COMP 382: Reasoning about Algorithms

More NP-Complete Reductions and Approximation Algorithms

Prof. Maryam Aliakbarpour

co-instructors: Prof. Anjum Chida & Prof. Konstantinos Mamouras

December 2, 2025

Today's Lecture

1. Approximation Algorithms

1.1 2-Approximation Algorithm for Vertex Cover

1.2 (In)Approximability of TSP

1.3 Metric TSP and Its Approximation Algorithm

2. Set Cover

2.1 SET-COVER Is NP-Complete

2.2 Approximation Algorithm for Set Cover

Reading:

- Chapter 12 of the *Algorithms* book [Erickson, 2019]
- Chapter 8.1 of [Tardos and Kleinberg, 2005]

Content adapted from the same references.

Approximation Algorithms

If we relax optimality, does it get easier?

Approximation Algorithms

NP-complete problems are hard to solve *exactly*. A natural question is:

If we stop insisting on the optimal solution, can we solve the problem efficiently?

Approximation Algorithms

NP-complete problems are hard to solve *exactly*. A natural question is:

If we stop insisting on the optimal solution, can we solve the problem efficiently?

Approximation Algorithms:

- Run in polynomial time.
- Always produce a *feasible* solution.
- Guarantee that the solution is within a factor α of the optimum.

Approximation Algorithms

NP-complete problems are hard to solve *exactly*. A natural question is:

If we stop insisting on the optimal solution, can we solve the problem efficiently?

Approximation Algorithms:

- Run in polynomial time.
- Always produce a *feasible* solution.
- Guarantee that the solution is within a factor α of the optimum.
- For minimization problems $\alpha > 1$, and smaller α means better approximation.

$$\mathrm{OPT} \;\leq\; \mathrm{cost(ALG)} \;\leq\; \alpha \cdot \mathrm{OPT}.$$

- For maximization problems $\alpha < 1$, and larger α means better approximation.

$$\mathrm{OPT} \;\geq\; \mathrm{cost(ALG)} \;\geq\; \alpha \cdot \mathrm{OPT}.$$

Approximability of Problems

Often we seek a solution that is *close* to optimal: a constant-factor approximation, or an α that grows slowly with n (e.g., $\log n$).

A natural question:

Does every NP-hard optimization problem admit a polynomial-time approximation algorithm?

- Some problems *do* admit good approximations.
- Others remain hard even to approximate within any reasonable factor.

This leads to an entire area of study: **approximability theory**.

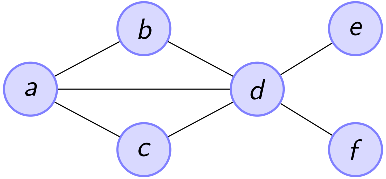
2-Approximation Algorithm for Vertex Cover

Recap: Vertex Covers

Let $G = (V, E)$ be a simple undirected graph.

A vertex cover in G is a subset $C \subseteq V$ such that every edge of G has at least one endpoint in C.

Equivalently: every edge is "touched" by C.

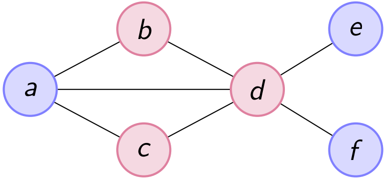


Recap: Vertex Covers

Let $G = (V, E)$ be a simple undirected graph.

A vertex cover in G is a subset $C \subseteq V$ such that every edge of G has at least one endpoint in C.

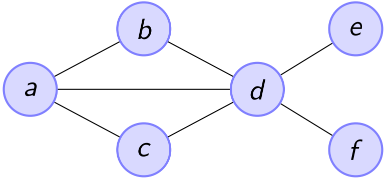
Equivalently: every edge is "touched" by C.



A vertex cover $C = \{b,\ c,\ d\}$ touches all edges.

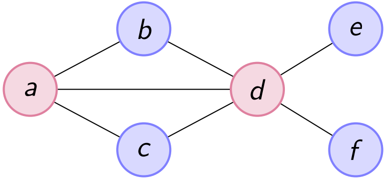
Minimum Vertex Cover Problem

VERTEX-COVER: Given a graph G and an integer k, does G contain a vertex cover of size at most k?



Minimum Vertex Cover Problem

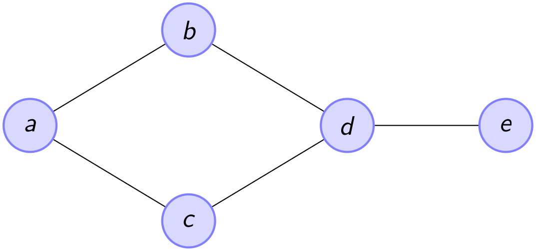
VERTEX-COVER: Given a graph G and an integer k, does G contain a vertex cover of size at most k?



Here $\{a,\ d\}$ is a vertex cover of size 2.

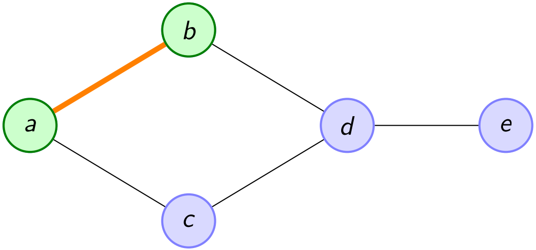
2-Approximation for Vertex Cover

The algorithm repeatedly picks an uncovered edge and adds *both* endpoints to the cover.



2-Approximation for Vertex Cover

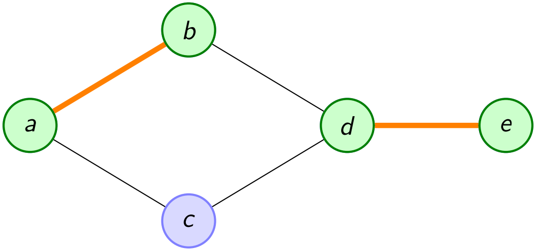
The algorithm repeatedly picks an uncovered edge and adds *both* endpoints to the cover.



Step 1: pick an uncovered edge (here (a, b)), add a and b.

2-Approximation for Vertex Cover

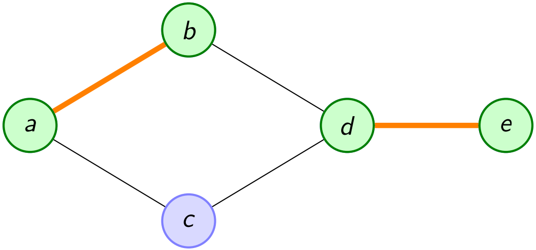
The algorithm repeatedly picks an uncovered edge and adds *both* endpoints to the cover.



Step 1: pick an uncovered edge (here (a, b)), add a and b.
Step 2: pick another uncovered edge (here (d, e)), add d and e.

