Approximation and String Algorithms

Thomas Nowak

These are the lecture notes for the M1 course Approximation and String Algorithms at ENS Paris-Saclay in the academic year 2026–2027. Good references for the material covered here include Williamson and Shmoys [1] for approximation algorithms and Lothaire [29] for the combinatorics on words underlying the string algorithms. The website for the course is at `https://algo.biodis.co/`.

This document will inevitably contain some errors. If you find any, I will be grateful and happy to hear from you. You can reach me by email at thomas@thomasnowak.net.

Thomas Nowak
ENS Paris-Saclay
Academic Year 2026–2027
Last Update: September 19, 2026

Contents

September 7, 2026

Lecture 1: Approximation Algorithms I

Many fundamental optimization problems are NP-hard, meaning that finding exact optimal solutions is computationally infeasible for large instances. Approximation algorithms provide a principled approach to finding near-optimal solutions efficiently, with provable guarantees on solution quality.

1.1 Definitions

An optimization problem asks us to find a feasible solution that minimizes (or maximizes) some objective function. For a minimization problem, let OPT denote the optimal solution value and ALG the value produced by algorithm A.

Definition 1.1

Algorithm A is an α-approximation algorithm if

$$\text{ALG} \leq \alpha \cdot \text{OPT}$$

for all instances of the problem, where $\alpha \geq 1$ is called the approximation ratio.

For maximization problems, we require $\text{ALG} \geq \text{OPT}/\alpha$. The closer α is to 1, the better the approximation guarantee. An approximation ratio of $\alpha = 2$ means our solution is at most twice as expensive as optimal.

The landscape of approximable problems is characterized by several complexity classes. The class APX contains problems that admit constant-factor polynomial-time approximation algorithms. These are the "nice" NP-hard problems: hard to solve exactly, but not impossibly hard to approximate. A Polynomial-Time Approximation Scheme (PTAS) is more refined: for any $\varepsilon > 0$, there exists a $(1+\varepsilon)$-approximation algorithm running in time polynomial in n (though possibly exponential or worse in $1/\varepsilon$). For example, running time $O(n^{1/\varepsilon})$ is allowed, which becomes impractical for small ε. The strongest guarantee is a Fully Polynomial-Time Approximation Scheme (FPTAS), which runs in time polynomial in both n and $1/\varepsilon$, such as $O(n^2/\varepsilon)$.

Not all NP-hard problems admit good approximations. Some problems are provably hard to approximate within any reasonable factor unless P = NP, while others admit PTAS or even FPTAS.

1.2 Vertex Cover

The vertex cover problem serves as our first case study. Given an undirected graph $G = (V, E)$, a vertex cover is a subset $C \subseteq V$ such that every edge has at least one endpoint in C. We want to find a vertex cover of minimum size.

A natural greedy approach is to repeatedly select the vertex of maximum degree, add it to the cover, and remove all incident edges. Let us analyze both the performance and limitations of this approach.

Theorem 1.1. The greedy maximum degree algorithm achieves an approximation ratio of $O(\log n)$ for vertex cover.

Proof. Let C be the cover produced by the greedy algorithm. We will track the number of edges remaining after each iteration. Let $E_0 = E$ and let E_i denote the set of edges remaining after the i-th iteration, when we have selected i vertices for the cover.

In iteration $i+1$, we select a vertex v of maximum degree Δ_i in the graph (V, E_i). This vertex covers Δ_i edges, so $|E_{i+1}| = |E_i| - \Delta_i$.

Now consider the optimal solution, which uses at most OPT vertices to cover all edges. In particular, these OPT vertices must cover the $|E_i|$ edges that remain in E_i. By the pigeonhole principle, at least one vertex in the optimal solution must cover at least $|E_i|/\mathrm{OPT}$ of these edges.

Since the greedy algorithm selects the vertex of maximum degree, we have:

$$\Delta_i \geq \frac{|E_i|}{\mathrm{OPT}}$$

Therefore:

$$|E_{i+1}| = |E_i| - \Delta_i \leq |E_i| - \frac{|E_i|}{\mathrm{OPT}} = |E_i| \left(1 - \frac{1}{\mathrm{OPT}}\right)$$

Unrolling this recurrence from $i = 0$ to $i = t - 1$ where t is the number of iterations:

$$|E_t| \leq |E_0| \left(1 - \frac{1}{\mathrm{OPT}}\right)^t$$

The algorithm terminates when $|E_t| = 0$, so we need:

$$|E_0| \left(1 - \frac{1}{\mathrm{OPT}}\right)^t < 1$$

Using the inequality $(1 - x) \leq e^{-x}$ for $x > 0$:

$$|E_0| \left(1 - \frac{1}{\mathrm{OPT}}\right)^t \leq |E_0|\, e^{-t/\mathrm{OPT}}$$

The right-hand side is less than 1 when $t > \mathrm{OPT} \cdot \log|E_0|$. Therefore the algorithm terminates with at most $\lceil \mathrm{OPT} \cdot \log|E_0| \rceil$ vertices. Since $|E_0| \leq \binom{n}{2} < n^2$, we have $|C| \leq \mathrm{OPT} \cdot \log n^2 + 1 = O(\mathrm{OPT} \cdot \log n)$. □

This seems promising: a logarithmic approximation ratio. However, the greedy approach can perform poorly on certain graph families.

Theorem 1.2. There exists a family of graphs on which the greedy maximum degree algorithm produces a cover of size $\Omega(\mathrm{OPT} \cdot \log n)$.

Proof. Consider the family of bipartite graphs $G_r = (L \cup R, E)$ constructed as follows. Let $L = \{\ell_1, \ldots, \ell_r\}$ and partition $R = R_1 \cup R_2 \cup \cdots \cup R_r$ where $|R_i| = \lfloor r/i \rfloor$. We connect each vertex in R_i to exactly i distinct vertices in L, subject to the constraint that every vertex in L has at most one neighbor in R_i. This is possible because R_i needs $i \cdot \lfloor r/i \rfloor \leq r$ distinct endpoints in L.

Concretely, label the vertices in R_i as $v_1^{(i)}, \ldots, v_{\lfloor r/i \rfloor}^{(i)}$ and connect $v_j^{(i)}$ to each of the i vertices $\ell_{(j-1)i+1}, \ell_{(j-1)i+2}, \ldots, \ell_{ji}$. This way, within each group R_i, the neighborhoods partition (a subset of) L into blocks of size i, so no vertex in L has more than one neighbor in R_i. As a result, every vertex in R_i has degree exactly i, and every vertex $\ell_j \in L$ has degree $|\{i : 1 \leq i \leq r,\ \ell_j \text{ has a neighbor in } R_i\}|$.

See Figure 1.1 for an illustration with $r = 4$.

The optimal solution is $\mathrm{OPT} = r$ (take all vertices in L, which covers every edge). However, the greedy algorithm can be forced to select all vertices in R. The maximum degree in G_r is r, achieved by the single vertex in R_r, so greedy selects it first. After removing its r incident edges, the remaining graph has maximum degree $r-1$ among the vertices in R_{r-1}, so greedy continues by picking vertices from R_{r-1}, then R_{r-2}, and so on down to R_1.

The total number of vertices selected is:

$$|R| = \sum_{i=1}^{r} \lfloor r/i \rfloor \approx r \sum_{i=1}^{r} \frac{1}{i} = r \cdot H_r \approx r \log r = \Omega(\mathrm{OPT} \cdot \log r)$$

This concludes the proof. □

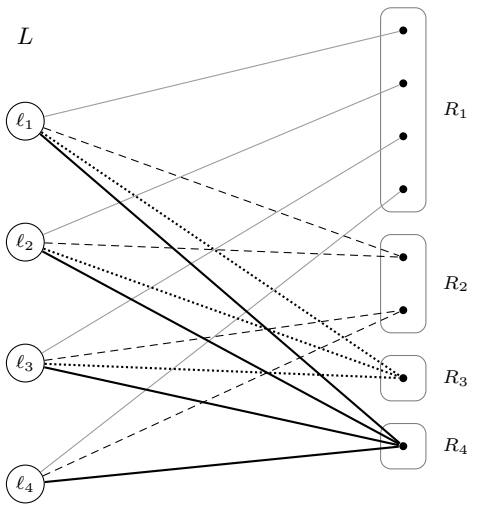

Figure 1.1: The graph G_4 with $r = 4$. Each vertex in R_i has degree exactly i. Edge styles distinguish groups: R_1 thin gray, R_2 dashed, R_3 dotted, R_4 thick solid.

This shows the greedy algorithm's approximation ratio is tight at $\Theta(\log n)$. Can we achieve a *constant-factor* approximation? Rather than guessing an algorithm and analyzing it after the fact, let us think about what a proof of a c-approximation would have to look like, and let the proof structure guide us to an algorithm. Suppose an algorithm produces a cover C and we claim $|C| \leq c \cdot \mathrm{OPT}$. Since computing OPT is NP-hard, the proof cannot simply compare $|C|$ to the optimal value. Instead, the proof must exhibit some efficiently verifiable *certificate* that OPT is large enough, that is, that $\mathrm{OPT} \geq |C|/c$.

What kind of certificate can witness that every vertex cover is large? Each edge imposes a *demand*: at least one of its endpoints must be in the cover. A single edge certifies $\mathrm{OPT} \geq 1$; two edges certify $\mathrm{OPT} \geq 2$—but only if they don't share an endpoint, since otherwise a single vertex could satisfy both demands simultaneously. More generally, to certify $\mathrm{OPT} \geq k$, we need k edges whose demands are independent, meaning no two of them can be satisfied by the same vertex. This happens exactly when the edges are *pairwise disjoint*: they share no endpoints.

So a c-approximation proof needs a set M of pairwise disjoint edges with $|C| \leq c \cdot |M|$. What if the algorithm itself builds M as a byproduct? The simplest idea: take both endpoints of every edge in M as the cover, giving $|C| = 2|M|$ and thus $c = 2$. But is this actually a valid vertex cover? An edge $e \notin M$ could escape coverage if neither of its endpoints belongs to C. This cannot happen if M is *maximal*: if

every edge in E shares an endpoint with some edge of M, then every edge is covered. This leads us to the following algorithm.

