Design and Analysis of Algorithms

Approximation Algorithms – Part 2

Pan Peng

School of Computer Science and Technology
University of Science and Technology of China

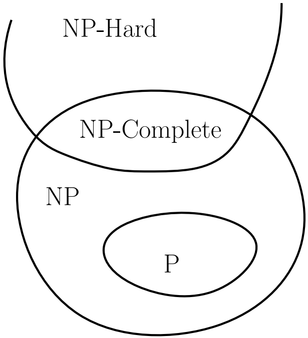
Outline

- Basics of approximation algorithms

- Some examples
  - Knapsack
  - minimum vertex cover

Basics of Approximation Algorithms

Recap: computational classes **P** versus **NP**



P polynomial-time solvable

NP non-deterministic polynomial-time solvable

NP-hard "at least as hard as the hardest problems in **NP**"

Background

Many **NP**-hard optimization problems:

- minimum vertex cover, minimum dominate set
- maximum independent set, maximum clique, maximum cut
- ...

Background

Many **NP**-hard optimization problems:

- minimum vertex cover, minimum dominate set
- maximum independent set, maximum clique, maximum cut
- ...

No efficient (i.e., polynomial-time) solution is believed to exist

A (trivial) theorem

Assuming $\mathbf{P} \neq \mathbf{NP}$, there is no polynomial time algorithm for *exactly* solving any **NP**-hard problem

Background

I can't find an efficient algorithm, but neither can all these famous people.

Source for comic: Computers and Intractability by Garey and Johnson.

One powerful way to deal with **NP**-hard problems:

Background

I can't find an efficient algorithm, but neither can all these famous people.

Source for comic: Computers and Intractability by Garey and Johnson.

One powerful way to deal with **NP**-hard problems:

- **Approximation Algorithms**

Approximation algorithms

Idea: If finding an optimal solution is hard, perhaps we could be satisfied with something close to it?

Approximation algorithms

Idea: If finding an optimal solution is hard, perhaps we could be satisfied with something close to it?

$\boxed{\text{We trade accuracy for time!}}$

Approximation algorithms

Idea: If finding an optimal solution is hard, perhaps we could be satisfied with something close to it?

We trade accuracy for time!

Approximation algorithms

- in polynomial time
- find a solution that is close to the optimal solution
- Obstacle: need to prove a solution's value is close to optimum value, without even knowing what the optimum value is!

Approximation algorithms

Idea: If finding an optimal solution is hard, perhaps we could be satisfied with something close to it?

We trade accuracy for time!

Approximation algorithms

- in polynomial time
- find a solution that is close to the optimal solution
- Obstacle: need to prove a solution's value is close to optimum value, without even knowing what the optimum value is!

This lecture: focus on **NP**-hard optimization problems

Basic definitions

Minimization problem: For $\rho(n) \geq 1$, an algorithm $\mathcal{A}$ is called a $\rho(n)$-approximation algorithm if, for any instance I,

- $\mathcal{A}$ runs in polynomial-time in the input size n, and
- $\mathcal{A}$ computes a solution with objective function value

$$\mathrm{OPT}(I) \leq \mathcal{A}(I) \leq \rho(n) \cdot \mathrm{OPT}(I).$$

Basic definitions

Minimization problem: For $\rho(n) \geq 1$, an algorithm $\mathcal{A}$ is called a $\rho(n)$-approximation algorithm if, for any instance I,

- $\mathcal{A}$ runs in polynomial-time in the input size n, and
- $\mathcal{A}$ computes a solution with objective function value

$$\mathrm{OPT}(I) \leq \mathcal{A}(I) \leq \rho(n) \cdot \mathrm{OPT}(I).$$

– $\rho(n)$ is the approximation ratio or performance guarantee,

Basic definitions

Minimization problem: For $\rho(n) \geq 1$, an algorithm $\mathcal{A}$ is called a $\rho(n)$-approximation algorithm if, for any instance I,

- $\mathcal{A}$ runs in polynomial-time in the input size n, and
- $\mathcal{A}$ computes a solution with objective function value

$$\mathrm{OPT}(I) \leq \mathcal{A}(I) \leq \rho(n) \cdot \mathrm{OPT}(I).$$

- $\rho(n)$ is the approximation ratio or performance guarantee,
- $\rho(n)$ can depend on the input size n or be constant,

Basic definitions

Minimization problem: For $\rho(n) \geq 1$, an algorithm $\mathcal{A}$ is called a $\rho(n)$-approximation algorithm if, for any instance I,

- $\mathcal{A}$ runs in polynomial-time in the input size n, and
- $\mathcal{A}$ computes a solution with objective function value

$$\mathrm{OPT}(I) \leq \mathcal{A}(I) \leq \rho(n) \cdot \mathrm{OPT}(I).$$

- $\rho(n)$ is the approximation ratio or performance guarantee,
- $\rho(n)$ can depend on the input size n or be constant,
- 1-approximation algorithms are exact.