2-Approximation for Vertex Cover

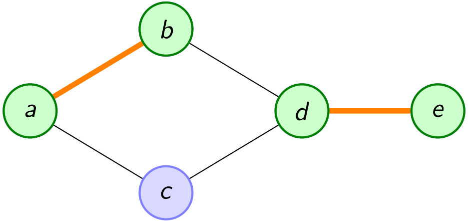
The algorithm repeatedly picks an uncovered edge and adds *both* endpoints to the cover.



Step 1: pick an uncovered edge (here (a, b)), add a and b.
Step 2: pick another uncovered edge (here (d, e)), add d and e.
All edges are covered. $\Rightarrow$ Final cover $= \{a, b, c, d\}$.

Why the Greedy Algorithm Is a 2-Approximation

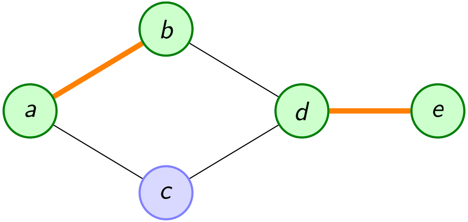
- Let M be the set of edges the algorithm picks. They are pairwise disjoint $\Rightarrow$ forming a matching.



Picked edges form a matching M

Why the Greedy Algorithm Is a 2-Approximation

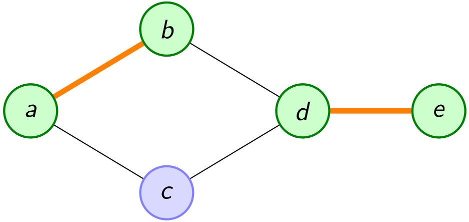
- Let M be the set of edges the algorithm picks. They are pairwise disjoint $\Rightarrow$ forming a matching.
- Any vertex cover must hit each edge of M. $\Rightarrow |C^*| \geq |M|$.



Picked edges form a matching M

Why the Greedy Algorithm Is a 2-Approximation

- Let M be the set of edges the algorithm picks. They are pairwise disjoint $\Rightarrow$ forming a matching.
- Any vertex cover must hit each edge of M. $\Rightarrow |C^*| \geq |M|$.
- The algorithm adds *both* endpoints of each edge in M. $\Rightarrow |C_{\text{ALG}}| = 2|M|$.

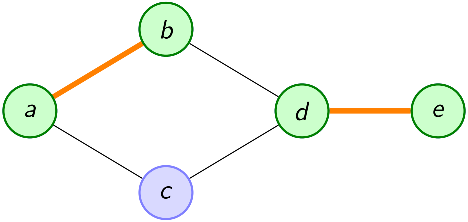


Picked edges form a matching M

Why the Greedy Algorithm Is a 2-Approximation

- Let M be the set of edges the algorithm picks. They are pairwise disjoint $\Rightarrow$ forming a matching.
- Any vertex cover must hit each edge of M. $\Rightarrow |C^*| \geq |M|$.
- The algorithm adds *both* endpoints of each edge in M. $\Rightarrow |C_{\mathrm{ALG}}| = 2|M|$.
- Therefore

$$|C_{\mathrm{ALG}}| = 2|M| \leq 2|C^*|.$$



Picked edges form a matching M

(In)Approximability of TSP

Approximating TSP

APPROX-TSP Given a complete graph G and a constant $\alpha > 1$, output a tour in G with cost at most $\alpha \cdot \mathrm{OPT}$.

Intuitively, this requirement is **weaker** than exact TSP.

- Any algorithm solving exact TSP automatically solves APPROX-TSP.

Approximating TSP

APPROX-TSP Given a complete graph G and a constant $\alpha > 1$, output a tour in G with cost at most $\alpha \cdot \mathrm{OPT}$.

Intuitively, this requirement is **weaker** than exact TSP.

- Any algorithm solving exact TSP automatically solves APPROX-TSP.

However, APPROX-TSP and TSP have essentially the same difficulty.

Inapproximability of TSP

Theorem

TSP admits *no* polynomial-time α-approximation algorithm for any constant $\alpha \geq 1$, unless $\mathsf{P} = \mathsf{NP}$.

Inapproximability of TSP

Theorem

TSP admits *no* polynomial-time α-approximation algorithm for any constant $\alpha \geq 1$, unless $\mathrm{P} = \mathrm{NP}$.

How do we prove this? We return to our standard reduction recipe:

Reduce a known NP-complete problem (Hamiltonian Cycle) to APPROX-TSP.

The Strategy: Distinguishing by Gap

Goal: Reduce Hamiltonian Cycle to α-Approx TSP.

To do this, we construct an instance where the optimal cost falls into two disjoint ranges separated by a factor of α.

The Strategy: Distinguishing by Gap

Goal: Reduce Hamiltonian Cycle to α-Approx TSP.

To do this, we construct an instance where the optimal cost falls into two disjoint ranges separated by a factor of α.

If an algorithm guarantees an α-approximation, it returns a tour of cost $C \leq \alpha \cdot \text{OPT}$. We need a gap such that:

- **Case Yes (Hamiltonian):** $\text{OPT} = n \implies C \leq \alpha n$.
- **Case No (Not Hamiltonian):** $\text{OPT} > \alpha n$.

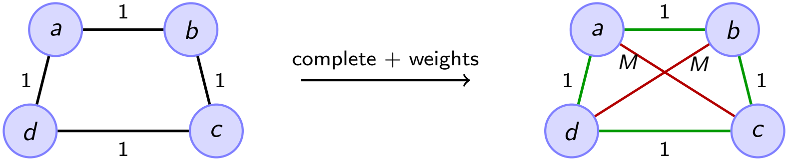
If we create this gap, checking if $C \leq \alpha n$ decides the Hamiltonian Cycle problem.

Creating the Gap

Construction: Build complete graph G' from G with weights:

$$w(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E(G) \quad \text{(Original)} \\ M & \text{if } (u,v) \notin E(G) \quad \text{(Non-edge)} \end{cases}$$

where M is a large number (chosen later).



Creating the Gap

The Costs:

- **If G is Hamiltonian:** We use n edges of weight 1.
  $\Rightarrow \mathrm{OPT}(G') = n$.

- **If G is NOT Hamiltonian:** We must use at least one weight M.
  $\Rightarrow \mathrm{OPT}(G') \geq M + (n-1)$.

Forcing the Gap: Set $M = \alpha n$.

$$\text{No Hamiltonian Cycle} \implies \mathrm{OPT}(G') \geq \alpha n + (n-1) > \alpha n.$$

The Reduction Algorithm

The Procedure

Input: Graph G, Approximation factor α.

1. **Set the Penalty:** Let $n = |V|$ and choose a large weight $M = \alpha n + 1$.
2. **Construct G':** Create a complete graph where:
   - Existing edges in G get weight **1**.
   - Missing edges (non-edges) get weight M.
3. **Run the Solver:** Let C be the cost returned by $\textsc{Approx-TSP}(G')$.
4. **The Decision:**
   - If $C \leq \alpha n \longrightarrow$ Return **YES**.
   - If $C > \alpha n \longrightarrow$ Return **NO**.