Algorithm:

1. Initialize $C \leftarrow \emptyset$ and $E' \leftarrow E$.
2. While $E' \neq \emptyset$:
   (a) Pick any edge $\{u, v\} \in E'$.
   (b) Add both u and v to C.
   (c) Remove all edges incident to u or v from E'.
3. Return C.

Note that this procedure computes a maximal matching M (the set of picked edges) and returns both endpoints of every edge in M.

Theorem 1.3. The edge-picking algorithm is a 2-approximation for vertex cover.

Proof. Let M denote the set of edges selected by the algorithm. By construction, M is a maximal matching: edges in M are pairwise disjoint, and no remaining edge can be added because all edges incident to selected vertices have been removed. Since we add both endpoints of each selected edge, $|C| = 2|M|$.

By maximality, every edge in E is incident to at least one vertex in C, so C is a valid vertex cover. Since the edges in M are pairwise disjoint, any vertex cover must include at least one distinct endpoint per edge, so $\mathrm{OPT} \geq |M|$. Therefore:

$$|C| = 2|M| \leq 2 \cdot \mathrm{OPT}$$

□

The 2-approximation for vertex cover has been known since the 1970s, and despite decades of effort, no polynomial-time algorithm with a better constant factor has been found. It is natural to ask whether we can do better than the 2-approximation.

On the hardness side, Dinur and Safra [2] proved that it is NP-hard to approximate vertex cover within a factor of 1.3606. This means that unless P = NP, no polynomial-time algorithm can guarantee a solution strictly better than $1.3606 \cdot \mathrm{OPT}$. Even stronger hardness results are known under the Unique Games Conjecture (UGC), introduced by Khot [4]. The UGC posits that a certain type of constraint satisfaction problem, where each constraint uniquely determines one variable given the other, is hard to approximate. Under this assumption, Khot and Regev [5] showed that vertex cover cannot be approximated within any factor $2 - \varepsilon$ for constant $\varepsilon > 0$. If the UGC is true, then our simple 2-approximation algorithm is essentially optimal. The gap between the 2-approximation upper bound and the current best hardness lower bound of 1.3606 (without assuming UGC) remains one of the major open problems in approximation algorithms.

1.3 Set Cover

The set cover problem generalizes vertex cover and many other covering problems. We are given a universe U with $|U| = n$ and a collection of subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ where $S_i \subseteq U$. The goal is to find a minimum cardinality subcollection $\mathcal{C} \subseteq \mathcal{S}$ such that the selected sets cover all elements, that is, $\bigcup_{S \in \mathcal{C}} S = U$.

Lemma 1.1. Vertex cover is a special case of set cover.

Proof. Consider a graph $G = (V, E)$. Let $U = E$ be the universe (the edges), and for each vertex $v \in V$, let $S_v = \{e \in E : v \in e\}$ be the set of edges incident to v. Then a vertex cover corresponds exactly to a set cover: we select vertices (sets) such that every edge (element) is covered.

Formally, $C \subseteq V$ is a vertex cover if and only if $\{S_v : v \in C\}$ is a set cover of $U = E$. Since $|C| = |\{S_v : v \in C\}|$, the optimal solutions have the same size. □

The natural greedy algorithm makes the locally optimal choice at each step: repeatedly select the set that covers the most uncovered elements.

Greedy Set Cover Algorithm:

1. Initialize $\mathcal{C} \leftarrow \emptyset$ and $R \leftarrow U$.
2. While $R \neq \emptyset$:
   (a) Select $S_i \in \mathcal{S}$ that maximizes $|S_i \cap R|$.
   (b) Add S_i to $\mathcal{C}$.
   (c) Update $R \leftarrow R \setminus S_i$.
3. Return $\mathcal{C}$.

To analyze this algorithm, we use a charging argument that assigns costs to elements as they are covered. This is a powerful technique: instead of directly bounding the number of sets, we assign a cost to each element and show the total cost is bounded. The beauty of this approach is that it reduces a combinatorial counting problem to an accounting problem.

The analysis uses the following bound on the harmonic numbers $H_n = \sum_{i=1}^{n} \frac{1}{i}$. It corresponds to a special case of the Euler–Maclaurin summation formula. Its basic message is that oftentimes it is possible to exchange an integral for a sum.

Lemma 1.2. For all $n \in \mathbb{N}$, we have $\log(n+1) \leq H_n \leq 1 + \log n$.

Proof. For the upper bound, we calculate:

$$\begin{aligned} H_n &= \sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i} \int_{i-1}^{i} 1 \, dx = 1 + \sum_{i=2}^{n} \int_{i-1}^{i} \frac{1}{i} \, dx \leq 1 + \sum_{i=2}^{n} \int_{i-1}^{i} \frac{1}{x} \, dx \\ &= 1 + \int_{1}^{n} \frac{1}{x} \, dx = 1 + \log n \end{aligned}$$

For the lower bound, we note:

$$\begin{aligned} H_n &= \sum_{i=1}^{n} \frac{1}{i} = \sum_{i=1}^{n} \frac{1}{i} \int_{i}^{i+1} 1 \, dx = \sum_{i=1}^{n} \int_{i}^{i+1} \frac{1}{i} \, dx \geq \sum_{i=1}^{n} \int_{i}^{i+1} \frac{1}{x} \, dx \\ &= \int_{1}^{n+1} \frac{1}{x} \, dx = \log(n+1) \end{aligned}$$

This concludes the proof. □

Theorem 1.4. The greedy algorithm for set cover is an $O(\log n)$-approximation.

Proof. We assign a cost to each element when it is covered by the greedy algorithm. Consider the iteration when the greedy algorithm selects a set S covering t new elements. We charge a cost of $1/t$ to each of these newly covered elements, so the total cost assigned in this iteration is exactly 1. Since the algorithm selects exactly $|\mathcal{C}|$ sets, and each set is "charged" a total cost of 1, the total cost equals $|\mathcal{C}|$.

Let $k = |R|$ denote the number of uncovered elements remaining before this iteration. Since an optimal solution must cover all n elements using at most OPT sets, by the pigeonhole principle at least one of the sets in the optimal solution must cover at least k/OPT of the remaining uncovered elements. The greedy algorithm selects the set covering the most uncovered elements, so it covers at least k/OPT elements, that is, $t \geq k/\mathrm{OPT}$.

Therefore, the cost charged per element in this iteration is:

$$\frac{1}{t} \leq \frac{\mathrm{OPT}}{k}$$

Now consider the perspective of a single element. Think about the element that is the j-th element to be covered, counting backwards from the end. That is, the last element covered has $j = 1$, the second-to-last has $j = 2$, and so on. When this element is covered, there are at least j elements still uncovered (including itself), so $k \geq j$. Thus, the cost assigned to this element is at most:

$$\frac{1}{t} \leq \frac{\mathrm{OPT}}{k} \leq \frac{\mathrm{OPT}}{j}$$

Summing over all n elements, the total cost assigned is:

$$\mathrm{Cost} \leq \sum_{j=1}^{n} \frac{\mathrm{OPT}}{j} = \mathrm{OPT} \sum_{j=1}^{n} \frac{1}{j} = \mathrm{OPT} \cdot H_n$$

Since each set selected by the greedy algorithm is assigned a total cost of 1 (distributed among the elements it covers), and the algorithm selects $|\mathcal{C}|$ sets, we have:

$$|\mathcal{C}| = \mathrm{Cost} \leq \mathrm{OPT} \cdot H_n$$

By Lemma 1.2, we have $H_n \leq 1 + \log n$, completing the proof. □

The charging argument is a fundamental technique in approximation algorithms. We will see it again in other contexts. The key idea is always the same: assign costs to objects (here, elements), argue that greedy makes good local decisions about these costs, and sum up to get a global bound.

Just as with vertex cover, it is natural to ask whether we can improve upon the logarithmic approximation ratio for set cover. The answer is largely negative. Feige [3] proved that set cover cannot be approximated within a factor of $(1 - \varepsilon) \log n$ for any $\varepsilon > 0$, unless NP has slightly unusual complexity (specifically, unless NP $\subseteq$ DTIME($n^{O(\log \log n)}$)). This shows that the greedy algorithm's $O(\log n)$-approximation is essentially optimal.

Two natural structural parameters control how far we can improve upon the $O(\log n)$ ratio for restricted instances: the *size* of the sets and the *frequency* of the elements.

If every set has size at most K, the greedy analysis tightens immediately. In the proof of Theorem 1.4, when a set S covers t new elements, each is charged $1/t$. Since $t \leq K$, the maximum charge any element receives across all iterations is bounded by $H_K = \sum_{i=1}^{K} 1/i$ rather than H_n. So the greedy algorithm achieves an H_K-approximation, which is $O(\log K)$; an improvement when $K \ll n$.

A stronger improvement comes from bounding the frequency. Define the *frequency* of an element as the number of sets containing it. If every element has frequency at most f, we can obtain an f-approximation by a simple rounding argument. Consider any optimal solution $\mathcal{C}^*$ with $|\mathcal{C}^*| = \text{OPT}$. For each element $e \in U$, assign it a weight of $1/f$. The total weight assigned to elements of any single set S_i is $|S_i|/f$. The total weight across all elements is n/f. On the other hand, we have $\sum_{S \in \mathcal{C}^*} |S| \geq n$. Now consider the algorithm that simply selects every set containing at least one uncovered element, greedily avoiding redundant sets: pick any uncovered element e; it belongs to at most f sets, and at least one of these sets is in $\mathcal{C}^*$. This means that any *maximal* sub-collection that covers U uses at most $f \cdot \text{OPT}$ sets, since each set in $\mathcal{C}^*$ "accounts for" at most f sets in our solution.

This explains why vertex cover admits a 2-approximation. Vertex cover is a special case of set cover (Lemma 1.1) where each element (edge) has frequency exactly 2: every edge belongs to exactly the two sets corresponding to its endpoints. The maximal matching algorithm from Theorem 1.3 is precisely an instantiation of the frequency-based argument with $f = 2$. Note also that the greedy maximum degree algorithm for vertex cover is the greedy set cover algorithm applied to this instance, so Theorem 1.4 gives an alternative $O(\log n)$ proof of Theorem 1.1.