Basic definitions II

Maximization problem: For $\rho(n) \leq 1$, an algorithm $\mathcal{A}$ is called a $\rho(n)$-approximation algorithm if, for any instance I,

- $\mathcal{A}$ runs in polynomial-time in the input size n, and
- $\mathcal{A}$ computes a solution with objective function value

$$\mathrm{OPT}(I) \geq \mathcal{A}(I) \geq \rho(n) \cdot \mathrm{OPT}(I).$$

Basic definitions II

Maximization problem: For $\rho(n) \leq 1$, an algorithm $\mathcal{A}$ is called a $\rho(n)$-approximation algorithm if, for any instance I,

- $\mathcal{A}$ runs in polynomial-time in the input size n, and
- $\mathcal{A}$ computes a solution with objective function value

$$\mathrm{OPT}(I) \geq \mathcal{A}(I) \geq \rho(n) \cdot \mathrm{OPT}(I).$$

– $\rho(n)$ is the approximation ratio or performance guarantee,

Basic definitions II

Maximization problem: For $\rho(n) \leq 1$, an algorithm $\mathcal{A}$ is called a $\rho(n)$-approximation algorithm if, for any instance I,

- $\mathcal{A}$ runs in polynomial-time in the input size n, and
- $\mathcal{A}$ computes a solution with objective function value

$$\mathrm{OPT}(I) \geq \mathcal{A}(I) \geq \rho(n) \cdot \mathrm{OPT}(I).$$

– $\rho(n)$ is the approximation ratio or performance guarantee,

– $\rho(n)$ can depend on the input size n or be constant,

Basic definitions II

Maximization problem: For $\rho(n) \leq 1$, an algorithm $\mathcal{A}$ is called a $\rho(n)$-approximation algorithm if, for any instance I,

- $\mathcal{A}$ runs in polynomial-time in the input size n, and
- $\mathcal{A}$ computes a solution with objective function value

$$\mathrm{OPT}(I) \geq \mathcal{A}(I) \geq \rho(n) \cdot \mathrm{OPT}(I).$$

- $\rho(n)$ is the approximation ratio or performance guarantee,
- $\rho(n)$ can depend on the input size n or be constant,
- 1-approximation algorithms are exact.

Basic definitions II

Maximization problem: For $\rho(n) \leq 1$, an algorithm $\mathcal{A}$ is called a $\rho(n)$-approximation algorithm if, for any instance I,

- $\mathcal{A}$ runs in polynomial-time in the input size n, and
- $\mathcal{A}$ computes a solution with objective function value

$$\mathrm{OPT}(I) \geq \mathcal{A}(I) \geq \rho(n) \cdot \mathrm{OPT}(I).$$

- $\rho(n)$ is the approximation ratio or performance guarantee,
- $\rho(n)$ can depend on the input size n or be constant,
- 1-approximation algorithms are exact.

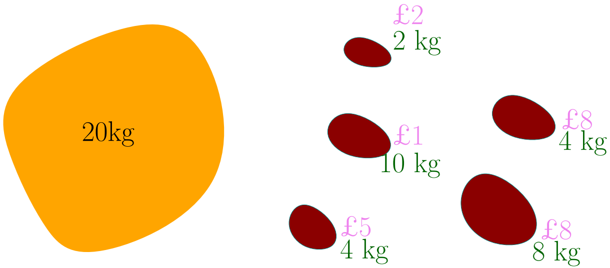
- Remark: Sometimes, people use $\frac{1}{\rho(n)}$ (≥ 1) as the approximation ratio for a maximization problem.

Knapsack problem: a $\frac{1}{2}$-approximation algorithm

Example: 0-1-knapsack problem

Input: n items $\{1, \ldots, n\}$ with weights w_j, values v_j, and a knapsack of capacity C.

Objective: to find items that fit together into the knapsack and maximize the value.



Some known facts

- Knapsack is **NP**-hard

Exact algorithms:

- $O(nC)$ time: dynamic programming
- $O(2^n)$ time: enumeration algorithm; branch-and-bound

A simple greedy algorithm GreedyKs

Idea keep picking whichever item that has the best value-per-weight until nothing else fits

A simple greedy algorithm GreedyKs

Idea keep picking whichever item that has the best value-per-weight until nothing else fits

GreedyKs:

1. sort the items by non-increasing value-to-weight ratio $\frac{v_j}{w_j}$ such that $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$;

A simple greedy algorithm GreedyKs

Idea keep picking whichever item that has the best value-per-weight until nothing else fits

GreedyKs:

1. sort the items by non-increasing value-to-weight ratio $\frac{v_j}{w_j}$ such that $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$;
2. initialize $K = \varnothing$, $w = 0$ and $v = 0$;

A simple greedy algorithm GreedyKs

Idea keep picking whichever item that has the best value-per-weight until nothing else fits

GreedyKs:

1. sort the items by non-increasing value-to-weight ratio $\frac{v_j}{w_j}$ such that $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$;
2. initialize $K = \varnothing$, $w = 0$ and $v = 0$;
3. for $j = 1, \ldots, n$:

A simple greedy algorithm GreedyKs

Idea keep picking whichever item that has the best value-per-weight until nothing else fits

GreedyKs:

1. sort the items by non-increasing value-to-weight ratio $\frac{v_j}{w_j}$ such that $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$;
2. initialize $K = \varnothing$, $w = 0$ and $v = 0$;
3. for $j = 1, \ldots, n$:
   - if $w + w_j \leq C$:
     set $K = K \cup \{j\}$, $w = w + w_j$ and $v = v + v_j$;

A simple greedy algorithm GreedyKs

Idea keep picking whichever item that has the best value-per-weight until nothing else fits

GreedyKs:

1. sort the items by non-increasing value-to-weight ratio $\frac{v_j}{w_j}$ such that $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$;
2. initialize $K = \varnothing$, $w = 0$ and $v = 0$;
3. for $j = 1, \ldots, n$:
    - if $w + w_j \leq C$:
      set $K = K \cup \{j\}$, $w = w + w_j$ and $v = v + v_j$;
4. output K, w and v.

