2025 Fall CSE5025

Combinatorial Optimization

组合优化

Instructor: 刘晟材

Lecture 3-1: Graph Optimization Problems and Approximation Algorithms

Agenda for Today's Lecture

In this lecture, we will focus on

- The Vertex Cover Problem and its approximation algorithms
- The Set Cover Problem and its classic greedy logarithmic-factor approximation
- The Traveling Salesman Problem (General TSP and Metric TSP)

Learning Objectives for this lecture

- Know fundamental combinatorial optimization problems on graph, and know classic approximation algorithms for them
- Learn how to design approximation algorithms using various techniques, including greedy choices, linear programming relaxation, and randomized methods
- Master the concept of the approximation ratio and be able to analyze it

What is an Approximation Algorithm?

An **approximation algorithm** is an algorithm that finds an approximate (i.e., not necessarily optimal) solution to an NP-hard optimization problem.

Key Properties:

- It must run in polynomial time.
- It must output a feasible solution
- There must be a provable guarantee on the quality of the solution

The Approximation Ratio (ρ):

This guarantee is quantified by the approximation ratio (or factor), denoted by ρ (rho).

For a minimization problem:

- An algorithm is a ρ-approximation if: Solution's Cost $\leq \rho \cdot$ Optimal Cost

For a maximization problem:

- An algorithm is a ρ-approximation if: Optimal Value $\leq \rho \cdot$ Solution's Value

In the above definition, the ratio ρ is always ≥ 1.

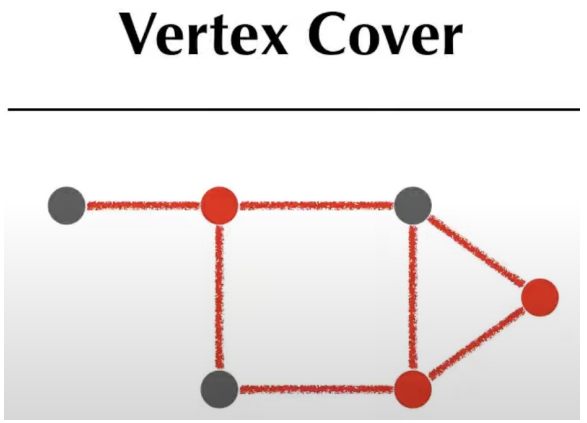
A 1-approximation algorithm is an exact algorithm.

The Vertex Cover (VC) Problem

Input: An undirected graph $G = (V, E)$.

Objective: Find a subset of vertices $C \subseteq V$ of minimum size, such that every edge in the graph is "covered" (i.e., at least one of its endpoints is in C).

Type: Minimization Problem.



Hardness: NP-hard. This means we do not expect to find a polynomial-time algorithm that always finds the optimal solution.

Approximation Algorithms for VC

We will explore three distinct approaches to approximating the minimum vertex cover:

- A simple strategy based on selecting edges
- An intuitive greedy strategy based on selecting high-degree vertices
- A more powerful method based on Linear Programming

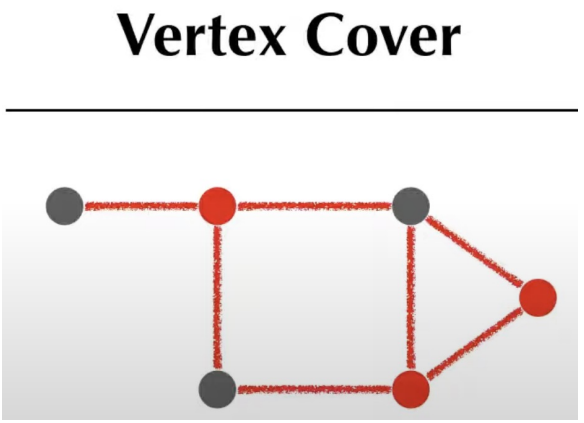
VC Approx. Algorithm 1: Maximal Matching

Core Idea: If an edge is uncovered, we must add at least one of its endpoints to our cover. This algorithm makes such a choice: it adds both. This ensures progress by covering at least one new edge in each step.

The set of edges (a set of non-adjacent edges) chosen forms a maximal matching (极大匹配).

VC Approx. Algorithm 1: Maximal Matching

```
Algorithm Approx_VC_Matching(Graph G=(V, E)):
    // C is the vertex cover to be constructed
    C = {}

    // E' is the set of uncovered edges
    E_prime = E

    while E_prime is not empty:
        // 1. Select an arbitrary uncovered edge
        let (u, v) be an edge in E_prime

        // 2. Add both endpoints to the cover
        C = C ∪ {u, v}

        // 3. Remove all edges covered by u or v
        remove from E_prime every edge incident on either u or v

  return C
```



Complexity Analysis

If E_prime is a simple list, the while loop runs at most $|E|$ times.