1.4 Subset Sum

The vertex cover and set cover problems we've seen so far achieve constant-factor or logarithmic approximations. Can we do better? For some problems, the answer is yes: we can achieve arbitrarily good approximations.

The subset sum problem provides our first example of a PTAS and an FPTAS. Given positive integers $a_1, \ldots, a_n$ and a target T, we want to find a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i$ is as close to T as possible without exceeding it.

The standard dynamic programming approach maintains a table $\text{DP}[i][s]$ which is true if we can achieve sum exactly s using a subset of the first i items. The recurrence is:

$$\text{DP}[i][s] = \text{DP}[i-1][s] \vee \text{DP}[i-1][s - a_i]$$

This runs in time $O(nT)$, which is pseudo-polynomial: it's polynomial in the value of T but exponential in the number of bits needed to represent T. The idea is to reduce the number of distinct sums we track by "trimming" the list of reachable sums. For any $\delta > 0$, we say two values s and s' are δ-close if $s \leq s' \leq (1+\delta)s$. When maintaining our list of reachable sums, whenever we have two values that are δ-close, we keep only the smaller one. Set $\delta = \varepsilon/(2n)$. We maintain a list L_i of reachable sums using items from $\{a_1, \ldots, a_i\}$, but we keep L_i trimmed so that no two values are δ-close.

Algorithm:

1. Initialize $L_0 \leftarrow \{0\}$.
2. For $i \leftarrow 1$ to n:
   (a) Compute $L_i' \leftarrow L_{i-1} \cup \{s + a_i : s \in L_{i-1}\}$.
   (b) Trim L_i' to obtain L_i:
   - Sort L_i' in increasing order.
   - Initialize $L_i \leftarrow \emptyset$ and last $\leftarrow 0$.
   - For each $s \in L_i'$ in sorted order:
     - If $s > (1+\delta) \cdot \text{last}$ and $s \leq T$, add s to L_i and set last $\leftarrow s$.
3. Return the largest value in L_n.

Theorem 1.5. For any $\varepsilon > 0$, the above algorithm is a $(1+\varepsilon)$-approximation for subset sum, running in time $O(n^2 \log^2(nT)/\varepsilon)$.

Proof. We first bound the approximation error. Let OPT_i denote the maximum sum achievable using items from $\{a_1, \ldots, a_i\}$ that does not exceed T. We prove by induction that L_i contains a value y_i such that:

$$y_i \geq \frac{\mathrm{OPT}_i}{(1+\delta)^i}$$

In the base case $i = 0$, we have $L_0 = \{0\}$ and $\mathrm{OPT}_0 = 0$, so the claim holds. Now assume the claim holds for $i-1$. By the inductive hypothesis, there exists $y_{i-1} \in L_{i-1}$ with $y_{i-1} \geq \mathrm{OPT}_{i-1}/(1+\delta)^{i-1}$. If $\mathrm{OPT}_i = \mathrm{OPT}_{i-1}$ (item a_i is not used in the optimal solution), then y_{i-1} is added to L'_i. During trimming, either y_{i-1} itself survives in L_i, or it is replaced by a value $z \leq y_{i-1}$ with $z \geq y_{i-1}/(1+\delta)$. In either case, there exists $y_i \in L_i$ with $y_i \geq y_{i-1}/(1+\delta) \geq \mathrm{OPT}_i/(1+\delta)^i$. Otherwise, $\mathrm{OPT}_i = \mathrm{OPT}_{i-1} + a_i$ for some optimal subset that includes a_i. Let OPT'_{i-1} be the sum of the optimal subset of $\{a_1, \ldots, a_{i-1}\}$ that is used in this optimal solution for OPT_i. Then $\mathrm{OPT}_i = \mathrm{OPT}'_{i-1} + a_i$. By the inductive hypothesis, there exists $y'_{i-1} \in L_{i-1}$ satisfying $y'_{i-1} \geq \mathrm{OPT}'_{i-1}/(1+\delta)^{i-1}$. The value $y'_{i-1} + a_i$ is computed and added to L'_i. During trimming, either $y'_{i-1} + a_i$ itself survives, or it is removed because there is a smaller value z within a factor $(1+\delta)$ of it. In either case, there exists $y_i \in L_i$ with:

$$y_i \geq \frac{y'_{i-1} + a_i}{1+\delta} \geq \frac{\mathrm{OPT}'_{i-1}/(1+\delta)^{i-1} + a_i}{1+\delta} \geq \frac{\mathrm{OPT}_i}{(1+\delta)^i}$$

Therefore, the largest value in L_n is at least:

$$\frac{\mathrm{OPT}}{(1+\delta)^n} = \frac{\mathrm{OPT}}{(1+\varepsilon/(2n))^n}$$

Using $(1+x)^n \leq e^{nx}$ for $x > 0$:

$$(1+\varepsilon/(2n))^n \leq e^{\varepsilon/2}$$

For $\varepsilon \leq 1$, we have $\varepsilon/2 \leq \log(1+\varepsilon)$, so $e^{\varepsilon/2} \leq 1+\varepsilon$. Hence, the algorithm returns a value at least $\mathrm{OPT}/(1+\varepsilon)$. Since subset sum is a maximization problem, this means $\mathrm{ALG} \geq \mathrm{OPT}/(1+\varepsilon)$, so the algorithm is a $(1+\varepsilon)$-approximation.

We next bound the size of each list. We claim that $|L_i| = O(n \log(nT)/\varepsilon)$ for all i. The values in L_i are sorted and consecutive values differ by a factor of at least $(1+\delta)$. If $L_i = \{s_1, s_2, \ldots, s_k\}$ with $s_1 < s_2 < \cdots < s_k$, then $s_j \geq s_1 \cdot (1+\delta)^{j-1} \geq (1+\delta)^{j-1}$. Since $s_k \leq \sum_{i=1}^{n} a_i \leq nT$, we have $(1+\delta)^{k-1} \leq nT$ and thus, taking logarithms, $(k-1)\log(1+\delta) \leq \log(nT)$. Using $\log(1+\delta) \geq \delta/2$ for $\delta \leq 1$, we get:

$$|L_i| = k \leq 1 + \frac{\log(nT)}{\delta/2} = O\left(\frac{n \log(nT)}{\varepsilon}\right)$$

For the running time, each iteration i processes the list L_{i-1}, whose size is $O(n \log(nT)/\varepsilon)$. Computing L'_i takes $O(|L_{i-1}|)$ time, trimming takes $O(|L'_i|)$ time, and sorting takes $O(|L'_i| \log |L'_i|) = O((n \log(nT)/\varepsilon) \log(nT))$ time. Over n iterations, the total time is:

$$O\left(n \cdot \frac{n \log(nT)}{\varepsilon} \log(nT)\right) = O\left(\frac{n^2 \log^2(nT)}{\varepsilon}\right)$$

This concludes the proof. □

September 7, 2026

Lecture 2: Approximation Algorithms II

In the previous lecture, we studied approximation algorithms for vertex cover, set cover, and subset sum. We saw constant-factor approximations, logarithmic approximations, and an FPTAS. In this lecture, we continue with three more problems. The metric traveling salesman problem illustrates how structural assumptions (the triangle inequality) can turn an inapproximable problem into one admitting a constant-factor guarantee. The knapsack problem gives a second example of an FPTAS, this time via profit scaling rather than trimming. Load balancing on identical machines shows how a simple algorithmic refinement (sorting the input) can dramatically improve an approximation ratio.

2.1 Metric Traveling Salesman Problem

The traveling salesman problem (TSP) asks for a minimum-cost tour that visits every vertex in a graph exactly once and returns to the start. Given a complete graph K_n on n vertices with edge costs $c(u, v) \geq 0$, we seek a Hamiltonian cycle of minimum total cost. In the general case, TSP is extremely hard to approximate. Sahni and Gonzalez [7] showed that no finite approximation ratio is achievable.

Theorem 2.1. Unless P = NP, for any polynomial-time computable function $\rho(n)$, there is no polynomial-time $\rho(n)$-approximation for the general TSP.

Proof. We reduce from the Hamiltonian Cycle problem. Given a graph $G = (V, E)$ on n vertices, construct a complete graph K_n on the same vertex set with costs:

$$c(u, v) = \begin{cases} 1 & \text{if } \{u, v\} \in E, \\ n \cdot \rho(n) + 1 & \text{otherwise.} \end{cases}$$

If G has a Hamiltonian cycle, the optimal TSP tour has cost n. If G has no Hamiltonian cycle, any tour must use at least one non-edge, so the optimal tour has cost at least $n - 1 + n\rho(n) + 1 = n \cdot \rho(n) + n > \rho(n) \cdot n$.

A $\rho(n)$-approximation algorithm would produce a tour of cost at most $\rho(n) \cdot n$ when OPT $= n$, distinguishing the two cases. Since Hamiltonian Cycle is NP-complete, no such algorithm exists unless P = NP. □

Why is the general case so hopeless? The reduction above constructs instances where non-edges receive artificially enormous costs, creating a gap between "yes" and "no" instances that no algorithm can bridge. The triangle inequality prevents exactly this kind of construction: if there is a path of cost k between u and w, then the direct edge $c(u, w)$ can cost at most k. Formally, the metric TSP requires edge costs to satisfy:

$$c(u, w) \leq c(u, v) + c(v, w) \quad \text{for all } u, v, w \in V.$$

This restriction is natural in many applications (e.g., Euclidean distances). It does not, however, make the problem easy: metric TSP is still NP-hard. The same reduction from Hamiltonian Cycle works, only with a bounded cost gap: put $c(u, v) = 1$ if $\{u, v\} \in E$ and $c(u, v) = 2$ otherwise. Every cost now lies in $\{1, 2\}$, so $c(u, w) \leq 2 \leq c(u, v) + c(v, w)$ holds for all $u, v, w \in V$ and the instance is metric. If G has a Hamiltonian cycle, the optimal tour costs exactly n; otherwise every tour uses a non-edge and costs at least $n + 1$. An exact algorithm for metric TSP would thus decide Hamiltonian Cycle.