Proof of Correctness

We analyze the output based on the structure of G:

Case 1: G has a Hamiltonian Cycle

- The optimal tour uses only original edges: $\mathrm{OPT}(G') = n$.
- The α-approx algorithm returns cost $C \leq \alpha \cdot \mathrm{OPT}$.
- Therefore, $C \leq \alpha n$.
- **Result:** Procedure correctly returns **YES**.

Proof of Correctness

We analyze the output based on the structure of G:

Case 1: G has a Hamiltonian Cycle

- The optimal tour uses only original edges: $\mathrm{OPT}(G') = n$.
- The α-approx algorithm returns cost $C \leq \alpha \cdot \mathrm{OPT}$.
- Therefore, $C \leq \alpha n$.
- **Result:** Procedure correctly returns **YES**.

Case 2: G has NO Hamiltonian Cycle

- Any tour must use at least one non-edge (weight M).
- Therefore, $\mathrm{OPT}(G') \geq M + (n-1) > \alpha n$.
- Since any tour cost $C \geq \mathrm{OPT}(G')$, we have $C > \alpha n$.
- **Result:** Procedure correctly returns **NO**.

Metric TSP and Its Approximation Algorithm

Turns out TSP is not so hopeless...

Metric TSP

Metric TSP An input instance to Metric TSP is a complete graph where edge weights are a metric.

- **Identity of indiscernibles:** $d(u, v) = 0 \ \Leftrightarrow \ u = v\,.$
- **Non-negativity:** $\forall u \neq v : \quad d(u, v) > 0.$
- **Symmetric:** $d(u, v) = d(v, u)\,.$
- **Triangle inequality:** $\forall u,\ v,\ w : \quad d(u, w) \leq d(u, v) + d(v, w)\,.$

Metric TSP

Metric TSP An input instance to Metric TSP is a complete graph where edge weights are a metric.

- **Identity of indiscernibles:** $d(u,v) = 0 \Leftrightarrow u = v\,.$
- **Non-negativity:** $\forall u \neq v : \quad d(u,v) > 0.$
- **Symmetric:** $d(u,v) = d(v,u)\,.$
- **Triangle inequality:** $\forall u,\ v,\ w : \quad d(u,w) \leq d(u,v) + d(v,w)\,.$

Why do we care?

- Many natural settings (FedEx, road networks, Euclidean distances) satisfy the triangle inequality.
- Triangle inequality enables good approximation algorithms.

2-Approximation for Metric TSP: MST Doubling

Algorithm ("double-tree"):

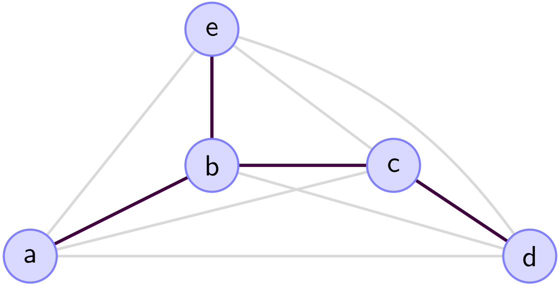
1. Compute a Minimum Spanning Tree (MST) T of the metric.
2. Double every edge of T to get an Eulerian multigraph.
3. Take an Euler tour and *shortcut* repeated vertices to obtain a TSP tour.

Visualizing the Double-Tree Algorithm

Step 1: Metric MST

We compute the MST of the metric graph (shown in **purple**).

Current Cost $\leq \mathrm{OPT}.$



Why?

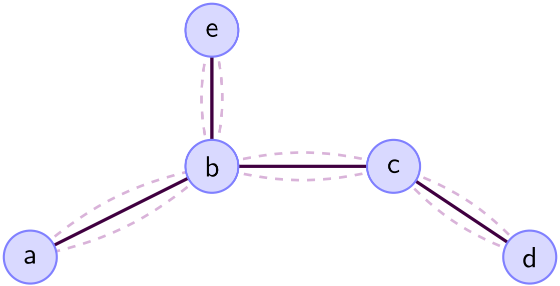
$$W(MST) \leq W(\text{ Hamiltonian Path}) \leq W(\text{OPT Hamiltonian Cycle})$$

Visualizing the Double-Tree Algorithm

Step 2: Doubling

We double every edge in the MST.
This creates an **Eulerian Multigraph** (every node has even degree).

Current Cost $\leq 2 \cdot \mathrm{OPT}$.

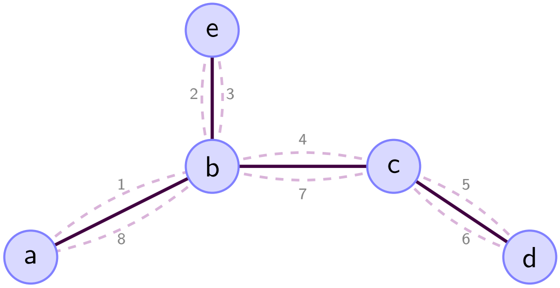


Visualizing the Double-Tree Algorithm

Step 3: Euler Tour

We traverse the doubled edges in a continuous loop (DFS order).

`a → b → e → b → c → d → c → b → a`

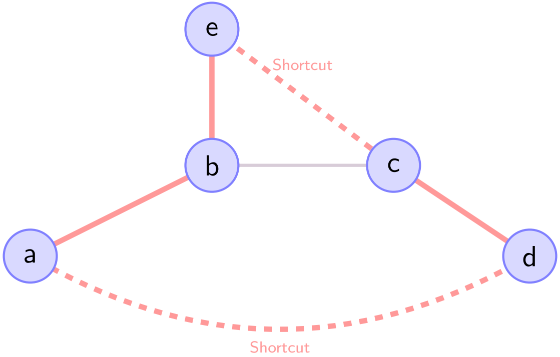


Visualizing the Double-Tree Algorithm

Step 4: Shortcutting

We delete repeated vertices from the Euler Tour. Thanks to the triangle inequality, taking a shortcut (red) is always cheaper.

Final Tour: `a → b → e → c → d → a`



$$W(\text{Tour with shortcuts}) \leq 2\,W(MST) \leq 2\,W(\text{OPT Hamiltonian Cycle})\,.$$

2-Approximation!

Set Cover

NP-Hard and NP-Complete Problems

NP-Hard Problem

A problem B is **NP-hard** if for every problem $A \in \mathrm{NP}$, A reduces to B.

NP-Hard and NP-Complete Problems

NP-Hard Problem

A problem B is **NP-hard** if for every problem $A \in \text{NP}$, A reduces to B.

NP-Complete Problem

A problem B is **NP-complete** if:

1. $B \in \text{NP}$, and
2. For every problem $A \in \text{NP}$, A reduces to B.