GreedyKs is NOT always good

Suppose that $n = 2$, the capacity of the knapsack is $C = M \gg 0$.

item j	1	2
value v_j	2	M
weight w_j	1	M

GreedyKs is NOT always good

Suppose that $n = 2$, the capacity of the knapsack is $C = M \gg 0$.

item j	1	2
value v_j	2	M
weight w_j	1	M

- The algorithm GreedyKs yields $K = \{1\}$ with value $v = 2$.

$\textsc{GreedyKs}$ is NOT always good

Suppose that $n = 2$, the capacity of the knapsack is $C = M \gg 0$.

item j	1	2
value v_j	2	M
weight w_j	1	M

- The algorithm $\textsc{GreedyKs}$ yields $K = \{1\}$ with value $v = 2$.
- The optimal solution is $K^* = \{2\}$ with value $v^* = M$.

GreedyKs is NOT always good

Suppose that $n = 2$, the capacity of the knapsack is $C = M \gg 0$.

item j	1	2
value v_j	2	M
weight w_j	1	M

- The algorithm GreedyKs yields $K = \{1\}$ with value $v = 2$.
- The optimal solution is $K^* = \{2\}$ with value $v^* = M$.
- The approximation ratio is $\frac{2}{M}$, which can get arbitrarily bad.

An extended greedy algorithm ExtGreedyKs

Idea Modify the previous algorithm:
either take the solution produced by GreedyKs, or take the item with highest value, whichever is better

An extended greedy algorithm ExtGreedyKs

Idea Modify the previous algorithm:
either take the solution produced by GreedyKs, or take the item with highest value, whichever is better

(We can assume that the item i with highest value has weight $w_i \leq C$; as otherwise, we can remove this item without affecting the solution)

An extended greedy algorithm ExtGreedyKs

Idea Modify the previous algorithm:
either take the solution produced by GreedyKs, or take the item with highest value, whichever is better

(We can assume that the item i with highest value has weight $w_i \leq C$; as otherwise, we can remove this item without affecting the solution)

ExtGreedyKs:

1. run GreedyKs, and let z^G be the outputted total value.

An extended greedy algorithm ExtGreedyKs

Idea Modify the previous algorithm:
either take the solution produced by GreedyKs, or take the item with highest value, whichever is better

(We can assume that the item i with highest value has weight $w_i \leq C$; as otherwise, we can remove this item without affecting the solution)

ExtGreedyKs:

1. run GreedyKs, and let z^G be the outputted total value.
2. let

$$z^{EG} = \max\{z^G, v_{\max}\},$$

where $v_{\max} := \max_{j=1,\ldots,n} v_j$

An extended greedy algorithm ExtGreedyKs

Idea Modify the previous algorithm:
either take the solution produced by GreedyKs, or take the item with highest value, whichever is better

(We can assume that the item i with highest value has weight $w_i \leq C$; as otherwise, we can remove this item without affecting the solution)

ExtGreedyKs:

1. run GreedyKs, and let z^G be the outputted total value.
2. let
$$z^{EG} = \max\{z^G, v_{\max}\},$$
where $v_{\max} := \max_{j=1,\ldots,n} v_j$
3. output z^{EG}

Analysis of $\textsc{ExtGreedyKs}$

Theorem

The algorithm $\textsc{ExtGreedyKs}$ *is a* $\frac{1}{2}$*-approximation algorithm for the* $0-1$*-knapsack problem.*

Analysis of ExtGreedyKs

Theorem

The algorithm ExtGreedyKs *is a* $\frac{1}{2}$*-approximation algorithm for the* $0-1$*-knapsack problem.*

Proof.

Running time: sorting and other steps all run in polynomial time

Analysis of $\textsc{ExtGreedyKs}$

Theorem

The algorithm $\textsc{ExtGreedyKs}$ *is a* $\frac{1}{2}$*-approximation algorithm for the* $0-1$*-knapsack problem.*

Proof.

Running time: sorting and other steps all run in polynomial time
Correctness:

- Let k be the index of the first item *skipped* by $\textsc{GreedyKs}$.

Analysis of ExtGreedyKs

Theorem

The algorithm ExtGreedyKs *is a* $\frac{1}{2}$*-approximation algorithm for the* $0-1$*-knapsack problem.*

Proof.

Running time: sorting and other steps all run in polynomial time
Correctness:

- Let k be the index of the first item *skipped* by GreedyKs.
- Note, that for divisible goods, the value of an optimal *fractional* solution is

$$z^{FRAC} = v_1 + v_2 + \ldots + v_{k-1} + \frac{W - \sum_{i=1}^{k-1} w_i}{w_k} v_k.$$

Analysis of $\textsc{ExtGreedyKs}$

Theorem

The algorithm $\textsc{ExtGreedyKs}$ *is a* $\frac{1}{2}$*-approximation algorithm for the* $0-1$*-knapsack problem.*

Proof.