For approximation, the triangle inequality has a crucial algorithmic consequence: *shortcuts are free*. If a walk visits vertices $u, v_1, \ldots, v_k, w$, we can skip the intermediate vertices and go directly from u to w

without increasing the cost, since $c(u, w) \leq c(u, v_1) + c(v_1, v_2) + \cdots + c(v_k, w)$. This means that any closed walk through all vertices can be converted into a Hamiltonian cycle of no greater cost. So the problem reduces to: find a cheap closed walk that visits every vertex. This is easier than finding a Hamiltonian cycle directly, because walks are allowed to revisit vertices.

As in the vertex cover analysis from the previous lecture, we start by looking for a good lower bound on OPT. An optimal tour visits all n vertices and returns to the start. Removing one edge from this tour yields a spanning path, which is in particular a spanning tree. So every tour contains a spanning tree, and the minimum spanning tree gives a lower bound. Crucially, MSTs can be computed efficiently: sort the edges by cost and greedily add each edge that does not create a cycle (Kruskal's algorithm), which runs in $O(m \log n)$ time.

Lemma 2.1. Let T be a minimum spanning tree of K_n. Then $c(T) \leq \text{OPT}$.

Proof. Removing any edge from an optimal tour yields a spanning tree of cost at most OPT. Since T is a minimum spanning tree, $c(T)$ is at most the cost of any spanning tree, so $c(T) \leq \text{OPT}$. □

Now the strategy is clear: use the MST as a backbone and turn it into a closed walk (which we can then shortcut into a tour). A tree is not a closed walk, but we can make it one by traversing each edge twice, once in each direction. This gives a multigraph where every vertex has even degree, which admits an Eulerian tour. Such a tour can be found in linear time: start at any vertex and greedily follow unused edges until returning to the start; whenever the resulting circuit misses some edges, splice in a sub-circuit from a vertex that still has unused edges, and repeat until all edges are used.

Algorithm (MST-doubling):

1. Compute a minimum spanning tree T of K_n.
2. Double every edge of T to obtain a multigraph H where every vertex has even degree.
3. Find an Eulerian tour of H.
4. Shortcut the Eulerian tour: skip any vertex already visited, going directly to the next unvisited vertex.

Theorem 2.2. The MST-doubling algorithm is a 2-approximation for metric TSP.

Proof. The Eulerian tour of H has cost $c(H) = 2 \cdot c(T) \leq 2 \cdot \text{OPT}$ by Lemma 2.1. Shortcutting can only decrease the cost by the triangle inequality. Therefore, the final tour has cost at most $2 \cdot \text{OPT}$. □

The factor of 2 comes from doubling *every* edge of the MST. But why did we double edges in the first place? Only to make all degrees even, so that an Eulerian tour exists. If a vertex already has even degree, doubling its incident edges is wasteful. The vertices that actually need fixing are precisely those with *odd* degree. The key idea, due to Christofides [6] and independently Serdyukov [30], is to add edges only where needed: a minimum-weight perfect matching on the odd-degree vertices.

Algorithm (Christofides–Serdyukov):

1. Compute a minimum spanning tree T of K_n.
2. Let S be the set of vertices with odd degree in T.
3. Compute a minimum-weight perfect matching M on the complete graph induced by S.

4. Combine T and M to form a multigraph $H = T \cup M$.
5. Find an Eulerian tour of H and shortcut repeated vertices.

Why does a perfect matching on S exist? The sum of degrees in any graph is even, so the number of odd-degree vertices $|S|$ is even. Since we compute the matching on the *complete* graph induced by S, any pairing of the $|S|$ vertices into $|S|/2$ pairs gives a perfect matching. Finding a minimum-weight such matching is a nontrivial algorithmic problem: the main difficulty is that, unlike in bipartite graphs, augmenting paths in general graphs can traverse odd cycles, which Edmonds [33] resolved by "shrinking" these cycles (called *blossoms*) into single vertices. The resulting algorithm runs in $O(n^3)$ time. Adding the matching edges to T increases the degree of each vertex in S by exactly one, making all degrees even and enabling the Eulerian tour.

Lemma 2.2. The minimum-weight perfect matching M on S satisfies $c(M) \le \mathrm{OPT}/2$.

Proof. Consider an optimal TSP tour and restrict it to the vertices in S. Since the tour visits every vertex, we can shortcut it to obtain a tour on S alone. By the triangle inequality, this restricted tour has cost at most OPT.

Let $|S| = 2k$. The restricted tour on S visits $2k$ vertices in some order $s_1, s_2, \ldots, s_{2k}, s_1$. Partition the edges of this tour into two perfect matchings:

$$M_1 = \{\{s_1, s_2\}, \{s_3, s_4\}, \ldots, \{s_{2k-1}, s_{2k}\}\}$$

$$M_2 = \{\{s_2, s_3\}, \{s_4, s_5\}, \ldots, \{s_{2k}, s_1\}\}$$

Both M_1 and M_2 are perfect matchings on S, and their total cost satisfies $c(M_1) + c(M_2) \le \mathrm{OPT}$. Therefore, $\min(c(M_1), c(M_2)) \le \mathrm{OPT}/2$. Since M is a minimum-weight perfect matching, $c(M) \le \mathrm{OPT}/2$. □

Theorem 2.3. The Christofides–Serdyukov algorithm is a $\frac{3}{2}$-approximation for metric TSP.

Proof. The multigraph $H = T \cup M$ has cost $c(H) = c(T) + c(M)$. By Lemma 2.1, $c(T) \le \mathrm{OPT}$. By Lemma 2.2, $c(M) \le \mathrm{OPT}/2$. Therefore:

$$c(H) = c(T) + c(M) \le \mathrm{OPT} + \frac{\mathrm{OPT}}{2} = \frac{3}{2} \cdot \mathrm{OPT}$$

As in the MST-doubling analysis, shortcutting the Eulerian tour of H can only decrease the cost by the triangle inequality. The final tour has cost at most $\frac{3}{2} \cdot \mathrm{OPT}$. □

The $\frac{3}{2}$-approximation ratio stood as the best known for nearly fifty years. In 2020, Karlin, Klein, and Oveis Gharan [31] achieved a breakthrough by obtaining a $(\frac{3}{2} - \varepsilon_0)$-approximation for some small constant $\varepsilon_0 > 0$. On the hardness side, metric TSP is known to be APX-hard: there is no PTAS unless P = NP.

2.2 Knapsack

The knapsack problem is one of the most fundamental problems in combinatorial optimization. We are given n items, where item i has weight $w_i > 0$ and profit $p_i > 0$, along with a knapsack capacity W. The goal is to find a subset $S \subseteq \{1, \ldots, n\}$ maximizing $\sum_{i \in S} p_i$ subject to $\sum_{i \in S} w_i \le W$. We may assume

without loss of generality that $w_i \le W$ for all i, since items that do not fit individually can be discarded.

Subset sum is the special case where $w_i = p_i$ for all i and W is the target value. In the previous lecture, we gave an FPTAS for subset sum using a trimming technique. Here we present a different approach to an FPTAS for the more general knapsack problem, based on profit scaling. The idea is again quite simple: we already know how to solve knapsack *exactly* by dynamic programming, but the running time depends on the size of the profit values. If we could make these values smaller, the DP would be faster. Rounding profits down loses some precision, but if we control the rounding error, the solution remains close to optimal.

The standard DP for knapsack indexes by weight and runs in time $O(nW)$. For our purposes, we index by *profit* instead. Let $A[i][p]$ be the minimum weight to achieve profit exactly p using items from $\{1, \dots, i\}$. The recurrence is:

$$A[i][p] = \min\big(A[i-1][p],\ A[i-1][p-p_i] + w_i\big)$$

with base cases $A[0][0] = 0$ and $A[0][p] = \infty$ for $p > 0$. The optimal profit is the largest p such that $A[n][p] \le W$. Let $P = \sum_{i=1}^{n} p_i$ denote the total profit. The table has $O(nP)$ entries, so this DP runs in $O(nP)$ time, which is pseudo-polynomial: polynomial in the *values* of the profits but potentially exponential in the input size. Why index by profit rather than weight? Because we are about to *scale down* the profits, making P polynomial in n and $1/\varepsilon$. We have no such control over W. How much can we afford to round? Replacing each profit p_i by $\lfloor p_i/K \rfloor$ for some scaling factor K introduces a per-item error of at most K. Over at most n items, the total error is at most nK. For this to be at most $\varepsilon \cdot \mathrm{OPT}$, we need $nK \le \varepsilon \cdot \mathrm{OPT}$. We don't know OPT, but we do know that $\mathrm{OPT} \ge p_{\max}$ (since every item fits individually). Setting $K = \varepsilon \cdot p_{\max}/n$ guarantees $nK = \varepsilon \cdot p_{\max} \le \varepsilon \cdot \mathrm{OPT}$.

Algorithm (FPTAS for Knapsack):

1. Let $p_{\max} \leftarrow \max_i p_i$.
2. Set the scaling factor $K \leftarrow \frac{\varepsilon \cdot p_{\max}}{n}$.
3. For each item i, define the scaled profit $p'_i \leftarrow \left\lfloor \frac{p_i}{K} \right\rfloor$.
4. Run the exact DP on the scaled profits $p'_1, \dots, p'_n$ with the original weights $w_1, \dots, w_n$ and capacity W.
5. Return the subset found by the DP.

Theorem 2.4. For any $\varepsilon > 0$, the above algorithm is a $(1+\varepsilon)$-approximation for knapsack, running in time $O(n^3/\varepsilon)$.

Proof. Each scaled profit satisfies:

$$p'_i = \left\lfloor \frac{p_i}{K} \right\rfloor \le \frac{p_i}{K} \le \frac{p_{\max}}{K} = \frac{n}{\varepsilon}$$

The total scaled profit is $P' = \sum_{i=1}^{n} p'_i \le n^2/\varepsilon$. The DP runs in $O(nP') = O(n^3/\varepsilon)$ time, which is polynomial in n and $1/\varepsilon$.