The NP-Completeness Recipe

To show a new problem B is NP-complete, start from a known NP-complete problem A. Show a polynomial-time reduction from A to B.

$$\left.\begin{array}{lr} \text{A is NP-complete:} & \text{NP} \le_p A \\ \text{A poly-time reduction from A to B:} & A \le_p B \end{array}\right\} \implies \text{NP} \le_p B$$

The NP-Completeness Recipe

To show a new problem B is NP-complete, start from a known NP-complete problem A. Show a polynomial-time reduction from A to B.

$$\left.\begin{array}{lr} \text{A is NP-complete:} & \text{NP} \leq_p A \\ \text{A poly-time reduction from A to B:} & A \leq_p B \end{array}\right\} \implies \text{NP} \leq_p B$$

The NP-Completeness Recipe

To show a new problem B is NP-complete, start from a known NP-complete problem A. Show a polynomial-time reduction from A to B.

$$\left.\begin{array}{ll} \text{A is NP-complete:} & \text{NP} \leq_p A \\ \text{A poly-time reduction from A to B:} & A \leq_p B \end{array}\right\} \implies \text{NP} \leq_p B$$

$\boxed{\text{B is NP-hard.}}$

The NP-Completeness Recipe

To show a new problem B is NP-complete, start from a known NP-complete problem A. Show a polynomial-time reduction from A to B.

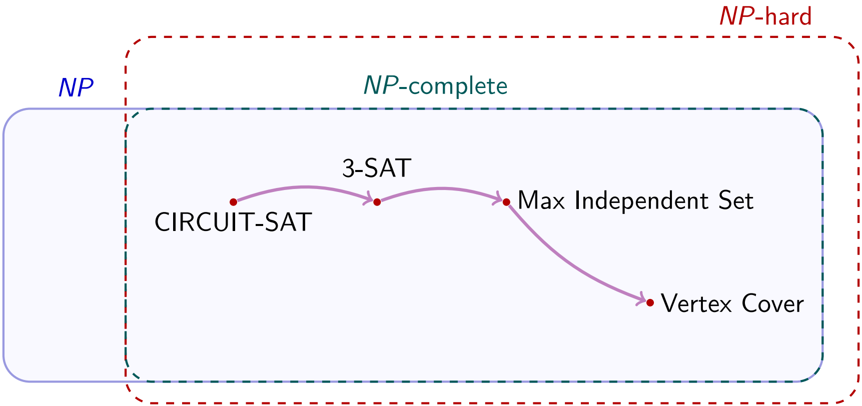
$$\left.\begin{array}{ll} \text{A is NP-complete:} & \text{NP} \leq_p A \\ \text{A poly-time reduction from A to B:} & A \leq_p B \end{array}\right\} \implies \text{NP} \leq_p B$$

$\boxed{\text{B is NP-hard.}}$

If we also show that B is in NP, then $\implies$ $\boxed{\text{B is NP-complete}}$

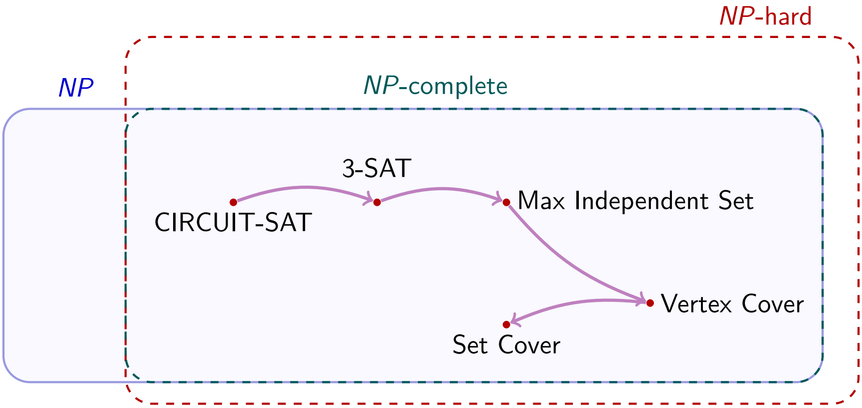
A Small NP-Completeness Family Portrait

Once one natural problem is shown NP-complete, the others follow by reductions.



A Small NP-Completeness Family Portrait

Once one natural problem is shown NP-complete, the others follow by reductions.



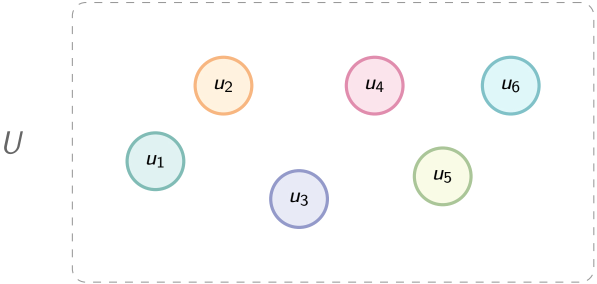
SET-COVER Is NP-Complete

A reduction from VERTEX-COVER

Set Cover

Let U be a finite set of n elements.

And, let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a family of subsets with $S_i \subseteq U$.

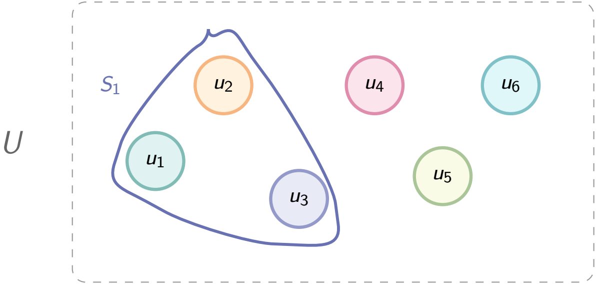


Set Cover

Let U be a finite set of n elements.

And, let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a family of subsets with $S_i \subseteq U$.

Example: $S_1 = \{u_1,\ u_2,\ u_3\}$ $S_2 = \{u_2,\ u_4,\ u_5\}$ $S_3 = \{u_3,\ u_4,\ u_5,\ u_6\}$

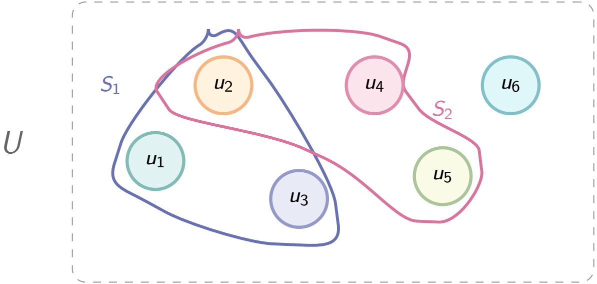


Set Cover

Let U be a finite set of n elements.

And, let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a family of subsets with $S_i \subseteq U$.

Example: $S_1 = \{u_1,\ u_2,\ u_3\}$ $S_2 = \{u_2,\ u_4,\ u_5\}$ $S_3 = \{u_3,\ u_4,\ u_5,\ u_6\}$

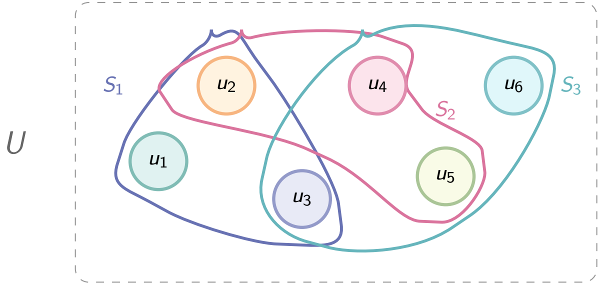


Set Cover

Let U be a finite set of n elements.