Running time: sorting and other steps all run in polynomial time

Correctness:

- Let k be the index of the first item *skipped* by $\textsc{GreedyKs}$.
- Note, that for divisible goods, the value of an optimal *fractional* solution is

$$z^{FRAC} = v_1 + v_2 + \ldots + v_{k-1} + \frac{W - \sum_{i=1}^{k-1} w_i}{w_k} v_k.$$

- Therefore,

$$2 \cdot z^{EG} \geq \underbrace{v_1 + v_2 + \ldots + v_{k-1}}_{\leq\ z^G\ \leq\ z^{EG}} + \underbrace{v_k}_{\leq\ z^{EG}} \geq z^{FRAC} \geq OPT.$$

Knapsack problem: a $(1 - \varepsilon)$-approximation algorithm

A dynamic programming algorithm

Definition: $\mathrm{OPT}(i, v) =$ min weight of a knapsack for which we can obtain a solution of value $\geq v$ using a subset of items $1, \ldots, i$.

Note: Optimal value is the largest value v such that $\mathrm{OPT}(n, v) \leq C$.

- Case 1: OPT does not select item i
  - OPT selects best of $1, \ldots, i\text{–}1$ that achieves value $\geq v$.
- Case 2: OPT selects item i
  - the item i consumes weight w_i, and value v_i
  - OPT selects best of $1, \ldots, i\text{–}1$ that achieves value $\geq v - v_i$

$$\mathrm{OPT}(i, v) = \begin{cases} 0 & \text{if } v \leq 0 \\ \infty & \text{if } i = 0 \text{ and } v > 0 \\ \min\{\mathrm{OPT}(i-1, v), w_i + \mathrm{OPT}(i-1, v - v_i)\} & \text{otherwise} \end{cases}$$

DynamicKs

DynamicKs

1. solve the previous recurrence

Note:

- DynamicKs finds the optimal solution
- can be implemented using the array

Implementation of DynamicKs

DynamicKs: Use an $(n+1)(n \cdot v_{\max}+1)$-dimensional array.
Input: $n, C, w_1, \ldots, w_n, v_1, \ldots, v_n$ (Here $v_{\max} = \max_{i=1,\ldots,n} v_i$)

```
for j = 0 to n · v_max
    M[0, j] = ∞

for i = 1 to n
    for v = 0 to n · v_max
        if v < v_i
            M[i, v] = min{M[i − 1, v], w_i}
        else
            M[i, v] = min{M[i − 1, v], w_i + M[i − 1, v − v_i]}
```

see next slides to continue

Implementation of DynamicKs - cont.

continue from the previous slide:

```
val ← 0
for v = n · v_max to 0
    if M[n, v] ≤ C
        val ← v
        BREAK

i ← n; c ← C; A ← ∅; v ← val
while i > 0 and v > 0
    if M[i, v] ≠ M[i − 1, v]
        A ← A ∪ {i}
        v ← v − v_i
    i ← i − 1
return A, val
```

Further remarks on DynamicKs

- run-time: $O(n^2 v_{\max})$, where $v_{\max}$ is the maximum of any value (reason: the optimal value can be $nv_{\max}$)

  (Note: the above assuming the values of items are integers)

- Not polynomial in input size
  (reason: $v_{\max}$ is exponential in the input length, since numbers are encoded in binary);
- but polynomial if the values are small integers

Towards an approximation algorithm

Intuition for obtaining an approximation algorithm

- turn the original instance into one with smaller range of values
- then solve the resulting "smaller" instance

Example

- original instance: $C \leq 9$; with a wide range of v values

j	1	2	3	4
v_j	23645	1524	256711	694760
w_j	2	4	1	4

- rounded instance: $C \leq 9$; a narrower range of $\hat{v}$ values

j	1	2	3	4
$\hat{v}_j$	31	2	336	914
w_j	2	4	1	4

An approximation algorithm DynamicAppKs

Given precision parameter $\varepsilon \in (0, 1)$ and $v_{\max}$ maximum value of the original instance, round the v values:

- θ scaling factor $= \varepsilon v_{\max}/2n$
- $\bar{v}_i = \lceil \frac{v_i}{\theta} \rceil \theta$, $\hat{v}_i = \lceil \frac{v_i}{\theta} \rceil$

DynamicAppKs

1. Round all the v values to $\hat{v}$ values as specified above
2. DynamicKs on the rounded instance
3. return optimal items in rounded instance

Note:

- optimal solutions to problem with $\bar{v}$ are equivalent to optimal solutions to problem with $\hat{v}$
- $\bar{v}$ close to v so optimal solution using $\bar{v}$ is nearly optimal
- $\hat{v}$ small and integral so DynamicKs on problem with $\hat{v}$ is fast.

The performance guarantee of $\textsc{DynamicAppKs}$

Theorem: For any $0 < \varepsilon < 1$, the algorithm $\textsc{DynamicAppKs}$ is a $(1 - \varepsilon)$-approximation for the $0 - 1$-knapsack problem with run-time $O(n^3/\varepsilon)$.