For the approximation guarantee, let S^* be an optimal solution with profit $\mathrm{OPT} = \sum_{i \in S^*} p_i$. Since S^* satisfies the weight constraint, the DP on scaled profits finds a solution S with $\sum_{i \in S} p'_i \ge \sum_{i \in S^*} p'_i$.

For each item i, the floor satisfies $p'_i \ge p_i/K - 1$, so $p'_i \cdot K \ge p_i - K$. Therefore the true profit of S satisfies:

$$\sum_{i \in S} p_i \ge \sum_{i \in S} p'_i \cdot K \ge \sum_{i \in S^*} p'_i \cdot K \ge \sum_{i \in S^*} (p_i - K) = \mathrm{OPT} - |S^*| \cdot K$$

Since $|S^*| \le n$ and $K = \varepsilon \cdot p_{\max}/n$, the total rounding error is $|S^*| \cdot K \le \varepsilon \cdot p_{\max} \le \varepsilon \cdot \mathrm{OPT}$. Hence $\mathrm{ALG} \ge (1-\varepsilon) \cdot \mathrm{OPT}$. To obtain a $(1+\varepsilon)$-approximation, we run the algorithm with parameter $\varepsilon/2$ instead of ε. This gives $\mathrm{ALG} \ge (1-\varepsilon/2) \cdot \mathrm{OPT} \ge \mathrm{OPT}/(1+\varepsilon)$, since $(1-\varepsilon/2)(1+\varepsilon) = 1+\varepsilon/2-\varepsilon^2/2 \ge 1$ for $\varepsilon \le 1$. The running time becomes $O(n^3/(\varepsilon/2)) = O(n^3/\varepsilon)$. □

The profit-scaling technique, introduced by Ibarra and Kim [8], is conceptually simpler than the trimming approach we saw for subset sum. Both achieve an FPTAS, but scaling has the advantage of reducing to an existing exact algorithm as a black box: we simply change the input and run the same DP.

2.3 Load Balancing

We now turn to a scheduling problem. We have m identical machines and n jobs with processing times $t_1, t_2, \ldots, t_n$. Each job must be assigned to exactly one machine, and the load of a machine is the total processing time of jobs assigned to it. The objective is to minimize the *makespan*: the maximum load over all machines.

Formally, given an assignment $\sigma \colon \{1, \ldots, n\} \to \{1, \ldots, m\}$, the makespan is:

$$\max_{1 \le j \le m} \sum_{i : \sigma(i) = j} t_i$$

As before, we start by identifying lower bounds on the optimal makespan. Two are immediate.

Lemma 2.3. The optimal makespan satisfies:

$$\mathrm{OPT} \ge \frac{1}{m} \sum_{i=1}^{n} t_i \quad \text{and} \quad \mathrm{OPT} \ge \max_{1 \le i \le n} t_i$$

Proof. The total work $\sum_i t_i$ is distributed among m machines, so by averaging the most loaded machine has load at least $\frac{1}{m} \sum_i t_i$. The second bound holds because whichever machine processes the longest job has load at least $\max_i t_i$. □

The simplest scheduling heuristic is list scheduling, introduced by Graham [9]: process jobs one by one, always assigning the next job to the least loaded machine. This is the greedy algorithm that tries to keep loads balanced at every step.

Algorithm (List Scheduling):

1. Process jobs in the order $1, 2, \ldots, n$.
2. Assign each job to the machine with the current smallest load.

To analyze list scheduling, we focus on the bottleneck machine and the moment the last job lands on it. The makespan decomposes as the load L before that job plus the job's processing time t_ℓ. We can bound each term separately using our two lower bounds.

Theorem 2.5. List scheduling is a $(2 - 1/m)$-approximation for makespan minimization.

Proof. Let M_j be the machine with the maximum load at the end, let t_ℓ be the last job assigned to it, and let L be the load of M_j just before job ℓ was assigned. The makespan is $L + t_\ell$.

When job ℓ was assigned, M_j had the smallest load among all machines, so every machine had load at least L. The total load on all m machines at that point is $\sum_{i<\ell} t_i \le \sum_i t_i - t_\ell$, and since M_j has the smallest load, $m \cdot L \le \sum_i t_i - t_\ell$. Therefore:

$$L \le \frac{1}{m}\left(\sum_{i=1}^{n} t_i - t_\ell\right) \le \mathrm{OPT} - \frac{t_\ell}{m}$$

by Lemma 2.3. Also, $t_\ell \le \max_i t_i \le \mathrm{OPT}$ by the same lemma. Therefore:

$$\begin{aligned}\mathrm{ALG} = L + t_\ell &\le \mathrm{OPT} - \frac{t_\ell}{m} + t_\ell = \mathrm{OPT} + \left(1 - \frac{1}{m}\right) t_\ell \\ &\le \mathrm{OPT} + \left(1 - \frac{1}{m}\right)\mathrm{OPT} = \left(2 - \frac{1}{m}\right)\mathrm{OPT}\end{aligned}$$

This concludes the proof. □

The bound $(2 - 1/m)$ is tight for list scheduling. The worst case arises when a large job arrives last and gets piled onto an already loaded machine. Consider $m(m-1)$ jobs of size 1 followed by one job of size m. List scheduling distributes the small jobs evenly, giving each machine load $m - 1$, then assigns the large job to one of them, resulting in makespan $2m - 1$. The optimal schedule places the large job alone on one machine and distributes the small jobs among the remaining $m - 1$ machines, achieving makespan m. The ratio is $(2m-1)/m = 2 - 1/m$.

The problem is clear: list scheduling is oblivious to job sizes, so a large job can arrive last and ruin the balance. The fix is equally clear: process large jobs first, when the machines are still lightly loaded. Graham [10] observed that this simple modification leads to a significantly improved guarantee.

Algorithm (Longest Processing Time first, LPT):

1. Sort jobs so that $t_1 \ge t_2 \ge \cdots \ge t_n$.
2. Apply list scheduling in this order.

The key consequence of sorting is that the last job assigned to the bottleneck machine is now the *smallest* job involved, not an arbitrarily large one. This lets us replace the bound $t_\ell \le \mathrm{OPT}$ with the much tighter $t_\ell \le \mathrm{OPT}/3$.

Theorem 2.6. LPT is a $(4/3 - 1/(3m))$-approximation for makespan minimization.

Proof. If $n \le m$, each job gets its own machine and the makespan equals $\max_i t_i = \mathrm{OPT}$. So assume $n > m$.

Let M_j be the bottleneck machine and let t_ℓ be the last job assigned to it. If M_j received only one job, then $\mathrm{ALG} = t_\ell \le \mathrm{OPT}$ and we are done. Otherwise, M_j received at least two jobs, so $\ell \ge m + 1$ (the first m jobs are assigned to distinct machines), and therefore $t_\ell \le t_{m+1}$.

We distinguish two cases.

Case 1: $t_\ell \le \mathrm{OPT}/3$. Using the same bound on L as in Theorem 2.5:

$$\begin{aligned}\mathrm{ALG} = L + t_\ell &\le \mathrm{OPT} - \frac{t_\ell}{m} + t_\ell = \mathrm{OPT} + \left(1 - \frac{1}{m}\right) t_\ell \\ &\le \mathrm{OPT} + \left(1 - \frac{1}{m}\right) \frac{\mathrm{OPT}}{3} = \left(\frac{4}{3} - \frac{1}{3m}\right) \mathrm{OPT}\end{aligned}$$

Case 2: $t_\ell > \mathrm{OPT}/3$. Since the jobs are sorted, $t_1 \ge \cdots \ge t_\ell > \mathrm{OPT}/3$, so no machine of an optimal schedule holds three of the jobs $1, \ldots, \ell$; in particular $\ell \le 2m$. Consider the moment LPT assigns job ℓ to M_j (Figure 2.1, left). The $\ell - 1 \ge m$ jobs assigned so far occupy every machine, since LPT fills empty machines first. Let k be the number of machines holding exactly one of them. The other $m - k$ machines hold at least two, so $\ell - 1 \ge k + 2(m-k)$, that is, $k \ge 2m - \ell + 1 \ge 1$. Since M_j has the smallest load, each of these k singleton jobs has processing time at least L.

Now consider an optimal schedule (Figure 2.1, right). If each of the k singleton jobs were the only job among $1, \ldots, \ell$ on its machine, the remaining $\ell - k$ of these jobs would lie on the remaining $m - k$ machines, at most two per machine, giving $\ell - k \le 2(m-k)$, that is, $k \le 2m - \ell$. This contradicts $k \ge 2m - \ell + 1$. Hence some machine of the optimal schedule holds a singleton job, of processing time at least L, together with another job of index at most ℓ, of processing time at least t_ℓ. Therefore $\mathrm{OPT} \ge L + t_\ell = \mathrm{ALG}$.

In both cases, $\mathrm{ALG} \le (4/3 - 1/(3m)) \cdot \mathrm{OPT}$. □

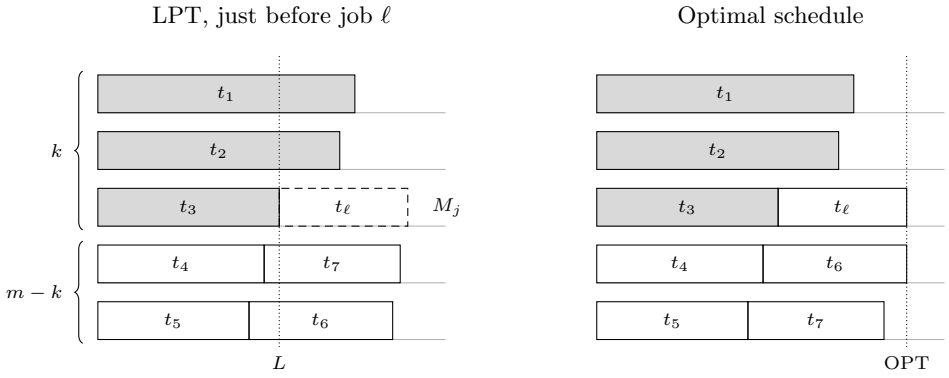

Figure 2.1: Case 2 in the proof of Theorem 2.6, with $m = 5$ and $\ell = 8$.