And, let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a family of subsets with $S_i \subseteq U$.

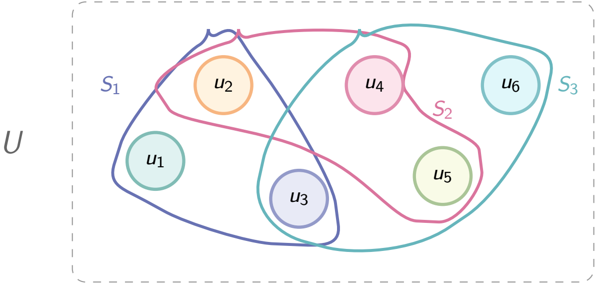
Example: $S_1 = \{u_1,\ u_2,\ u_3\}$ $S_2 = \{u_2,\ u_4,\ u_5\}$ $S_3 = \{u_3,\ u_4,\ u_5,\ u_6\}$



Set Covers

A set cover is a subfamily $C' \subseteq \mathcal{S}$ such that every element of U lies in at least one member of C'.

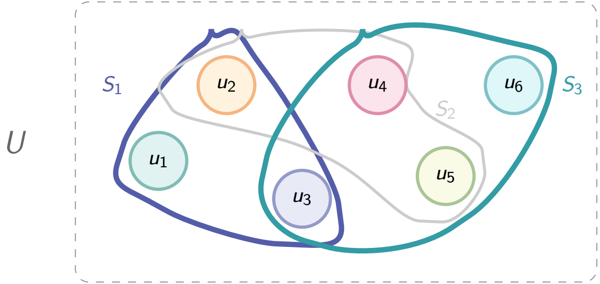
Equivalently: the union of the chosen sets equals U.



Set Covers

A set cover is a subfamily $C' \subseteq \mathcal{S}$ such that every element of U lies in at least one member of C'.

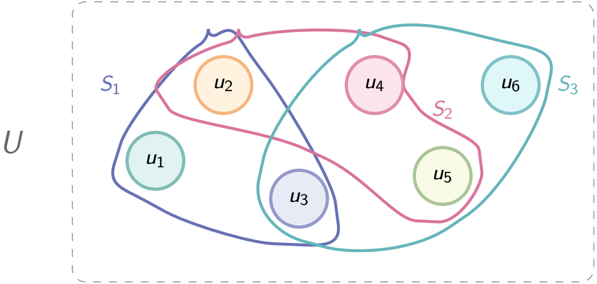
Equivalently: the union of the chosen sets equals U.



A set cover is, for example, $C' = \{S_1,\ S_3\}$ since $S_1 \cup S_3 = U$.

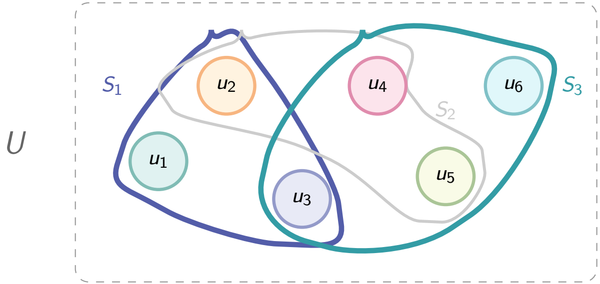
Set Cover (Decision Problem)

SET-COVER: Given a universe U, a family $\mathcal{S} = \{S_1, \ldots, S_m\}$ with $S_i \subseteq U$, and an integer k, does there exist a subfamily of size at most k whose union equals U?



Set Cover (Decision Problem)

SET-COVER: Given a universe U, a family $\mathcal{S} = \{S_1, \ldots, S_m\}$ with $S_i \subseteq U$, and an integer k, does there exist a subfamily of size at most k whose union equals U?

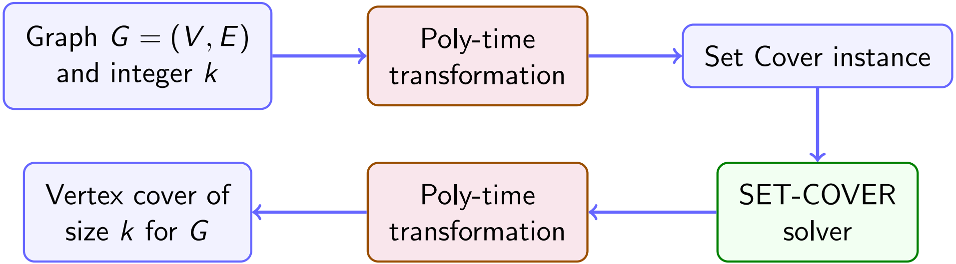


Here $k = 2$ and $\{S_1,\ S_3\}$ is a valid cover.

Plan: Reduce VERTEX-COVER to SET-COVER

- Input on the Vertex Cover side: An undirected graph $G = (V, E)$ and an integer k.
- We will build a Set Cover instance such that G has a vertex cover of size k iff $(U, \mathcal{S}, k)$ has a set cover of size k.

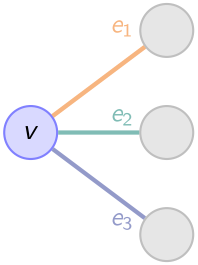
Reduction: VERTEX-COVER $\leq_p$ SET-COVER



The Resemblance: Vertex Cover vs. Set Cover

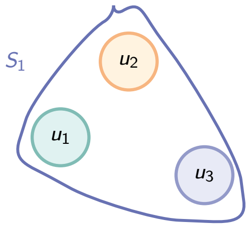
Vertex Cover

- **Items to cover:** Edges.
- **Objects to use:** Vertices.



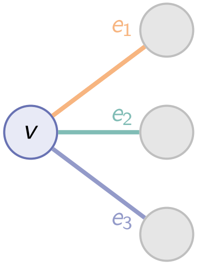
Set Cover

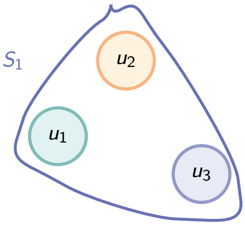
- **Items to cover:** Elements.
- **Objects to use:** Sets.



The Resemblance: Vertex Cover vs. Set Cover

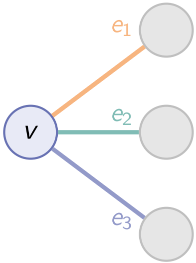
- Vertex Cover is essentially a **special case** of Set Cover.

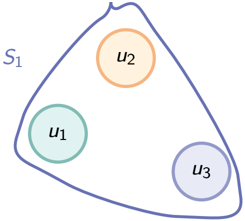




The Resemblance: Vertex Cover vs. Set Cover

- Vertex Cover is essentially a **special case** of Set Cover.
- Vertex Cover: cover all edges using the graph's vertex sets (each edge has 2 options).
- Set Cover: cover all elements using arbitrary sets (no limit on set size).