Inside the loop, the step “remove from E_prime every edge...” would require iterating through the entire list to find and delete edges. This can take $O(|E|)$ time in each iteration.

Total Time Complexity: $O(|E|) \times O(|E|) = O(|E|^2)$. This is polynomial.

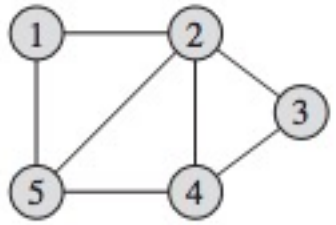
Can we do better?

A More Efficient Implementation (1)

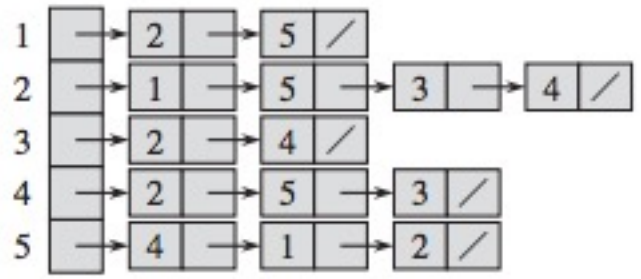
The efficiency of this algorithm depends on how we implement the process of finding an uncovered edge and removing incident edges.

Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges.

Data Structures: We can represent the graph using **adjacency lists**.



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

To keep track of which edges are covered, we can use a simple Boolean array, *is_edge_covered*, of size m, initialized to false.

A More Efficient Implementation (2)

Main Loop: The algorithm iterates through all edges of the graph **just once.**

We perform a single pass through the list of all `m` edges.

For each edge `e = (u,v)` in the list:

- We check if `e` has already been covered. If `is_edge_covered[e]` is `true`, we do nothing and move to the next edge.
- If `is_edge_covered[e]` is `false`, this means we've found an uncovered edge. This is the "select an arbitrary edge" step.
  - **Action 1:** Add its endpoints `u` and `v` to our cover set `C`.
  - **Action 2:** Mark all incident edges as covered. We traverse the adjacency list of `u` and `v`. For each neighbor `w` of `u`, we set `is_edge_covered[(u,w)] = true`. We do the same for `v`.

A More Efficient Implementation (3)

Initializing the `is_edge_covered` array takes $O(m)$ time.

The main loop iterates `m` times. The check inside is $O(1)$.

The crucial part is the total work done in **Action 2** across the entire algorithm. A vertex `v` is added to the cover `C` at most once. When it is added, we traverse its adjacency list, which has `deg(v)` entries.

The total number of these adjacency list traversals is therefore bounded by the sum of the degrees of all vertices that end up in the cover. In the worst case, this is the sum of degrees of all vertices in the graph, which is $\sum_{v \in V} \deg(v) = 2m$.

Total Time Complexity = $O(m) + O(m) = O(m)$**.** If the graph is represented by adjacency lists, the total time is $O(m + n)$ or simply $O(|V| + |E|)$.

Approx_VC_Matching: Approximation Ratio (1)

Theorem: The Maximal Matching algorithm is a **2-approximation** algorithm for VC.

Proof:

1) Let M be the set of edges selected by the algorithm in the while loop.

2) The size of our algorithm's cover is exactly $|C| = 2|M|$, because for each edge in M, we added two vertices to C.

3) By construction, no two edges in M share a vertex (otherwise, selecting the first edge would have caused the second to be removed). M is a matching.

Approx_VC_Matching: Approximation Ratio (2)

4) Now, consider the optimal cover, C_{OPT}. To cover the edges in M, C_{OPT} must include at least one endpoint for each edge in M.

5) Since no two edges in M share an endpoint, C_{OPT} must contain at least $|M|$ distinct vertices to cover all of them.

6) Therefore, we have the inequality: $|C_{OPT}| \geq |M|$. Combining these facts: $|C| = 2 \cdot |M| \leq 2 \cdot |C_{OPT}|$.

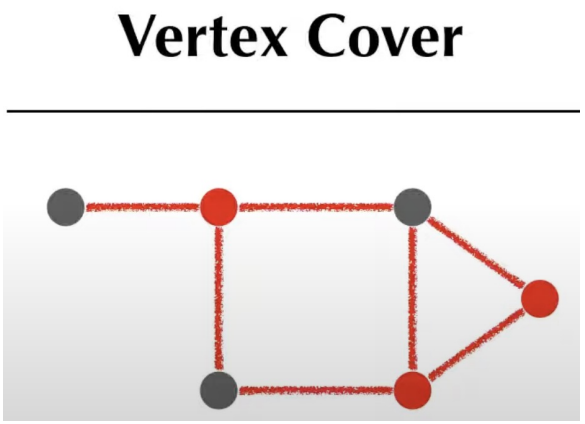
VC Approx. Algorithm 2: Greedy by Degree

Core Idea: This is a more intuitive “bang-for-the-buck” (直接) approach. In each step, we select the vertex that *covers the most currently uncovered edges*. This seems like it should be very effective.