The 4/3 ratio of LPT is a significant improvement over the factor of 2 from plain list scheduling, obtained simply by sorting the input. For the load balancing problem, Hochbaum and Shmoys [32] showed that a PTAS exists: for any $\varepsilon > 0$, a $(1+\varepsilon)$-approximation can be found in polynomial time, though the algorithm is considerably more involved than the simple LPT rule.

September 14, 2026

Lecture 3: Approximation Algorithms III

In the previous two lectures, we designed approximation algorithms using combinatorial techniques: greedy algorithms, matchings, dynamic programming, and scaling. In this lecture, we add two more tools to our repertoire. First, we introduce linear programming (LP) relaxation: formulate the problem as an integer program, relax the integrality constraints, solve the resulting LP, and round the fractional solution. We illustrate this on weighted vertex cover and weighted set cover. Then, we study MAX-CUT via semidefinite programming (SDP), a powerful generalization of LP relaxation.

3.1 Linear Programming Relaxation

A linear program (LP) asks to optimize a linear objective function subject to linear constraints. In its standard form, an LP minimizes or maximizes ${}^t c\, x$ subject to $Ax \leq b$ and $x \geq 0$, where $x \in \mathbb{R}^n$. Linear programs can be solved in polynomial time, for example by interior point methods.

An integer linear program (ILP) is an LP with the additional constraint that the variables must take integer values: $x \in \mathbb{Z}^n$. In general, solving an ILP is NP-hard. For combinatorial optimization problems, the variables are usually binary ($x \in \{0,1\}^n$), and we focus on this case.

The key idea behind LP relaxation is to drop the integrality constraints, obtaining a problem that is easier to solve and whose optimal value provides a bound on the true optimum.

Definition 3.1: LP relaxation

Given an ILP with binary variables $x_i \in \{0,1\}$, its LP relaxation is obtained by replacing each integrality constraint with the continuous constraint $0 \leq x_i \leq 1$.

For a minimization problem, the LP relaxation has a larger feasible region (every integer solution is also a fractional solution), so $\mathrm{OPT}_{\mathrm{LP}} \leq \mathrm{OPT}$. For a maximization problem, $\mathrm{OPT}_{\mathrm{LP}} \geq \mathrm{OPT}$. In both cases, the LP optimum provides a bound on the true optimum that we can exploit algorithmically.

Definition 3.2: Integrality gap

The integrality gap of an LP relaxation is the worst-case ratio between the optimal integer solution and the optimal fractional solution, taken over all instances. For a minimization problem, it is $\sup_I \frac{\mathrm{OPT}(I)}{\mathrm{OPT}_{\mathrm{LP}}(I)}$; for a maximization problem, it is $\inf_I \frac{\mathrm{OPT}(I)}{\mathrm{OPT}_{\mathrm{LP}}(I)}$.

The integrality gap characterizes the power of a relaxation: no rounding scheme can achieve an approximation ratio better than the integrality gap, and conversely, if a rounding scheme achieves cost at most $\alpha \cdot \mathrm{OPT}_{\mathrm{LP}}$ on every instance (for a minimization problem), then it is an α-approximation algorithm, since $\mathrm{OPT}_{\mathrm{LP}} \leq \mathrm{OPT}$. The integrality gap is therefore exactly the best approximation ratio achievable by rounding a given relaxation.

Not all LP relaxations are useful. For example, the natural LP relaxation for maximum-weight independent set (maximize $\sum_v w_v x_v$ subject to $x_u + x_v \leq 1$ for all edges, $0 \leq x_v \leq 1$) has an integrality gap of $n/2$ on the complete graph K_n: the LP sets every $x_v = 1/2$ for a value of $n/2$, while the integer optimum is 1. No rounding scheme can overcome such a gap. The problems below have much better integrality gaps.

3.2 Weighted Vertex Cover

We now revisit the vertex cover problem, this time in its weighted version. Given an undirected graph $G = (V, E)$ and vertex weights $w_v \geq 0$ for each $v \in V$, the goal is to find a vertex cover $C \subseteq V$ (a set such that

every edge has at least one endpoint in C) of minimum total weight $w(C) = \sum_{v \in C} w_v$.

In Lecture 1, the edge-picking algorithm (Theorem 1.3) gave a 2-approximation for unweighted vertex cover. However, that combinatorial approach does not directly extend to the weighted case, since picking both endpoints of an edge may include a very heavy vertex unnecessarily. LP relaxation provides a clean way to handle weights.

We formulate the problem as an ILP. For each vertex $v \in V$, introduce a variable $x_v \in \{0, 1\}$ indicating whether v is in the cover. The ILP is:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w_v \, x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 && \text{for all } \{u, v\} \in E \\ & x_v \in \{0, 1\} && \text{for all } v \in V \end{aligned}$$

The LP relaxation replaces $x_v \in \{0, 1\}$ with $0 \leq x_v \leq 1$:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w_v \, x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 && \text{for all } \{u, v\} \in E \\ & 0 \leq x_v \leq 1 && \text{for all } v \in V \end{aligned} \tag{1}$$

Let x^* be an optimal solution to the LP relaxation. We round this fractional solution to an integer solution by a simple threshold rule.

Algorithm:

1. Solve the LP relaxation (1) to obtain an optimal solution x^*.
2. For each vertex $v \in V$: include v in C if and only if $x_v^* \geq 1/2$.
3. Return C.

Theorem 3.1. The LP rounding algorithm is a 2-approximation for weighted vertex cover.

Proof. We must verify two things: that C is a valid vertex cover, and that its weight is at most $2 \cdot \text{OPT}$.

For feasibility, consider any edge $\{u, v\} \in E$. The LP constraint gives $x_u^* + x_v^* \geq 1$. Therefore, at least one of x_u^* or x_v^* is at least $1/2$, so at least one of u or v is included in C. Hence C is a vertex cover.

For the cost bound, for every vertex $v \in C$, we have $x_v^* \geq 1/2$, so $1 \leq 2x_v^*$. Therefore:

$$w(C) = \sum_{v \in C} w_v \leq \sum_{v \in C} w_v \cdot 2x_v^* \leq 2 \sum_{v \in V} w_v \, x_v^* = 2 \cdot \text{OPT}_{\text{LP}} \leq 2 \cdot \text{OPT}$$

This completes the proof. □

We now show that the factor of 2 is tight for this LP relaxation, meaning no rounding scheme can do better.

Theorem 3.2. The integrality gap of the vertex cover LP relaxation (1) is exactly 2.

Proof. Theorem 3.1 shows that the integrality gap is at most 2. For the lower bound, consider the complete graph K_n with unit weights $w_v = 1$ for all v. The fractional solution $x_v^* = 1/2$ for all v is feasible: for every edge $\{u, v\}$, we have $x_u^* + x_v^* = 1 \geq 1$. Its cost is $\mathrm{OPT}_{\mathrm{LP}} \leq n/2$. On the other hand, any integer vertex cover must include at least $n - 1$ vertices (omitting at most one vertex), so $\mathrm{OPT} = n - 1$. The ratio $\mathrm{OPT}/\mathrm{OPT}_{\mathrm{LP}} \geq (n-1)/(n/2) = 2(n-1)/n$, which approaches 2 as $n \to \infty$. □

We remark that the vertex cover LP has a remarkable structural property: every basic feasible solution of the LP relaxation (1) is half-integral, meaning that every variable takes a value in $\{0, 1/2, 1\}$. This is a consequence of the total dual integrality of the constraint matrix and explains why the simple threshold $1/2$ works so well.

3.3 Weighted Set Cover

In the weighted set cover problem, we are given a universe $U = \{e_1, \ldots, e_n\}$ of n elements, a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of subsets of U, and a cost $c_j \geq 0$ for each set S_j. The goal is to find a subcollection of minimum total cost that covers every element, i.e., whose union is U. In Lecture 1, we showed that the greedy algorithm is an H_n-approximation (Theorem 1.4), and that this is essentially optimal [3]. LP relaxation gives a different, incomparable guarantee.

The *frequency* of an element e is the number of sets containing it: $f_e = |\{j : e \in S_j\}|$. Let $f = \max_e f_e$ be the maximum frequency.

We formulate the problem as an ILP with a variable $x_j \in \{0, 1\}$ for each set S_j:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} c_j \, x_j & & \\ \text{subject to} \quad & \sum_{j : e \in S_j} x_j \geq 1 & & \text{for all } e \in U \\ & x_j \in \{0, 1\} & & \text{for all } j \end{aligned} \tag{2}$$

The LP relaxation replaces $x_j \in \{0, 1\}$ with $0 \leq x_j \leq 1$. Let x^* be an optimal LP solution with cost $\mathrm{OPT}_{\mathrm{LP}} \leq \mathrm{OPT}$.

Algorithm:

1. Solve the LP relaxation to obtain x^*.
2. Include S_j in the cover if and only if $x_j^* \geq 1/f$.
3. Return the selected sets.

Theorem 3.3. The LP rounding algorithm is an f-approximation for set cover.

Proof. For feasibility, consider any element $e \in U$. The LP constraint gives $\sum_{j:e\in S_j} x_j^* \geq 1$. This sum has at most $f_e \leq f$ terms, so at least one of them satisfies $x_j^* \geq 1/f$. That set is included, and e is covered.

For the cost bound, every selected set has $x_j^* \geq 1/f$, so $1 \leq f \cdot x_j^*$. Therefore:

$$\sum_{j\in\text{cover}} c_j \leq \sum_{j\in\text{cover}} c_j \cdot f \cdot x_j^* \leq f \sum_{j=1}^{m} c_j\, x_j^* = f \cdot \text{OPT}_{\text{LP}} \leq f \cdot \text{OPT}$$

This concludes the proof. □

When f is a small constant, this gives a constant-factor approximation. Vertex cover is the special case where $U = E$, each set $S_v = \{e \in E : v \in e\}$ corresponds to a vertex, and every element has frequency exactly 2; the f-approximation recovers the factor of 2. When f is large, the greedy H_n-approximation from Lecture 1 is better.