The performance guarantee of $\textsc{DynamicAppKs}$

Theorem: For any $0 < \varepsilon < 1$, the algorithm $\textsc{DynamicAppKs}$ is a $(1-\varepsilon)$-approximation for the $0-1$-knapsack problem with run-time $O(n^3/\varepsilon)$.

Proof:

Running time: dynamic programming $\textsc{DynamicKs}$ runs in $O(n^2 \hat{v}_{\max})$ time, where
$\hat{v}_{\max} = \lceil \frac{v_{\max}}{\theta} \rceil = \lceil \frac{2n}{\varepsilon} \rceil$

The performance guarantee of $\textsc{DynamicAppKs}$

Theorem: For any $0 < \varepsilon < 1$, the algorithm $\textsc{DynamicAppKs}$ is a $(1-\varepsilon)$-approximation for the $0-1$-knapsack problem with run-time $O(n^3/\varepsilon)$.

Proof:

Running time: dynamic programming $\textsc{DynamicKs}$ runs in $O(n^2\hat{v}_{\max})$ time, where $\hat{v}_{\max} = \lceil \frac{v_{\max}}{\theta} \rceil = \lceil \frac{2n}{\varepsilon} \rceil$

Correctness: Let S^* be any feasible solution; let S be the solution found by rounding algorithm. Then we will show that

$$\sum_{i \in S} v_i \geq \frac{1}{1+\varepsilon} \sum_{i \in S^*} v_i \geq (1-\varepsilon) \sum_{i \in S^*} v_i$$

Note: once we have proven the above, then we can take S^* to be the optimal solution. Thus the total value of the solution found by the rounding algorithm is at least $(1-\varepsilon) \cdot \mathrm{OPT}$, where OPT is the total value of the optimal solution.

The performance guarantee of DynamicAppKs

Now we prove: $\sum_{i \in S} v_i \geq \frac{1}{1+\varepsilon} \sum_{i \in S^*} v_i$.
It holds that:

$$
\begin{aligned}
\sum_{i \in S^*} v_i &\leq \sum_{i \in S^*} \bar{v}_i && \text{always round up} \\
&\leq \sum_{i \in S} \bar{v}_i && \text{solve rounded instance optimally} \\
&\leq \sum_{i \in S} (v_i + \theta) && \text{never round up by more than } \theta \\
&\leq \sum_{i \in S} v_i + n\theta && |S| \leq n \\
&= \sum_{i \in S} v_i + \frac{1}{2}\varepsilon v_{\max} && \theta = \varepsilon v_{\max}/2n \\
&\leq (1+\varepsilon) \sum_{i \in S} v_i && v_{\max} \leq 2 \sum_{i \in S} v_i
\end{aligned}
$$

In the last inequality, why does it hold that $v_{\max} \leq 2\sum_{i\in S} v_i$?

We can take S^* to be the subset containing only the item of largest value. Then

$$
\begin{aligned}
v_{\max} &\leq \sum_{i\in S} v_i + \frac{1}{2}\varepsilon v_{\max} \\
&\leq \sum_{i\in S} v_i + \frac{1}{2} v_{\max}
\end{aligned}
$$

This gives:

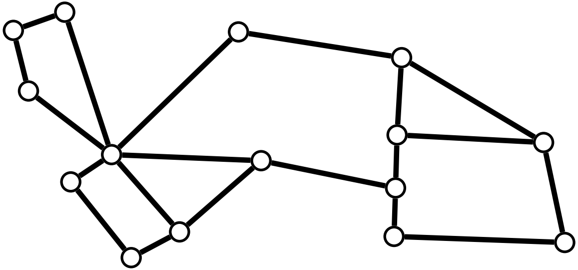
$$
v_{\max} \leq 2\sum_{i\in S} v_i
$$

Minimum Vertex Cover Problem I

Example: minimum vertex cover problem

Vertex cover of a graph: a subset C of its vertices such that for each edge $\{u, v\}$ at least one endpoint u or v is in C

> minimum vertex cover problem
>
> Input: an undirected graph $G = (V, E)$
> Objective: find a vertex cover of G of smallest possible size.

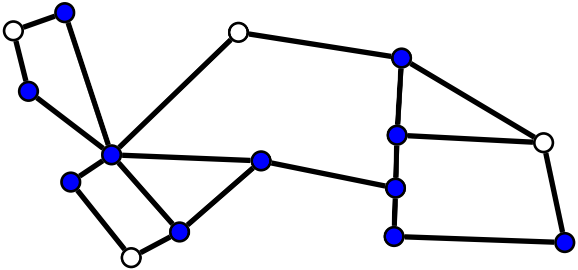


Example: minimum vertex cover problem

Vertex cover of a graph: a subset C of its vertices such that for each edge $\{u, v\}$ at least one endpoint u or v is in C

minimum vertex cover problem

Input: an undirected graph $G = (V, E)$
Objective: find a vertex cover of G of smallest possible size.