Intuition: Prioritize the “most connected” vertices, as they are the most efficient at covering edges.

VC Approx. Algorithm 2: Pseudocode

```
Algorithm Approx_VC_Degree(Graph G=(V, E)):
    C = {}
    E_uncovered = E

    while E_uncovered is not empty:
        // 1. Find the most efficient vertex
        select a vertex v in V that covers the maximum number of edges in E_uncovered

      // 2. Add it to the cover
        C = C ∪ {v}

      // 3. Update the set of uncovered edges
        remove from E_uncovered all edges incident on v

  return C
```



Complexity Analysis

The while loop runs at most $|V|$ times, as one vertex is added to the cover in each iteration.

Inside the loop:

- To find the vertex v that covers the maximum number of uncovered edges, we must iterate through all vertices and count their incident edges in the remaining graph. This can take up to $O(|E|)$ in each iteration.
- Removing the edges incident on v also takes time proportional to its degree.

Total Complexity: The dominant step is finding the vertex. This leads to a worst-case complexity of approximately $O(|V| \cdot |E|)$. **Can we do better?**

Approx_VC_Degree: Approximation Ratio

Question: Is this more intuitive greedy algorithm better than the first one?

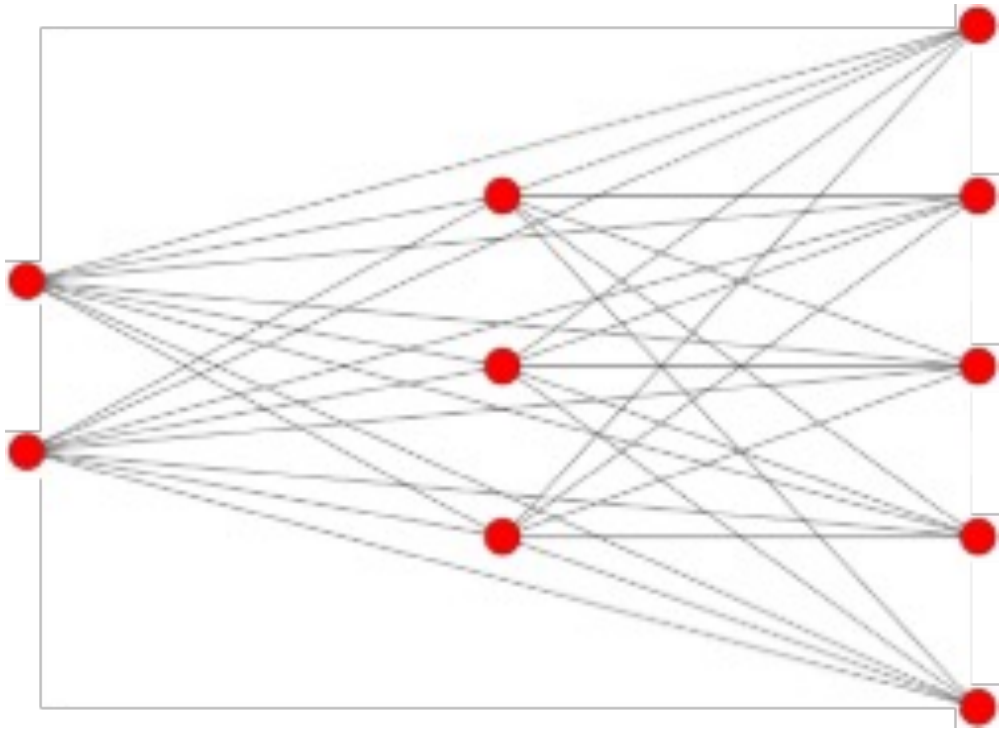
Answer: Surprisingly, no. While it often performs well in practice, its worst-case guarantee is much weaker.

Theorem: The Approx_VC_Degree algorithm is an $H(n)$-approximation algorithm for VC, where n is the number of vertices and $H(n)$ is the n-th Harmonic number ($H(n) \approx \ln n$)

This is not a constant-factor approximation.

Approx_VC_Degree: Intuition

Worst-Case Intuition: It is possible to construct special graphs (called "k-partite graphs", *k*分图) where the algorithm is repeatedly tricked.