We proved the greedy H_n-approximation combinatorially in Lecture 1. LP duality allows us to reinterpret that proof and determine the integrality gap of the set cover LP. Every LP has an associated *dual* program. For a primal LP in the form

$$\text{minimize } {}^t\!c\, x \text{ subject to } Ax \geq b,\ x \geq 0,$$

the dual is

$$\text{maximize } {}^t\!b\, y \text{ subject to } {}^t\!A\, y \leq c,\ y \geq 0.$$

The dual has one variable per primal constraint and one constraint per primal variable.

Theorem 3.4 (Weak duality). If x is primal feasible and y is dual feasible, then ${}^t\!c\, x \geq {}^t\!b\, y$.

Proof. Since ${}^t\!A\, y \leq c$ and $x \geq 0$, we have ${}^t\!c\, x \geq {}^t({}^t\!A\, y)\, x = {}^t\!y\, Ax$. Since $Ax \geq b$ and $y \geq 0$, we have ${}^t\!y\, Ax \geq {}^t\!y\, b = {}^t\!b\, y$. □

In other words, every dual feasible solution provides a lower bound on the primal optimum (for minimization problems). We do not need to solve the LP to use this: any feasible dual solution, however obtained, gives a certifiable lower bound.

The dual of the set cover LP (2) introduces a variable $y_e \geq 0$ for each element $e \in U$:

$$\begin{aligned} \text{maximize} \quad & \sum_{e\in U} y_e & & \\ \text{subject to} \quad & \sum_{e\in S_j} y_e \leq c_j & & \text{for all } j \\ & y_e \geq 0 & & \text{for all } e \in U \end{aligned}$$

This is a *packing* problem: assign nonnegative values to elements so that no set is overloaded, and maximize the total value. By weak duality, any feasible packing gives a lower bound: $\sum_e y_e \leq \text{OPT}$.

The greedy algorithm from Lecture 1 implicitly constructs a near-feasible dual solution. Recall that the greedy analysis assigns a price $\text{price}(e)$ to each element e when it is first covered: if e is covered by a set S_j that covers k new elements at that step, then $\text{price}(e) = c_j/k$. Setting $y_e = \text{price}(e)$, the total cost of the greedy solution equals $\sum_{e\in U} y_e$. This y is not dual feasible in general: for a set S_j, we may have

$\sum_{e \in S_j} y_e > c_j$. However, the greedy proof (Theorem 1.4) shows that $\sum_{e \in S_j} y_e \leq H_n \cdot c_j$ for every set S_j. Therefore, y/H_n is dual feasible, giving:

$$\frac{1}{H_n} \sum_{e \in U} y_e = \sum_{e \in U} \frac{y_e}{H_n} \leq \text{OPT}$$

Hence $\text{ALG} = \sum_e y_e \leq H_n \cdot \text{OPT}_{\text{LP}} \leq H_n \cdot \text{OPT}$. The worst-case approximation ratio is the same as in Lecture 1, but we learn something new: the integrality gap of the set cover LP is at least H_n. Indeed, the dual solution has value ALG, so $\text{OPT}_{\text{LP}} \geq \text{ALG}/H_n$ by weak duality, and there are instances where $\text{ALG} = H_n \cdot \text{OPT}$. Combined with the f-approximation from LP rounding (Theorem 3.3), this shows that the LP relaxation captures the approximability of set cover: the gap between the LP and integer optima is $\Theta(H_n)$ in the worst case, matching the best possible approximation ratio. This technique of building an approximately feasible dual solution and rescaling it is called *dual fitting*.

3.4 MAX-CUT

We now turn to a maximization problem. Given an undirected graph $G = (V, E)$ with nonnegative edge weights $w_{ij} \geq 0$, the MAX-CUT problem asks for a partition of V into two sets S and $\bar{S} = V \setminus S$ maximizing the total weight of edges crossing the cut: $w(S) = \sum_{\{i,j\} \in E,\, i \in S,\, j \notin S} w_{ij}$. We write $W = \sum_{\{i,j\} \in E} w_{ij}$ for the total edge weight; clearly $w(S) \leq W$ for every cut. MAX-CUT is NP-hard. Moreover, it is APX-hard: no PTAS exists unless P = NP.

A simple algorithm assigns each vertex to S independently with probability $1/2$.

Theorem 3.5. The randomized algorithm satisfies $\mathbb{E}[w(S)] \geq \frac{1}{2} W \geq \frac{1}{2} \cdot \text{OPT}$.

Proof. By linearity of expectation:

$$\mathbb{E}[w(S)] = \sum_{\{i,j\} \in E} w_{ij} \cdot \mathbb{P}[i \text{ and } j \text{ on different sides}] = \sum_{\{i,j\} \in E} w_{ij} \cdot \frac{1}{2} = \frac{W}{2}.$$

Since $\text{OPT} \leq W$, we have $\mathbb{E}[w(S)] \geq \frac{1}{2} \cdot \text{OPT}$. □

This can be derandomized by the *method of conditional expectations*. The idea is to fix the random choices one at a time, each time choosing the value that does not decrease the conditional expectation of the cut weight. The expectation is $W/2$ before anything is fixed, and once everything is fixed, the expectation is just the weight of the resulting cut, which is therefore at least $W/2$. The method works because these conditional expectations can be computed exactly.

Order the vertices as $1, \ldots, n$ and let $Z_i \in \{0, 1\}$ indicate whether vertex i is placed in S. Suppose the first k vertices have been assigned values $z_1, \ldots, z_k$. An edge with both endpoints among these vertices is cut or not, depending on whether $z_i \neq z_j$. An edge with at least one endpoint not yet assigned is cut with probability exactly $1/2$, since that endpoint is placed uniformly at random, independently of everything else. Hence

$$\mathbb{E}[w(S) \mid Z_1 = z_1, \ldots, Z_k = z_k] = \sum_{\substack{\{i,j\} \in E \\ i,j \leq k,\ z_i \neq z_j}} w_{ij} + \frac{1}{2} \sum_{\substack{\{i,j\} \in E \\ i > k \text{ or } j > k}} w_{ij}. \tag{3}$$

For the algorithm, we do not even need to evaluate this expression: only the difference between the two choices for the next vertex matters, and this difference involves only the edges to the vertices placed so far.

Algorithm (Greedy MAX-CUT):

1. Initialize $S \leftarrow \emptyset$ and $\bar{S} \leftarrow \emptyset$.
2. For $k = 1, \ldots, n$: let $a_k = \sum_{j \in S} w_{kj}$ and $b_k = \sum_{j \in \bar{S}} w_{kj}$ be the total weight of edges from vertex k to the vertices already in S and in $\bar{S}$, respectively. If $a_k \geq b_k$, add k to $\bar{S}$; otherwise, add k to S.
3. Return the cut $(S, \bar{S})$.

Theorem 3.6. The greedy algorithm returns a cut of weight at least $W/2 \geq \mathrm{OPT}/2$ in time $O(n+m)$.

Proof. Let $z_1, \ldots, z_n$ be the values chosen by the algorithm, and let

$$E_k = \mathbb{E}[w(S) \mid Z_1 = z_1, \ldots, Z_k = z_k]$$

be the conditional expectation of the random cut weight given that the first k vertices are placed as the algorithm placed them. Then $E_0 = \mathbb{E}[w(S)] = W/2$ by Theorem 3.5, and E_n is the weight of the returned cut, since conditioning on all n values leaves nothing random. It therefore suffices to show $E_k \geq E_{k-1}$ for every k.

Fix $z_1, \ldots, z_{k-1}$. Since Z_k is uniform on $\{0,1\}$ and independent of $Z_1, \ldots, Z_{k-1}$, the law of total expectation gives

$$\begin{aligned} E_{k-1} = {} & \frac{1}{2}\,\mathbb{E}[w(S) \mid Z_1 = z_1, \ldots, Z_{k-1} = z_{k-1}, Z_k = 0] \\ & + \frac{1}{2}\,\mathbb{E}[w(S) \mid Z_1 = z_1, \ldots, Z_{k-1} = z_{k-1}, Z_k = 1]. \end{aligned}$$

An average of two numbers is at most their maximum, so if z_k is chosen to maximize the conditional expectation, then $E_k \geq E_{k-1}$. It remains to check that the algorithm makes this choice. By (3), the two conditional expectations differ only in the edges between k and $\{1, \ldots, k-1\}$: all other edges either do not involve k or have an endpoint beyond k, and contribute the same in both cases. Placing k in $\bar{S}$ cuts exactly the edges from k to S, of total weight a_k, and placing k in S cuts exactly those to $\bar{S}$, of total weight b_k. The algorithm chooses the larger of the two, which is the maximizing choice.

For the running time, each edge $\{j, k\}$ with $j < k$ is examined exactly once, when vertex k is placed. This concludes the proof. □

Alternatively, any locally optimal cut (where no single vertex move improves the cut) has weight at least $W/2$: local optimality means that each vertex has at least half its weighted degree on cut edges, and summing over all vertices gives $w(S) \geq W/2$.

We can formulate MAX-CUT as an ILP: for each edge $\{i, j\}$, introduce $y_{ij} \in \{0,1\}$ indicating whether it is cut, and for each vertex i, introduce $x_i \in \{0,1\}$ indicating its side. An edge is cut when its endpoints are on different sides:

$$\begin{aligned} \text{maximize} \quad & \sum_{\{i,j\} \in E} w_{ij}\, y_{ij} \\ \text{subject to} \quad & y_{ij} \leq x_i + x_j, \quad y_{ij} \leq 2 - x_i - x_j && \text{for all } \{i,j\} \in E \\ & x_i \in \{0,1\},\ y_{ij} \in \{0,1\} && \text{for all } i, \{i,j\} \end{aligned}$$

The LP relaxation replaces integrality with $0 \leq x_i, y_{ij} \leq 1$. Unfortunately, this relaxation is too weak: on any graph, setting $x_i = 1/2$ for all i and $y_{ij} = 1$ for all edges is feasible, with LP value W. Since $\mathrm{OPT} \leq W$, the LP always achieves $\mathrm{OPT}_{\mathrm{LP}} = W$. On the complete graph K_n with unit weights, $\mathrm{OPT} \approx n^2/4$ while $W = \binom{n}{2} \approx n^2/2$, so the integrality gap is $1/2$—no better than the trivial randomized algorithm. LP relaxation provides no advantage for MAX-CUT.

