'s) to be real values instead of integers.
Many NP-hard problems can be formulated as integer programs

- Finding an optimal solution for an integer program is NP-hard
- There is a polynomial time algorithms for linear programs (E.g., Simplex algorithm)

To provide an approximation algorithm for a problem P:

1. Formulate P as an Integer program
2. Treat it as a Linear program and find optimal real-values for Integer programs
3. 'Round' the real values to integer values to get an approximation.

Rounding should be done in a way that:

- The result is a feasible solution (e.g., each edge has an endpoint in the vertex cover after rounding)
- The approximation factor is bounded

Many approximation algorithms for different problems are based on solving Linear programs.
Approximations & Linear Programs

- Many NP-hard problems can be formulated as integer programs
  - Finding an optimal solution for an integer program is NP-hard
  - There is a polynomial time algorithms for linear programs (E.g., Simplex algorithm)

- To provide an approximation algorithm for a problem $P$:
  - Formulate $P$ as an Integer program
  - Treat it as a Linear program and find optimal real-values for Integer programs
  - ‘Round’ the real values to integer values to get an approximation.
    - E.g., values $\leq 1/2$ are rounded to 0 and those larger than 1/2 round to 1.
Many NP-hard problems can be formulated as integer programs.

Finding an optimal solution for an integer program is NP-hard.

There is a polynomial time algorithms for linear programs (e.g., Simplex algorithm).

To provide an approximation algorithm for a problem $P$:

- Formulate $P$ as an Integer program.
- Treat it as a Linear program and find optimal real-values for Integer programs.
- ‘Round’ the real values to integer values to get an approximation.
  - E.g., values $\leq 1/2$ are rounded to 0 and those larger than 1/2 round to 1.
- Rounding should be done in a way that:
  - The result is a feasible solution (e.g., each edge has an endpoint in the vertex cover after rounding).
  - The approximation factor is bounded.
Many NP-hard problems can be formulated as integer programs

Finding an optimal solution for an integer program is NP-hard
There is a polynomial time algorithms for linear programs (E.g., Simplex algorithm)

To provide an approximation algorithm for a problem $P$:

Formulate $P$ as an Integer program

Treat it as a Linear program and find optimal real-values for Integer programs

‘Round’ the real values to integer values to get an approximation.

E.g., values $\leq 1/2$ are rounded to 0 and those larger than 1/2 round to 1.

Rounding should be done in a way that:

The result is a feasible solution (e.g., each edge has an endpoint in the vertex cover after rounding)
The approximation factor is bounded

Many approximation algorithms for different problems are based on solving Linear programs.
The Ending

Observation

You should aim for the stars - and hopefully avoid ending up in the clouds!  
Roxanne McKee

- We covered some materials about algorithms & complexity; the goal was not to cover everything; but prepare you to get interested and discover yourself in your future career.

- When dealing with a problem, we are interested in:
  - designing algorithms for them (using tools such as data structures)
  - analyzing algorithms (based on time complexity, memory requirement, approximation ratio, etc.) to provide guarantees.
  - understanding the restrictions of algorithms (lower bounds and complexity classes).

- 99 percent of people who talk about algorithms (e.g., in media, news, etc.) don’t understand them. Hopefully you are not one of them any more.
The Ending

Observation

You should aim for the stars - and hopefully avoid ending up in the clouds!  

Roxanne McKee

- Template for final will be posted. If any thing in the slides is not clear, ask me to explain it on Piazza.
The Ending

Observation

You should aim for the stars - and hopefully avoid ending up in the clouds!  

Roxanne McKee

- Template for final will be posted. If any thing in the slides is not clear, ask me to explain it on Piazza.
- Your feedback is appreciated; if something can be improved (which is 100 percent the case), let me know.
The Ending

Observation

You should aim for the stars - and hopefully avoid ending up in the clouds!  Roxanne McKee

- Template for final will be posted. If any thing in the slides is not clear, ask me to explain it on Piazza.

- Your feedback is appreciated; if something can be improved (which is 100 percent the case), let me know.

- I hope to see you in Fall 2018 in the course on Online Algorithms