VC Approx. Algorithm 3: LP Relaxation + Rounding

Core Idea: This powerful technique bridges continuous and discrete optimization. We formulate the problem as an integer program, "relax" it into a continuous problem that's easy to solve, and then use the fractional solution to guide us to a good integer solution.

Step 1: Integer Linear Programming (ILP) Formulation

- For each vertex $v \in V$, create a binary variable $x_v \in \{0, 1\}$.
- $x_v = 1$ if we include vertex v in the cover, 0 otherwise.
- **Minimize:** $\sum_{v \in V} x_v$ (Minimize the size of the cover)
- **Subject to:** For every edge $(u, v) \in E$, we must have $x_u + x_v \geq 1$.
  - (This constraint ensures that for every edge, at least one of its endpoints is chosen).

VC Approx. Algorithm 3: LP Relaxation + Rounding

Step 2: LP Relaxation

- We relax the integer constraint $x_v \in \{0, 1\}$ to be a continuous constraint: $0 \leq x_v \leq 1$.
- This is now a Linear Program (LP), which can be solved optimally in polynomial time.
- The solution will be a set of fractional values x_v^*, e.g., $x_a^* = 0.5, x_b^* = 0.5, x_c^* = 1$.

Step 3: Rounding

- How do we convert these fractions back into a 0/1 decision? The simplest way is a threshold-based rounding.
- **Algorithm:**
  1. Solve the LP relaxation to get the optimal fractional solution x_v^*.
  2. Create a cover `C` by including every vertex `v` for which $x_v^* \geq 0.5$.

VC Approx. Algorithm 3: Pseudocode

```
Algorithm Approx_VC_LP_Rounding(Graph G=(V, E)):
    // 1. Define the LP for Vertex Cover on G
    LP_problem = Formulate_VC_LP(G)

    // 2. Solve the LP to get the optimal fractional solution
    x_star = Solve_LP(LP_problem)

    // 3. Round the solution to get an integer cover
    C = {}
    for each vertex v in V:
        if x_star[v] >= 0.5:
            C = C ∪ {v}

  return C
```

VC Approx. Algorithm 3: Complexity

Formulating the Linear Program: We need to define: n variables (one for each vertex), n non-negativity constraints, m main constraints $\rightarrow O(|V| + |E|)$

The LP could be solved in polynomial time in the size of problem input $\rightarrow$ Poly$(|V| + |E|)$

Rounding $\rightarrow O(|V|)$

Total complexity in polynomial time is clearly dominated by the LP solver.

VC Approx. Algorithm 3: Solution Feasibility

Question: Is the cover `C` returned by this rounding procedure always a valid vertex cover?

Answer: Yes!

Proof:

- Consider any edge $(u, v) \in E$.
- From the LP constraints, we know that the fractional solution must satisfy $x_u^* + x_v^* \geq 1$.
- It is impossible for *both* x_u^* and x_v^* to be less than 0.5, because their sum would be less than 1.
- Therefore, at least one of them must be ≥ 0.5.
- This means that our rounding rule will place at least one of the vertices (`u` or `v`) into the cover `C` .
- Since this holds for every edge, `C` is a valid vertex cover.

VC Approx. Algorithm 3: Approximation Ratio

Theorem: The LP Rounding algorithm is a **2-approximation** algorithm for Vertex Cover.

Proof Sketch:

1. Let Z_{LP}^* be the value of the optimal solution to the LP relaxation. Let Z_{OPT} be the value of the optimal integer solution (the true minimum vertex cover).
2. Since the LP has fewer constraints (it can use fractions), its optimal solution must be less than or equal to the optimal integer solution: $Z_{LP}^* \leq Z_{OPT}$.
3. Let `C` be the cover produced by our rounding algorithm. Its size is $|C| = \sum_{v \in C} 1$.
4. Since we only include vertices where $x_v^* \geq 0.5$, we can say that for any $v \in C$, $1 \leq 2 \cdot x_v^*$.
5. Therefore, the size of our cover is:
   $|C| = \sum_{v \in C} 1 \leq \sum_{v \in C} 2 \cdot x_v^* \leq \sum_{v \in V} 2 \cdot x_v^* = 2 \cdot \sum_{v \in V} x_v^* = 2 \cdot Z_{LP}^*$.
6. Combining these inequalities:
   $|C| \leq 2 \cdot Z_{LP}^* \leq 2 \cdot Z_{OPT}$.

Conclusion: The solution is feasible and its size is at most twice the optimal size.

From Vertex Cover to Set Cover

We saw that a vertex in a graph can be thought of as “covering” the set of edges connected to it.