The Reduction: Vertex-Cover $\leq_p$ Set-Cover

Given a graph $G = (V, E)$ and integer k, we build a Set Cover instance:

- The **universe** is the set of edges:

$$U := E.$$

The Reduction: Vertex-Cover $\leq_p$ Set-Cover

Given a graph $G = (V, E)$ and integer k, we build a Set Cover instance:

- The **universe** is the set of edges:

$$U := E.$$

- For every vertex $v \in V$, we create a **set**

$$S_v := \{\, e \in E : e \text{ is incident to } v \,\}.$$

The Reduction: Vertex-Cover $\leq_p$ Set-Cover

Given a graph $G = (V, E)$ and integer k, we build a Set Cover instance:

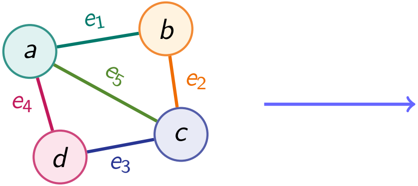
- The **universe** is the set of edges:
$$U := E.$$
- For every vertex $v \in V$, we create a **set**
$$S_v := \{\, e \in E : e \text{ is incident to } v \,\}.$$
- The family of sets is
$$\mathcal{S} = \{S_v : v \in V\}.$$

The Reduction: Vertex-Cover $\leq_p$ Set-Cover

Given a graph $G = (V, E)$ and integer k, we build a Set Cover instance:

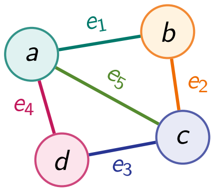
- The **universe** is the set of edges:
$$U := E.$$
- For every vertex $v \in V$, we create a **set**
$$S_v := \{\, e \in E : e \text{ is incident to } v \,\}.$$
- The family of sets is
$$\mathcal{S} = \{S_v : v \in V\}.$$
- The Set Cover instance asks:
$$\exists\ C' \subseteq \mathcal{S} \text{ of size } \leq k \text{ such that } \bigcup_{S \in C'} S = U?$$

The Reduction: VERTEX-COVER $\leq_p$ SET-COVER

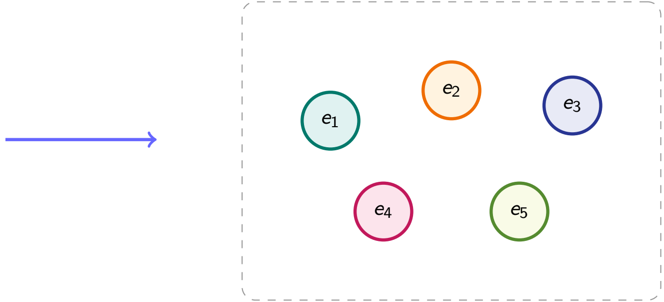


Graph G

The Reduction: VERTEX-COVER $\leq_p$ SET-COVER

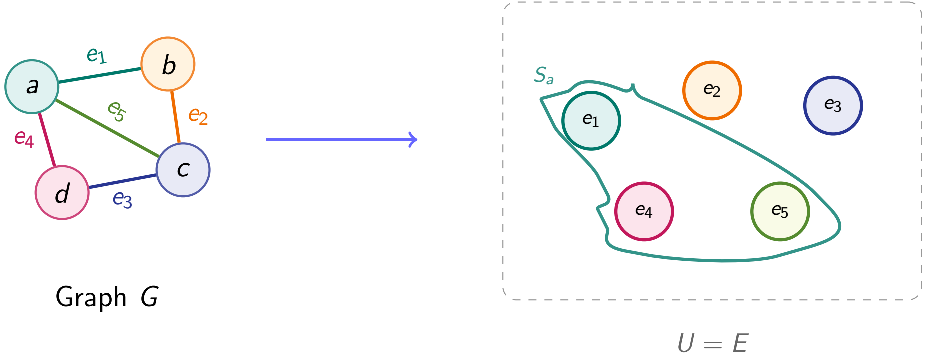


Graph G

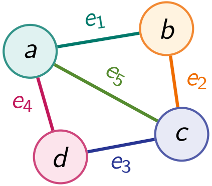


$U = E$

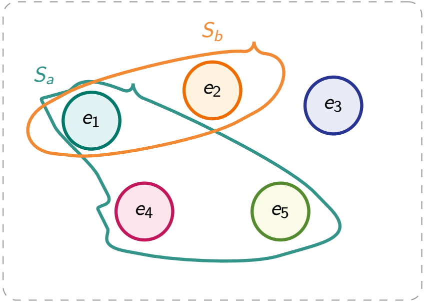
The Reduction: VERTEX-COVER $\leq_p$ SET-COVER



The Reduction: VERTEX-COVER $\leq_p$ SET-COVER

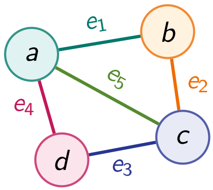


Graph G

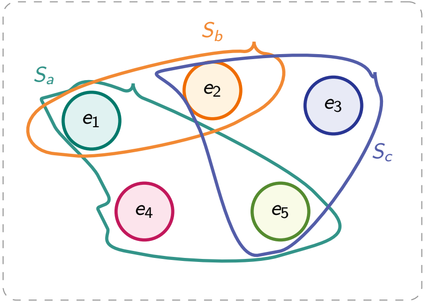


$U = E$

The Reduction: VERTEX-COVER $\leq_p$ SET-COVER

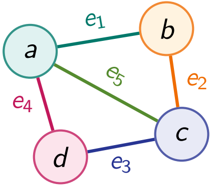


Graph G

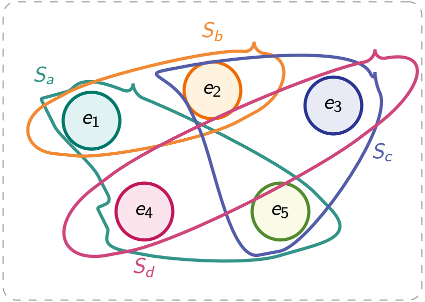


$U = E$

The Reduction: VERTEX-COVER $\leq_p$ SET-COVER

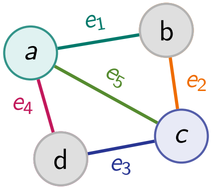


Graph G

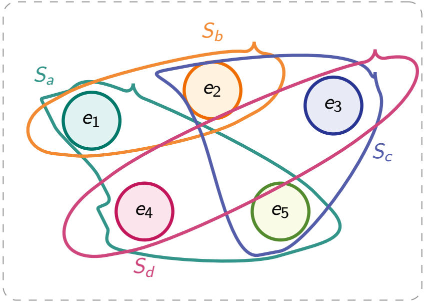


$U = E$

The Reduction: VERTEX-COVER $\leq_p$ SET-COVER



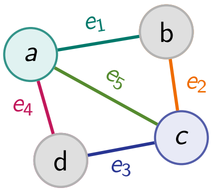
Graph G



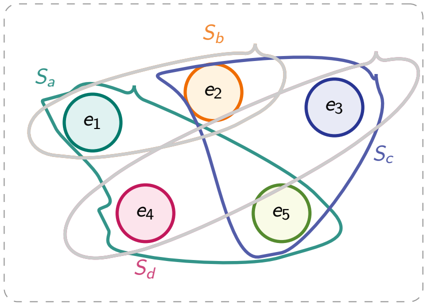
$U = E$

smallest vertex cover: $C = \{a, c\}$

The Reduction: VERTEX-COVER $\leq_p$ SET-COVER



Graph G



$U = E$

smallest vertex cover: $C = \{a, c\}$

smallest set cover: $C' = \{S_a, S_c\}$

Correctness of the Reduction

To prove the reduction is correct, we must show that the two instances have the same answer.