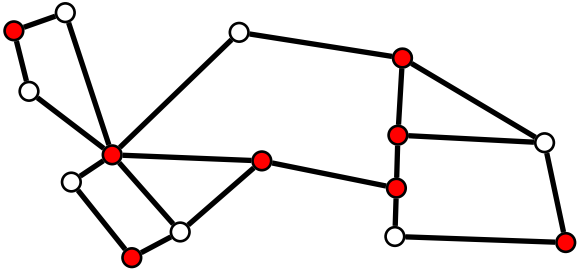


A vertex cover of size 11

Example: minimum vertex cover problem

Vertex cover of a graph: a subset C of its vertices such that for each edge $\{u, v\}$ at least one endpoint u or v is in C

minimum vertex cover problem
Input: an undirected graph $G = (V, E)$ Objective: find a vertex cover of G of smallest possible size.



A vertex cover of size 8

Some known facts

- Minimum vertex cover problem is **NP**-hard

Exact algorithms:

- $O(c^n)$ time, for some constant $1 < c \leq 2$: branch-and-bound

A simple greedy algorithm GreedyVC

Idea keep adding endpoints of edges that have no shared endpoints with previous edges.

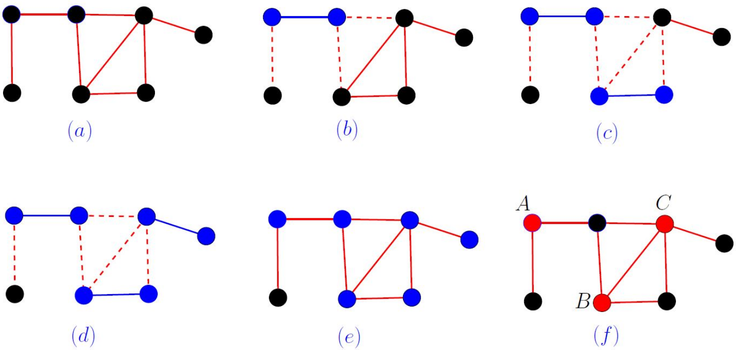
A simple greedy algorithm GreedyVC

Idea keep adding endpoints of edges that have no shared endpoints with previous edges.

GreedyVC:

1. $C \leftarrow \varnothing$
2. repeat until all edges deleted:
   - i pick any edge $\{u, v\}$
   - ii add u and v to C
   - iii delete all edges incident to u or v
3. return C

Algorithm GreedyVC on an example



The returned set C: blue vertices. $|C| = 6$

The optimum set C^*: red vertices. $|C^*| = 3$

Analysis of GreedyVC

Theorem

GreedyVC *is a 2-approximation algorithm for the vertex cover problem.*

Analysis of GreedyVC

Theorem

GreedyVC *is a 2-approximation algorithm for the vertex cover problem.*

Proof:

Runs in polynomial time

- In each iteration at least one edge is deleted $\Rightarrow$ time $\mathcal{O}(m)$.

Analysis of GreedyVC

Theorem

GreedyVC *is a 2-approximation algorithm for the vertex cover problem.*

Proof:

Runs in polynomial time

- In each iteration at least one edge is deleted $\Rightarrow$ time $\mathcal{O}(m)$.

Correctness

- The set C is a vertex cover by termination condition of loop.

Analysis of $\textsc{GreedyVC}$

Theorem

$\textsc{GreedyVC}$ *is a* 2*-approximation algorithm for the vertex cover problem.*

Proof:

Runs in polynomial time

- In each iteration at least one edge is deleted $\Rightarrow$ time $\mathcal{O}(m)$.

Correctness

- The set C is a vertex cover by termination condition of loop.
- Let A be the set of edges picked by the algorithm.

Analysis of GreedyVC

Theorem

GreedyVC *is a 2-approximation algorithm for the vertex cover problem.*

Proof:

Runs in polynomial time

- In each iteration at least one edge is deleted $\Rightarrow$ time $\mathcal{O}(m)$.

Correctness

- The set C is a vertex cover by termination condition of loop.
- Let A be the set of edges picked by the algorithm.
- Edges in A do not share any endpoints, and any vertex cover has to contain at least one endpoint of each, so $OPT \geq |A|$.

Analysis of GreedyVC

Theorem

GreedyVC *is a 2-approximation algorithm for the vertex cover problem.*

Proof:

Runs in polynomial time

- In each iteration at least one edge is deleted $\Rightarrow$ time $\mathcal{O}(m)$.

Correctness

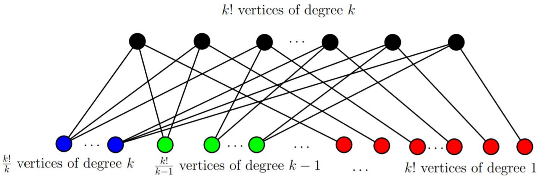
- The set C is a vertex cover by termination condition of loop.
- Let A be the set of edges picked by the algorithm.
- Edges in A do not share any endpoints, and any vertex cover has to contain at least one endpoint of each, so $OPT \geq |A|$.
- Therefore, $|C| = 2 \cdot |A| \leq 2 \cdot |OPT|$.

Another greedy algorithm?

ANOTHERGREEDYVC:

1. $C \leftarrow \varnothing$
2. repeat until all edges deleted:
   i pick a vertex $v \in V$ with maximum degree
   ii add v to C
   iii delete all edges incident v
3. return C

- The performance of the above is bad, i.e,. it has large approximation ratio! Exercise!