The **Set Cover (SC) problem** is a powerful generalization of this idea. Instead of vertices covering edges, we have abstract sets covering elements.

Because it is more general, it can model an even wider range of real-world problems. It is one of the most fundamental problems in combinatorial optimization.

The SC Problem: Definition

Given

- A universe of n elements, $U = \{e_1, e_2, \ldots, e_n\}$
- A collection of m subsets of U, denoted by $S = \{S_1, S_2, \ldots S_m\}$, where each $S_i \subseteq U$

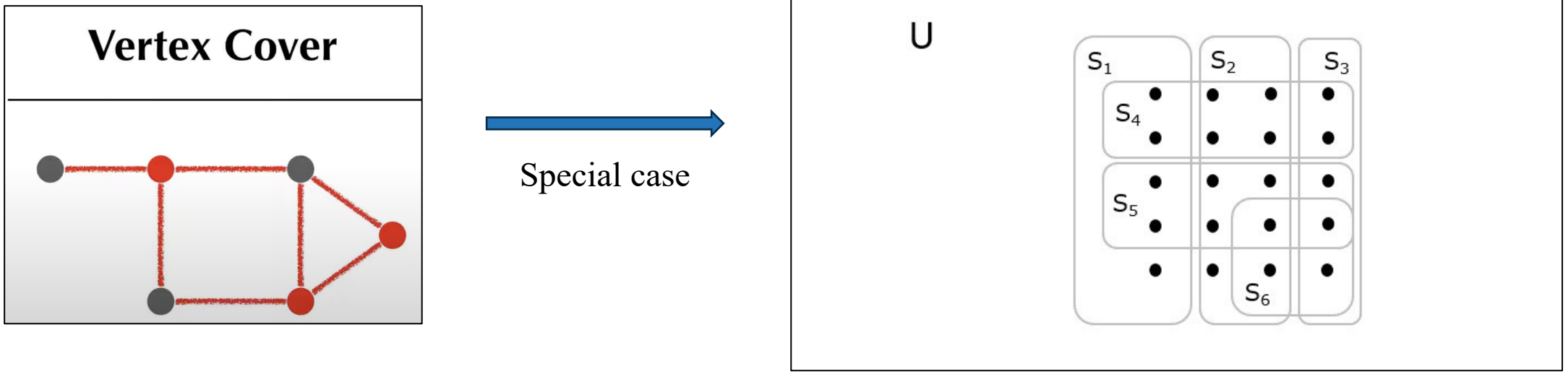
Solution

- A sub-collection of these sets, $C \subseteq S$, whose union is U

Objective: find a set cover (SC) C with the minimum number of sets

Type: Minimization Problem

The SC Problem: Example



Q1: The SC problem is NP-Hard, **why**?

Q2: Is set cover problem a special case of vertex cover problem?

The Greedy Algorithm for SC

Core Idea: This is the most natural and widely used approximation algorithm for SC. The strategy is to make the most efficient choice at each step. “Efficiency” here means covering the maximum number of currently uncovered elements.

Intuition: At each stage of building our cover, we ask: “Which single set gives me the most new coverage right now?” We add that set to our solution and repeat the process.

Example

```
Algorithm Greedy_Set_Cover(Universe U, Collection S):
    // U_covered keeps track of elements covered so far
    U_covered = {}

    // C is the final cover we are building
    C = {}

    while U_covered != U:
        // 1. Find the set that covers the most uncovered elements
        let S_best be the set in S that maximizes |S_i ∩ U_covered|

        // 2. Add this best set to our cover
        C = C U {S_best}

        // 3. Update the set of covered elements
        U_covered = U_covered  ∪  S_best

  return C
```

Greedy_Set_Cover: Approximation ratio (1)

Theorem: The greedy algorithm for SC is an $H(n)$-approximation algorithm, where $n = |U|$ is the size of the universe and $H(n)$ is the n-th Harmonic number.

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Asymptotic Behavior: $H(n) \approx \ln(n) + \gamma$, where γ is the Euler-Mascheroni constant (~0.577).

Greedy_Set_Cover: Approximation ratio (2)

This is an $O(\log n)$-approximation.

What this means:

- The approximation ratio is **not a constant.**
- It **grows very slowly** with the size of the problem.
- For a problem with 1,000,000 elements, the solution found by the greedy algorithm is guaranteed to be no more than about $\ln(106) \approx 13.8$ times the size of the optimal solution.

Proof of the Approximation Ratio (1)

Our Goal: We aim to prove that the size of the greedy solution, $|C|$, is bounded by $|C| \leq H(n) \cdot |C_{OPT}|$. The proof relies on a clever "cost sharing" or "amortized analysis" argument.