(⇒) Suppose $C \subseteq V$ is a vertex cover of size at most k. Every edge $e = \{u, v\} \in E$ has at least one endpoint in C.
Thus every element $e \in U$ is contained in at least one of the sets $C' = \{S_v : v \in C\}$, meaning these $|C| \leq k$ sets form a valid set cover.

(⇐) Suppose we have a set cover $C' = \{S_{v_1}, \ldots, S_{v_t}\}$ with $t \leq k$. Since C' covers all elements of $U = E$, for every edge $e = \{u, v\}$, at least one of the sets S_{v_i} contains e.
However, $e \in S_{v_i}$ exactly when v_i is an endpoint of e.
Therefore the vertices $C = \{v_1, \ldots, v_t\}$ touch every edge of G, so C is a vertex cover of size at most k.

Correctness of the Reduction

Efficiency. The construction is polynomial-time:

$$|U| = |E|, \qquad |\mathcal{S}| = |V|, \qquad \text{each edge is added to exactly two sets.}$$

SET-COVER is in NP. A certificate is a subfamily $\{S_{i_1}, \ldots, S_{i_t}\}$ with $t \leq k$. A polynomial-time verifier checks:

- $t \leq k$,
- for every $u \in U$ there exists j with $u \in S_{i_j}$.

Conclusion. Since VERTEX-COVER is NP-complete and we have given a polynomial-time reduction from VERTEX-COVER to SET-COVER, it follows that SET-COVER is NP-hard. Since it is also in NP, SET-COVER is NP-complete.

Approximation Algorithm for Set Cover

Greedy Algorithm for Set Cover

Greedy rule: At each step, pick the set that covers the *largest number of currently uncovered* elements. Break ties arbitrarily; repeat until all elements are covered.

The Algorithm:

1. Initialize $C \leftarrow \emptyset$ and $U_{\text{left}} \leftarrow U$.

Greedy Algorithm for Set Cover

Greedy rule: At each step, pick the set that covers the *largest number of currently uncovered* elements. Break ties arbitrarily; repeat until all elements are covered.

The Algorithm:

1. Initialize $C \leftarrow \emptyset$ and $U_{\text{left}} \leftarrow U$.
2. While $U_{\text{left}} \neq \emptyset$:

Greedy Algorithm for Set Cover

Greedy rule: At each step, pick the set that covers the *largest number of currently uncovered* elements. Break ties arbitrarily; repeat until all elements are covered.

The Algorithm:

1. Initialize $C \leftarrow \emptyset$ and $U_{\text{left}} \leftarrow U$.
2. While $U_{\text{left}} \neq \emptyset$:
   - Choose $S \in \mathcal{S}$ maximizing $|S \cap U_{\text{left}}|$.

Greedy Algorithm for Set Cover

Greedy rule: At each step, pick the set that covers the *largest number of currently uncovered* elements. Break ties arbitrarily; repeat until all elements are covered.

The Algorithm:

1. Initialize $C \leftarrow \emptyset$ and $U_{\text{left}} \leftarrow U$.
2. While $U_{\text{left}} \neq \emptyset$:
   - Choose $S \in \mathcal{S}$ maximizing $|S \cap U_{\text{left}}|$.
   - Add S to C and set $U_{\text{left}} \leftarrow U_{\text{left}} \setminus S$.

Greedy Algorithm for Set Cover

Greedy rule: At each step, pick the set that covers the *largest number of currently uncovered* elements. Break ties arbitrarily; repeat until all elements are covered.

The Algorithm:

1. Initialize $C \leftarrow \emptyset$ and $U_{\text{left}} \leftarrow U$.
2. While $U_{\text{left}} \neq \emptyset$:
   - Choose $S \in \mathcal{S}$ maximizing $|S \cap U_{\text{left}}|$.
   - Add S to C and set $U_{\text{left}} \leftarrow U_{\text{left}} \setminus S$.
3. Return C.

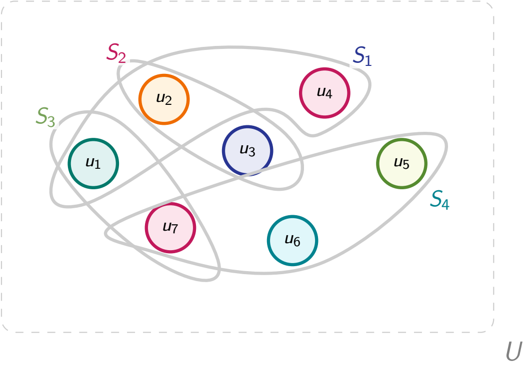
Greedy Set Cover Approximation

Algorithm Status:

Pick set with max *uncovered* elements.

Set	Elements	Count
S_1	u_1, u_2, u_4	**3**
S_4	u_5, u_6, u_7	**3**
S_2	u_2, u_3	2
S_3	u_1, u_7	2

Initial State: All elements uncovered.



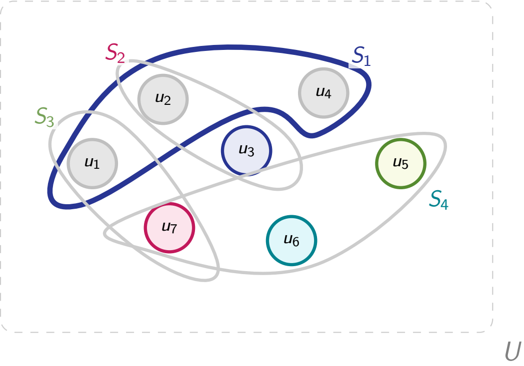
Greedy Set Cover Approximation

Algorithm Status:

Pick set with max *uncovered* elements.

Set	Elements	Count
S_4	u_5, u_6, u_7	**3**
S_2	$\cancel{u_2}$, u_3	1
S_3	$\cancel{u_1}$, u_7	1
S_4	Selected	–

Step 1: Pick S_1. Covers $\{u_1, u_2, u_4\}$.