Minimum Vertex Cover Problem II

Minimum-weight vertex cover problem

Minimum-weight vertex cover problem:
Input: Undirected graph $G = (V, E)$ and weights $w_v \in \mathbb{R}_+$ for all vertices $v \in V$.

Goal: Find a minimum-weight vertex cover, i.e., a subset $C \subseteq V$ such that, for each edge $\{u, v\} \in E$, at least one endpoint u or v is in subset C and $w(C) = \sum_{v \in C} w_v$ is minimal.

Minimum-weight vertex cover problem

Minimum-weight vertex cover problem:
Input: Undirected graph $G = (V, E)$ and weights $w_v \in \mathbb{R}_+$ for all vertices $v \in V$.

Goal: Find a minimum-weight vertex cover, i.e., a subset $C \subseteq V$ such that, for each edge $\{u, v\} \in E$, at least one endpoint u or v is in subset C and $w(C) = \sum_{v \in C} w_v$ is minimal.

Note:

- a generalization of the unweighted version, in which $w_v = 1$ for all $v \in V$

Minimum-weight vertex cover problem

Minimum-weight vertex cover problem:

Input: Undirected graph $G = (V, E)$ and weights $w_v \in \mathbb{R}_+$ for all vertices $v \in V$.

Goal: Find a minimum-weight vertex cover, i.e., a subset $C \subseteq V$ such that, for each edge $\{u, v\} \in E$, at least one endpoint u or v is in subset C and $w(C) = \sum_{v \in C} w_v$ is minimal.

Note:

- a generalization of the unweighted version, in which $w_v = 1$ for all $v \in V$
- the previous algorithm $\textsc{GreedyVC}$ fails

Minimum-weight vertex cover problem

Minimum-weight vertex cover problem:
Input: Undirected graph $G = (V, E)$ and weights $w_v \in \mathbb{R}_+$ for all vertices $v \in V$.

Goal: Find a minimum-weight vertex cover, i.e., a subset $C \subseteq V$ such that, for each edge $\{u, v\} \in E$, at least one endpoint u or v is in subset C and $w(C) = \sum_{v \in C} w_v$ is minimal.

Note:

- a generalization of the unweighted version, in which $w_v = 1$ for all $v \in V$
- the previous algorithm $\textsc{GreedyVC}$ fails
- now we use the linear programming to design an approximation algorithm

The linear programming based algorithm

Algorithm LP-VC:

The linear programming based algorithm

Algorithm $\mathrm{LP\text{-}VC}$:

1. Write down the corresponding Integer Linear Program (ILP):

$$
\begin{aligned}
\min \quad & \sum_{v \in V} w_v x_v \\
\text{s.t.} \quad & x_u + x_v \geq 1 \quad \forall \{u, v\} \in E \\
& x_v \in \{0, 1\} \qquad \forall\, v \in V
\end{aligned}
$$

The linear programming based algorithm

Algorithm $\mathrm{LP\text{-}VC}$:

1. Write down the corresponding Integer Linear Program (ILP):

$$
\begin{aligned}
\min \quad & \sum_{v \in V} w_v x_v \\
\text{s.t.} \quad & x_u + x_v \geq 1 \quad \forall \{u, v\} \in E \\
& x_v \in \{0, 1\} \qquad \forall\, v \in V
\end{aligned}
$$

- (Remark: ILP is NP-hard, so we do not directly solve it)

The linear programming based algorithm

Algorithm LP-VC:

1. Write down the corresponding Integer Linear Program (ILP):

$$
\begin{aligned}
\min \quad & \sum_{v \in V} w_v x_v \\
\text{s.t.} \quad & x_u + x_v \geq 1 \quad \forall \{u, v\} \in E \\
& x_v \in \{0, 1\} \qquad \forall\, v \in V
\end{aligned}
$$

- (Remark: ILP is NP-hard, so we do not directly solve it)

2. Compute an optimal solution to the Linear Program (LP) relaxation:

$$
\begin{aligned}
\min \quad & \sum_{v \in V} w_v \overline{x}_v \\
\text{s.t.} \quad & \overline{x}_u + \overline{x}_v \geq 1 \quad \forall \{u, v\} \in E \\
& 0 \leq \overline{x}_v \leq 1 \qquad \forall\, v \in V
\end{aligned}
$$

The linear programming based algorithm

Algorithm LP-VC:

1. Write down the corresponding Integer Linear Program (ILP):

$$
\begin{aligned}
\min \quad & \sum_{v \in V} w_v x_v \\
\text{s.t.} \quad & x_u + x_v \geq 1 \quad \forall \{u, v\} \in E \\
& x_v \in \{0, 1\} \qquad \forall\, v \in V
\end{aligned}
$$

- (Remark: ILP is NP-hard, so we do not directly solve it)

2. Compute an optimal solution to the Linear Program (LP) relaxation:

$$
\begin{aligned}
\min \quad & \sum_{v \in V} w_v \overline{x}_v \\
\text{s.t.} \quad & \overline{x}_u + \overline{x}_v \geq 1 \quad \forall \{u, v\} \in E \\
& 0 \leq \overline{x}_v \leq 1 \qquad \forall\, v \in V
\end{aligned}
$$

3. Round the fractional solution, i.e., set

$$
x_v = \begin{cases} 1 & \text{if } \overline{x}_v \geq \frac{1}{2}, \\ 0 & \text{if } \overline{x}_v < \frac{1}{2}. \end{cases}
$$

The performance guarantee

Theorem The above algorithm $\mathrm{LP\text{-}VC}$ is a 2-approximation algorithm for the minimum-weight vertex cover problem.