The Cost Sharing Idea:

1. The total cost of our solution is the number of sets we select, which is $|C|$.
2. Each time our algorithm adds a set `S_best` to `C`, we "pay" a cost of 1.
3. We will distribute this cost of 1 equally among the elements that are **newly covered** in that specific step.

Proof of the Approximation Ratio (2)

Defining the "Price" of an Element:

- Let an element `e` be covered for the first time when we select the set `S_best`.
- Let `k` be the number of new elements covered by `S_best` in that step.
- We define the **price** paid for covering element `e` as:

$$price(e) = \frac{1}{k}$$

- The total cost of our solution is therefore the sum of the prices paid for each element in the universe:

$$|C| = \sum_{e \in U} price(e)$$

Proof of the Approximation Ratio (3)

The Core Lemma We Will Prove:

- The entire proof hinges on finding an upper bound for the price of each element. We will prove the following lemma:
- The price paid for the `j`-th element to be covered by the algorithm is at most:

$$\frac{|C_{OPT}|}{n-j+1}$$

Proof of the Approximation Ratio (4)

Let's analyze a single iteration of the `while` loop to establish a bound on the price.

State of the Algorithm:

- Let U_{rem} be the set of elements that are still uncovered at the beginning of this iteration.
- Let $|C_{OPT}|$ be the size of an optimal solution.

The Key Insight: The optimal solution C_{OPT} is a collection of $|C_{OPT}|$ sets that, by definition, must cover **all** elements in the universe `U`. This implies they must also cover all the currently remaining elements in U_{rem}.

Proof of the Approximation Ratio (5)

The Pigeonhole Principle / Averaging Argument:

- The $|C_{OPT}|$ sets in the optimal solution collectively cover the $|U_{rem}|$ elements.
- Let's consider the total number of "coverings" these optimal sets provide for the remaining elements. This sum is $\sum_{S_i \in C_{OPT}} |S_i \cap U_{rem}|$. This sum must be at least $|U_{rem}|$, since every element is covered at least once.
- Therefore, the **average** number of remaining elements covered by a set in C_{OPT} is at least $\frac{|U_{rem}|}{|C_{OPT}|}$.
- This means there must exist at least **one set** in the optimal solution, let's call it S_{opt_best}, that covers at least this average number of elements.

$$|S_{opt\_best} \cap U_{rem}| \geq \frac{|U_{rem}|}{|C_{OPT}|}$$

Proof of the Approximation Ratio (6)

Connecting the Greedy Choice to the Optimal Average:

- In this iteration, our greedy algorithm selects the set `S_best` that maximizes the number of newly covered elements.
- By its very definition, the greedy choice must be *at least as good as* the choice of any other single set, including the best-performing set from the optimal solution, S_{opt_best}.
- Therefore, the number of new elements covered by our greedy choice `S_best` is:

$$|S_{best} \cap U_{rem}| \geq |S_{opt\_best} \cap U_{rem}| \geq \frac{|U_{rem}|}{|C_{OPT}|}$$

Proof of the Approximation Ratio (7)

Bounding the Price (Proving the Lemma):

- The price paid by any element `e` covered in this step is exactly $\frac{1}{|S_{best} \cap U_{rem}|}$.
- Using the inequality above, we can bound this price:

$$price(e) \leq \frac{1}{|U_{rem}|/|C_{OPT}|} = \frac{|C_{OPT}|}{|U_{rem}|}$$

- Let's say `e` was the `j`-th element to be covered overall. At the start of its covering step, there were exactly $n - (j-1) = n - j + 1$ elements remaining. So, $|U_{rem}| = n - j + 1$.
- Substituting this in, we get: $price(e) \leq \frac{|C_{OPT}|}{n-j+1}$. The lemma is proven.

Proof of the Approximation Ratio (8)

Summing the Prices to Get the Final Ratio:

1. Let's order the elements $e_1, e_2, \ldots, e_n$ in the sequence they were covered.
2. The price of the `j`-th element covered, e_j, is at most $\frac{|C_{OPT}|}{n-j+1}$.
3. The total cost of our solution, $|C|$, is the sum of all these prices:

$$|C| = \sum_{j=1}^{n} price(e_j) \leq \sum_{j=1}^{n} \frac{|C_{OPT}|}{n-j+1}$$

4. By changing the index of summation (let $k = n - j + 1$), the sum becomes:

$$|C| \leq |C_{OPT}| \cdot \sum_{k=1}^{n} \frac{1}{k} = |C_{OPT}| \cdot H(n)$$

5. **Q.E.D.** We have proven that the cost of the greedy solution is at most $H(n)$ times the cost of the optimal solution.