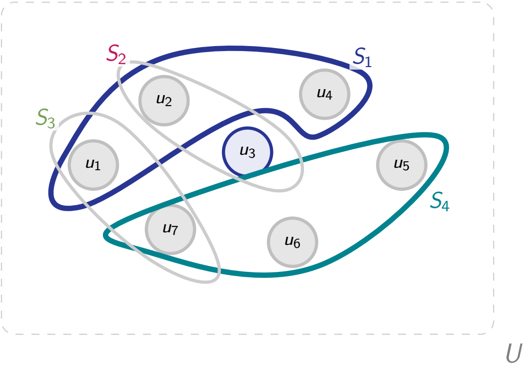
Greedy Set Cover Approximation

Algorithm Status:

Pick set with max *uncovered* elements.

Set	Elements	Count
S_2	$\cancel{u_2}$, u_3	**1**
S_3	$\cancel{u_1}$, $\cancel{u_7}$	0
S_1	Selected	–
S_4	Selected	–

Step 2: Pick S_4. Covers $\{u_5, u_6, u_7\}$.



Greedy Set Cover Approximation

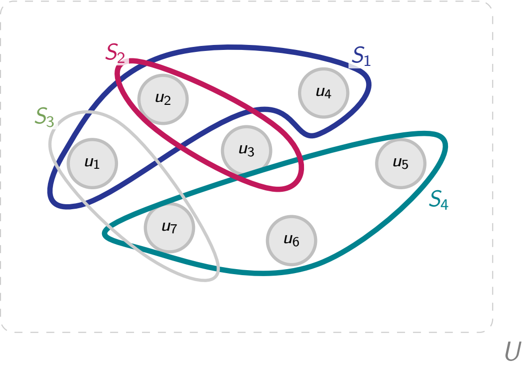
Algorithm Status:

Pick set with max *uncovered* elements.

Set	Elements	Count
S_3	~~u_1~~, ~~u_7~~	0
S_1	Selected	–
S_4	Selected	–
S_2	Selected	–

Step 3: Pick S_2. Covers $\{u_3\}$.

Done! All elements covered.



Greedy is a $(\ln n + 1)$-Approximation

- Let OPT be the minimum number of sets in an optimal cover.

Greedy is a $(\ln n + 1)$-Approximation

- Let OPT be the minimum number of sets in an optimal cover.
- Suppose t elements remain uncovered.

Greedy is a $(\ln n + 1)$-Approximation

- Let OPT be the minimum number of sets in an optimal cover.
- Suppose t elements remain uncovered.
- Since OPT uses OPT sets to cover all t of them,

$$\exists S \in \mathcal{S} : |S \cap \text{uncovered}| \geq \frac{t}{\text{OPT}}.$$

Greedy is a $(\ln n + 1)$-Approximation

- Let OPT be the minimum number of sets in an optimal cover.
- Suppose t elements remain uncovered.
- Since OPT uses OPT sets to cover all t of them,

$$\exists S \in \mathcal{S} : |S \cap \text{uncovered}| \geq \frac{t}{\text{OPT}}.$$

- Greedy picks a set covering *at least this many*.

Greedy is a $(\ln n + 1)$-Approximation

- Let t_i denote the number of uncovered elements at the end of step i.

Greedy is a $(\ln n + 1)$-Approximation

- Let t_i denote the number of uncovered elements at the end of step i.
- Greedy choice at this step covers $\frac{t_i}{\mathrm{OPT}}$ many elements.

Greedy is a $(\ln n + 1)$-Approximation

- Let t_i denote the number of uncovered elements at the end of step i.
- Greedy choice at this step covers $\frac{t_i}{\mathrm{OPT}}$ many elements.
- Therefore the number of uncovered elements shrinks by a factor:

$$t_{i+1} \;=\; t_i - \#\,\text{elements covered in step } i+1 \;\leq\; t_i\left(1 - \frac{1}{\mathrm{OPT}}\right).$$

Greedy is a $(\ln n + 1)$-Approximation

- Let t_i denote the number of uncovered elements at the end of step i.
- Greedy choice at this step covers $\frac{t_i}{\mathrm{OPT}}$ many elements.
- Therefore the number of uncovered elements shrinks by a factor:

$$t_{i+1} \; = \; t_i - \#\text{ elements covered in step } i+1 \; \leq \; t_i \left(1 - \frac{1}{\mathrm{OPT}}\right).$$

- After k steps:

$$t_k \leq t_{k-1}\left(1 - \frac{1}{\mathrm{OPT}}\right) \leq t_{k-2}\left(1 - \frac{1}{\mathrm{OPT}}\right)^2 \leq \cdots \leq n\left(1 - \frac{1}{\mathrm{OPT}}\right)^k < n\; e^{-k/\mathrm{OPT}}.$$

Greedy is a $(\ln n + 1)$-Approximation

- If k is sufficiently large then then the number of uncovered elements should drop below 1:

$$k = \lceil \mathrm{OPT} \cdot \ln n \rceil \qquad \Longrightarrow \qquad t_k < n\, e^{-0k/\mathrm{OPT}} \leq 1\,.$$

- Thus, the algorithm certainly terminates after k steps since no uncovered element left.

Greedy is a $(\ln n + 1)$-Approximation

- If k is sufficiently large then then the number of uncovered elements should drop below 1:

$$k = \lceil \mathrm{OPT} \cdot \ln n \rceil \qquad \Longrightarrow \qquad t_k < n\, e^{-0k/\mathrm{OPT}} \leq 1\,.$$

- Thus, the algorithm certainly terminates after k steps since no uncovered element left.

Thus greedy uses at most $(\ln n + 1) \cdot \mathrm{OPT}$ sets.

Summary: Greedy Approximation for Set Cover

- Greedy repeatedly picks the set with the largest number of uncovered elements.

Summary: Greedy Approximation for Set Cover

- Greedy repeatedly picks the set with the largest number of uncovered elements.
- Each step removes a **constant fraction** of what remains.

Summary: Greedy Approximation for Set Cover

- Greedy repeatedly picks the set with the largest number of uncovered elements.
- Each step removes a **constant fraction** of what remains.
- After at most $(\ln n + 1) \cdot \mathrm{OPT}$ steps, all elements are covered.

Summary: Greedy Approximation for Set Cover

- Greedy repeatedly picks the set with the largest number of uncovered elements.
- Each step removes a **constant fraction** of what remains.
- After at most $(\ln n + 1) \cdot \mathrm{OPT}$ steps, all elements are covered.
- Therefore:

$$|\mathcal{C}_{\text{greedy}}| \;\leq\; (\ln n + 1) \cdot |\mathcal{C}^*|.$$

Summary: Greedy Approximation for Set Cover

- Greedy repeatedly picks the set with the largest number of uncovered elements.
- Each step removes a **constant fraction** of what remains.
- After at most $(\ln n + 1) \cdot \text{OPT}$ steps, all elements are covered.
- Therefore:
$$|\mathcal{C}_{\text{greedy}}| \;\leq\; (\ln n + 1) \cdot |\mathcal{C}^*|.$$
- This is optimal up to constants unless $\mathrm{P} = \mathrm{NP}$.

References

Erickson, J. (2019).
Algorithms.
Self-published.

Tardos, E. and Kleinberg, J. (2005).
Algorithm Design.
Pearson.