Proof

- The procedure runs in polynomial time, since linear programs can be solved in polynomial time, e.g., by interior point methods.

The performance guarantee

Theorem The above algorithm $\mathrm{LP\text{-}VC}$ is a 2-approximation algorithm for the minimum-weight vertex cover problem.

Proof

- The procedure runs in polynomial time, since linear programs can be solved in polynomial time, e.g., by interior point methods.
- Let $C^* \subseteq V$ be a minimum-weight vertex cover with objective function value z^* and $\overline{x} \in [0,1]^{|V|}$ be an optimal solution for the LP with objective function value $\overline{z}$.

The performance guarantee

Theorem The above algorithm $\mathrm{LP\text{-}VC}$ is a 2-approximation algorithm for the minimum-weight vertex cover problem.

Proof

- The procedure runs in polynomial time, since linear programs can be solved in polynomial time, e.g., by interior point methods.
- Let $C^* \subseteq V$ be a minimum-weight vertex cover with objective function value z^* and $\overline{x} \in [0,1]^{|V|}$ be an optimal solution for the LP with objective function value $\overline{z}$.
- $\overline{z} \leq z^*$ since the LP relaxation gives a lower bound.

The performance guarantee – cont.

Proof cont.

- Rounding produces a vertex cover $C \subseteq V$:

The performance guarantee – cont.

Proof cont.

- Rounding produces a vertex cover $C \subseteq V$:
  - since, for each edge $\{u, v\} \in E$, $\overline{x}_u + \overline{x}_v \geq 1$, at least one $\overline{x}_u$ or $\overline{x}_v$ is larger or equal than $\frac{1}{2}$,

The performance guarantee – cont.

Proof cont.

- Rounding produces a vertex cover $C \subseteq V$:
  - since, for each edge $\{u, v\} \in E$, $\overline{x}_u + \overline{x}_v \geq 1$, at least one $\overline{x}_u$ or $\overline{x}_v$ is larger or equal than $\frac{1}{2}$,
  - for the rounded solution $x \in \{0, 1\}^{|V|}$, at least one x_u or x_v
    is equal to 1,

The performance guarantee – cont.

Proof cont.

- Rounding produces a vertex cover $C \subseteq V$:
  - since, for each edge $\{u, v\} \in E$, $\overline{x}_u + \overline{x}_v \geq 1$, at least one $\overline{x}_u$ or $\overline{x}_v$ is larger or equal than $\frac{1}{2}$,
  - for the rounded solution $x \in \{0, 1\}^{|V|}$, at least one x_u or x_v is equal to 1,
  - $C = \{v \in V : x_v = 1\}$ is a vertex cover.

The performance guarantee – cont.

Proof cont.

- Rounding produces a vertex cover $C \subseteq V$:
  - since, for each edge $\{u, v\} \in E$, $\overline{x}_u + \overline{x}_v \geq 1$, at least one $\overline{x}_u$ or $\overline{x}_v$ is larger or equal than $\frac{1}{2}$,
  - for the rounded solution $x \in \{0,1\}^{|V|}$, at least one x_u or x_v is equal to 1,
  - $C = \{v \in V : x_v = 1\}$ is a vertex cover.
- Considering the objective function value, it holds:

$$\begin{aligned}\overline{z} = \sum_{v \in V} w_v \overline{x}_v &\geq \sum_{v \in V \,:\, \overline{x}_v \geq \frac{1}{2}} w_v \, \overline{x}_v \\ &\geq \sum_{v \in V \,:\, \overline{x}_v \geq \frac{1}{2}} w_v \, \frac{1}{2} = \frac{1}{2} \sum_{v \in C} w_v = \frac{1}{2} w(C)\end{aligned}$$

The performance guarantee – cont.

Proof cont.

- Rounding produces a vertex cover $C \subseteq V$:
  - since, for each edge $\{u, v\} \in E$, $\overline{x}_u + \overline{x}_v \geq 1$, at least one $\overline{x}_u$ or $\overline{x}_v$ is larger or equal than $\frac{1}{2}$,
  - for the rounded solution $x \in \{0, 1\}^{|V|}$, at least one x_u or x_v is equal to 1,
  - $C = \{v \in V : x_v = 1\}$ is a vertex cover.
- Considering the objective function value, it holds:

$$\begin{aligned} \overline{z} = \sum_{v \in V} w_v \overline{x}_v &\geq \sum_{v \in V\,:\,\overline{x}_v \geq \frac{1}{2}} w_v\, \overline{x}_v \\ &\geq \sum_{v \in V\,:\,\overline{x}_v \geq \frac{1}{2}} w_v\, \frac{1}{2} = \frac{1}{2} \sum_{v \in C} w_v = \frac{1}{2} w(C) \end{aligned}$$

- Thus, $w(C) \leq 2\overline{z} \leq 2z^*$.