Hardness of Approximating SC

Theorem (Feige, 1998): Unless NP has quasi-polynomial time algorithms (a complexity theory conjecture stronger than $P \neq NP$), there is no polynomial-time $(1-\epsilon)\ln(n)$-approximation algorithm for SC for any $\epsilon > 0$.

What this means: The simple greedy algorithm is essentially the best possible polynomial-time approximation algorithm we can hope for. Finding a constant-factor approximation (like the 2-approximation we found for Vertex Cover) for SC would be a monumental breakthrough in computer science, implying $P = NP$.

The Traveling Salesman Problem (TSP)

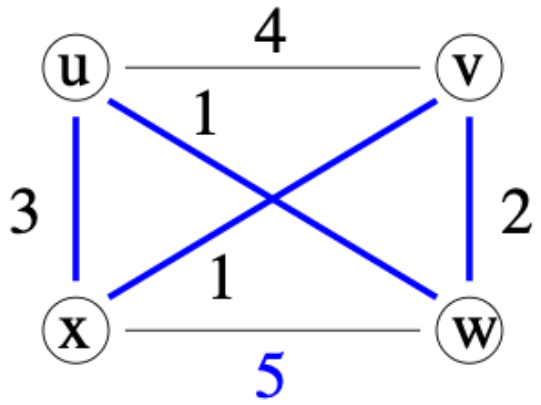
We now shift from problems of selection and covering to problems of sequencing and routing.

The goal is no longer just to pick a set of objects, but to find the best order in which to visit them.

This brings us to perhaps the most famous NP-hard problem of all: the Traveling Salesman Problem (TSP).

TSP

Optimization Problem TSP: Given a complete weighted undirected graph with n vertices, and a cost function $c(u, v)$ that gives the the weight (cost) of each edge (u, v) is given by the, **find a Hamiltonian cycle of least weight**.



Tour (Hamiltonian cycle)
with minimum cost 7

Solution: A tour (cycle) that visits every city exactly once and returns to the starting city.

Type: Minimization Problem

General TSP

General TSP: Edge costs (cost function) can be arbitrary. This version is **extremely hard to approximate**.

Claim: If we could approximate it with any constant factor ρ, we could solve the Hamiltonian Cycle problem in polynomial time (implying $P = NP$).

What this means: Finding even a "bad" approximation for General TSP (e.g., a tour guaranteed to be no more than 1000 times the optimal length) is just as hard as finding the absolute best solution to any other NP-complete problem.

Why?

Proof of the Claim (1)

Proof Strategy: We will prove this using a reduction.

We will show that if such a hypothetical ρ-approximation algorithm for TSP existed, we could use it as a "oracle" to build a new, polynomial-time algorithm that solves the Decision Hamiltonian Cycle (DHamCyc) problem exactly.

Since DHamCyc is NP-complete, this would imply $P = NP$.

Proof of the Claim (2)

Part 1: The reduction from DhamCyc

Our goal is to create a polynomial-time algorithm that can solve any instance of the DHamCyc problem.

Input for DHamCyc: An instance is a graph $G = (V, E)$.

- (Note: `G` is an undirected graph, and it is **not necessarily a complete graph**.)

Proof of the Claim (3)

The Reduction: We transform this graph `G` into a special instance of the General TSP problem, let's call it `G'`.

1. **Vertices:** The TSP instance `G'` will have the same set of vertices, `V`. Let $n = |V|$.
2. **Edges & Costs:** `G'` will be a **complete graph**. We define the cost $c(u, v)$ for every pair of vertices (u, v) as follows:

$$c(u,v) = \begin{cases} 1 & \text{if the edge } (u,v) \text{ exists in the original graph } G \\ M & \text{if the edge } (u,v) \text{ does not exist in } G \end{cases}$$

3. **Choosing the "Expensive" Cost `M`:** `M` must be a very large number. To make the proof work, we will choose `M` strategically based on our hypothetical approximation ratio ρ. We set:

$$M = \rho \cdot n + 1$$

This transformation from `G` to `G'` can clearly be done in polynomial time.

Proof of the Claim (4)

Part 2: Creating the Critical Cost Gap

Now, let's analyze the optimal TSP tour cost in our constructed instance `G'`, which we denote as C_{OPT}. The value of C_{OPT} depends entirely on whether the original graph `G` has a Hamiltonian Cycle.

Case 1: The original graph `G` HAS a Hamiltonian Cycle.

- If a DHamCyc exists in `G`, then there is a tour in `G'` that uses only the edges that were present in `G`.
- Every edge in this tour has a cost of **1**.
- Since a tour on `n` vertices has `n` edges, the total cost of this tour is exactly $n \times 1 = n$.
- This must be the optimal TSP tour, because any other tour would be forced to use at least one edge of cost `M`, which is much larger than `n`.
- **Conclusion for Case 1:** $C_{OPT} = n$.

Proof of the Claim (5)

Case 2: The original graph `G` does NOT have a Hamiltonian Cycle.

- In this case, any tour in the complete graph `G'` **must** use at least one edge that was *not* in the original graph `G` (otherwise, the tour itself would constitute a DHamCyc in `G`).
- This means any tour must use at least one "expensive" edge of cost `M`.
- The cost of such a tour would be composed of at most `n-1` "cheap" edges (cost 1) and at least one "expensive" edge (cost M).
- The total cost of any tour is therefore bounded below by:
  $C_{OPT} \geq (n-1) \cdot 1 + 1 \cdot M = n - 1 + (\rho \cdot n + 1) = \rho \cdot n + n.$
- **Conclusion for Case 2:** $C_{OPT} > \rho \cdot n.$

Proof of the Claim (5)

Part 3: The Contradiction

Now, we use our hypothetical polynomial-time ρ-approximation algorithm for General TSP, which we'll call `APPROX-TSP`.

Our Complete Algorithm for the DHamCyc Problem:

1. Given the input graph `G` for DHamCyc, construct the TSP instance `G'` with costs 1 and `M = ρn + 1`. (This step is polynomial time).
2. Run `APPROX-TSP` on `G'`. Let the cost of the tour it returns be C_A. (This step is polynomial time by our assumption).
3. By the definition of a ρ-approximation algorithm, we are guaranteed that $C_A \leq \rho \cdot C_{OPT}$.

Proof of the Claim (6)

4. **Analyze the result C_A based on our two cases:**
   - If **Case 1** was true (G has a DHamCyc), then $C_{OPT} = n$. Our algorithm's returned cost will satisfy: $C_A \leq \rho \cdot n$.
   - If **Case 2** was true (G has no DHamCyc), then $C_{OPT} > \rho n$. Since any tour must have a cost of at least C_{OPT}, the returned tour cost C_A must also satisfy $C_A \geq C_{OPT}$. Therefore, $C_A > \rho \cdot n$.

Proof of the Claim (7)

The Final Step - The Decision:

- After running `APPROX-TSP` and getting the cost C_A, we can make a definitive decision:

$$\text{If } C_A \leq \rho \cdot n, \text{ then OUTPUT "Yes, G has a Hamiltonian Cycle."}$$

$$\text{If } C_A > \rho \cdot n, \text{ then OUTPUT "No, G does not have a Hamiltonian Cycle."}$$

Conclusion: We have constructed a complete, deterministic, polynomial-time algorithm that correctly solves the NP-complete DHamCyc problem. This implies that **P = NP**. Since the scientific consensus is that P ≠ NP, our initial assumption must be false. Therefore, **no such polynomial-time constant-factor approximation algorithm for General TSP can exist.**

Metric TSP

Metric TSP: The edge costs satisfy the triangle inequality: for any three cities A, B, and C, the direct cost $c(A, C)$ is no more than the indirect cost via B, i.e., $c(A, C) \leq c(A, B) + c(B, C)$.

Why this matters: Most real-world TSP instances based on physical distances are metric. This property is the key that unlocks efficient approximation algorithms. From now on, we will only consider Metric TSP.

Summary of Approximation Algorithms

Problem	Algorithm	Strategy	Approximation ratio
Vertex cover	Maximal Matching	Iteratively pick any uncovered edge and add **both** of its endpoints to the cover.	2
	Greedy By Degree	Iteratively pick the vertex that covers the most remaining (uncovered) edges.	O(logn)
	LP Relaxation + Rounding	Solve a "relaxed" continuous version of the problem, then round the fractional results (e.g., any value ≥ 0.5 becomes 1).	2
Set cover	Greedy by coverage	Iteratively pick the set that covers the most new (currently uncovered) elements.	O(logn)
General Traveling Salesman Problem	No known efficient approximation algorithm	N/A	No constant-factor approximation exists unless P=NP.

Assignment 1

Q1. 1) Build a transformation that can transform any instance of the vertex cover (VC) problem (on page 6) to an instance of the set cover (SC) problem (on page 28**) in polynomial time**, and 2) given VC is NP-hard, prove SC is also NP-hard using **Cook reduction.** (4 points)

Q2. Develop a more efficient implementation of the "VC Approx. Algorithm 2: Greedy by Degree" (on page 17) with complexity better than $O(|V| \cdot |E|)$. Describe the used data structures and analyze the runtime for each algorithmic step. (3 points)

Q3. Prove the nearest neighbor algorithm has no **constant-factor approximation ratio** for the metric TSP. (3 points)