To break the 1/2 barrier, Goemans and Williamson [34] used a stronger relaxation based on *semidefinite programming* (SDP). The starting point is a different integer formulation, better suited to relaxation. Instead of $x_i \in \{0,1\}$, assign $z_i \in \{-1,+1\}$ to each vertex: $S = \{i : z_i = +1\}$. An edge $\{i,j\}$ is cut if and only if $z_i \neq z_j$, which happens when $z_i z_j = -1$. The objective becomes:

$$\text{maximize} \quad \sum_{\{i,j\} \in E} w_{ij} \cdot \frac{1 - z_i z_j}{2}$$

subject to $z_i \in \{-1,+1\}$ for all i.

The SDP relaxation replaces each scalar z_i with a unit vector $v_i \in \mathbb{R}^n$, and each product $z_i z_j$ with the inner product $v_i \cdot v_j$. The constraint $z_i^2 = 1$ becomes $\|v_i\| = 1$. The relaxation is:

$$\begin{aligned} \text{maximize} \quad & \sum_{\{i,j\} \in E} w_{ij} \cdot \frac{1 - v_i \cdot v_j}{2} \\ \text{subject to} \quad & \|v_i\| = 1 \qquad \text{for all } i \in V \end{aligned} \tag{4}$$

This is a relaxation: every $z \in \{-1,+1\}^n$ yields the feasible solution $v_i = z_i e_1$, where e_1 is the first standard basis vector, with the same objective value, since $v_i \cdot v_j = z_i z_j$. Hence $\mathrm{OPT}_{\mathrm{SDP}} \geq \mathrm{OPT}$. The dimension n of the vectors is not a restriction either: n vectors always span a space of dimension at most n.

Optimizing over vectors looks like a nonlinear, nonconvex problem. The reason it is nonetheless tractable is that the objective and the constraints depend on the vectors only through their pairwise inner products $v_i \cdot v_j$, and the set of all possible inner product matrices has a clean description. To state it, we need some linear algebra.

A symmetric matrix $Y \in \mathbb{R}^{n \times n}$ is *positive semidefinite*, written $Y \succeq 0$, if ${}^t x Y x \geq 0$ for all $x \in \mathbb{R}^n$. For a symmetric matrix Y, the following are equivalent:

1. $Y \succeq 0$;
2. all eigenvalues of Y are nonnegative;
3. $Y = {}^t V V$ for some matrix $V \in \mathbb{R}^{d \times n}$, i.e., $Y_{ij} = v_i \cdot v_j$ is the *Gram matrix* of the columns $v_1, \ldots, v_n \in \mathbb{R}^d$ of V.

Indeed, if $Y = {}^t V V$, then ${}^t x Y x = {}^t(Vx)(Vx) = \|Vx\|^2 \geq 0$, so (3) implies (1). Testing (1) on an eigenvector x with eigenvalue λ gives $\lambda \|x\|^2 = {}^t x Y x \geq 0$, so (1) implies (2). Finally, a symmetric matrix has a spectral decomposition $Y = Q \Lambda {}^t Q$ with Q orthogonal and Λ the diagonal matrix of eigenvalues; if these are nonnegative, then $V = \Lambda^{1/2} {}^t Q$ satisfies ${}^t V V = Y$, so (2) implies (3). In practice, a factorization $Y = {}^t L L$ with L upper triangular is computed by the Cholesky decomposition in $O(n^3)$ time.

The positive semidefinite matrices form a convex cone: if $Y, Y' \succeq 0$ and $\lambda, \mu \geq 0$, then $\lambda Y + \mu Y' \succeq 0$, directly from the definition. A semidefinite program optimizes a linear function of the entries of Y over this cone intersected with finitely many linear equations.

Definition 3.3: Semidefinite program

A semidefinite program (SDP) is an optimization problem of the form

$$\begin{aligned} \text{maximize} \quad & \langle C, Y \rangle \\ \text{subject to} \quad & \langle A_k, Y \rangle = b_k \qquad \text{for } k = 1, \ldots, p \\ & Y \succeq 0 \end{aligned} \tag{5}$$

where $C, A_1, \ldots, A_p \in \mathbb{R}^{n \times n}$ are symmetric matrices, $b \in \mathbb{R}^p$, and $\langle A, Y \rangle = \sum_{i,j} A_{ij} Y_{ij} = \mathrm{tr}({}^t A Y)$ is the entrywise inner product of matrices.

An SDP is thus a linear program whose variable is a matrix, with the constraint $x \geq 0$ on a vector replaced by the constraint $Y \succeq 0$ on a matrix. It is a genuine generalization of LP: if C and all A_k are diagonal, only the diagonal of Y matters, and a diagonal matrix is positive semidefinite if and only if its diagonal entries are nonnegative, so (5) is an LP in the variables $Y_{11}, \dots, Y_{nn}$. Inequality constraints $\langle A_k, Y \rangle \leq b_k$ can be added by introducing slack variables on the diagonal of an enlarged matrix. Unlike an LP, the feasible region of an SDP is not a polyhedron: for instance, $\left(\begin{smallmatrix} x & y \\ y & z \end{smallmatrix}\right) \succeq 0$ if and only if $x, z \geq 0$ and $xz \geq y^2$, a region with a curved boundary. It is, however, convex, and this is what makes SDPs tractable: for every $\varepsilon > 0$, a feasible solution of value at least $\mathrm{OPT}_{\mathrm{SDP}} - \varepsilon$ can be computed in time polynomial in the input size and in $\log(1/\varepsilon)$, for example by the ellipsoid method or by interior point methods. Exact solutions are out of reach in general, since the optimum of an SDP with rational data may be irrational. In the rounding algorithm below, the additive loss of ε is absorbed into the approximation ratio, and we ignore it from now on.

We can now write the vector program (4) as an SDP. Let Y be the Gram matrix of the vectors, $Y_{ij} = v_i \cdot v_j$. The constraint $\|v_i\| = 1$ becomes $Y_{ii} = 1$, and by the equivalence above, a symmetric matrix Y is the Gram matrix of some vectors if and only if $Y \succeq 0$. So the vector program is equivalent to:

$$\begin{aligned} \text{maximize} \quad & \sum_{\{i,j\} \in E} w_{ij} \cdot \frac{1 - Y_{ij}}{2} \\ \text{subject to} \quad & Y_{ii} = 1 \qquad\qquad \text{for all } i \in V \\ & Y \succeq 0 \end{aligned}$$

This is of the form (5): the constraint $Y_{ii} = 1$ is $\langle e_i \, {}^t e_i, Y \rangle = 1$, and using $Y_{ii} = Y_{jj} = 1$, the objective can be written as $\frac{1}{4} \sum_{\{i,j\} \in E} w_{ij}(Y_{ii} + Y_{jj} - 2Y_{ij}) = \frac{1}{4}\langle L_G, Y \rangle$, where $L_G = D - A_G$ is the weighted Laplacian of G, with A_G the weighted adjacency matrix and D the diagonal matrix of weighted degrees. To recover vectors for rounding from an optimal matrix Y^*, we compute the Cholesky decomposition $Y^* = {}^t L \, L$; the columns of L are unit vectors $v_1^*, \dots, v_n^*$ with $v_i^* \cdot v_j^* = Y_{ij}^*$.

Let $v_1^*, \dots, v_n^*$ be an optimal SDP solution with value $\mathrm{OPT}_{\mathrm{SDP}} \geq \mathrm{OPT}$. The rounding step is a single random choice: a hyperplane through the origin, which partitions the vertices.

Goemans–Williamson algorithm:

1. Solve the SDP relaxation (4) to obtain unit vectors $v_1^*, \dots, v_n^*$.
2. Choose a uniformly random unit vector r in the same space as the v_i^*.
3. Set $S = \{i \in V : v_i^* \cdot r \geq 0\}$.
4. Return the cut $(S, \bar{S})$.

The analysis rests on a geometric lemma: the probability that a random hyperplane separates two unit vectors equals the angle between them divided by π.

Lemma 3.1. For unit vectors $v_i, v_j \in \mathbb{R}^d$ and a uniformly random unit vector $r \in \mathbb{R}^d$,

$$\mathbb{P}[\mathrm{sign}(v_i \cdot r) \neq \mathrm{sign}(v_j \cdot r)] = \frac{\arccos(v_i \cdot v_j)}{\pi}.$$

Proof. The random hyperplane $\{x : r \cdot x = 0\}$ separates v_i and v_j if and only if it passes between them. Restricting to the two-dimensional plane spanned by v_i and v_j, the hyperplane intersects this plane in a random line through the origin. If the angle between v_i and v_j is $\theta = \arccos(v_i \cdot v_j)$, the line separates them if and only if it falls in one of the two arcs of angle θ (see Figure 3.1), which happens with probability $2\theta/(2\pi) = \theta/\pi$. □

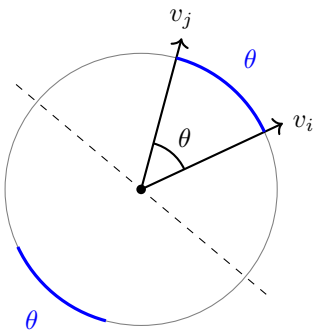

Figure 3.1: The random hyperplane (dashed line through the origin) separates v_i and v_j if and only if it passes through one of the two arcs of angle θ (shown in blue). This occurs with probability $2\theta/(2\pi) = \theta/\pi$.

Theorem 3.7 (Goemans–Williamson). The algorithm above achieves

$$\mathbb{E}[w(S)] \geq \alpha \cdot \mathrm{OPT}, \quad \text{where } \alpha = \min_{0 \leq \theta \leq \pi} \frac{2}{\pi} \cdot \frac{\theta}{1 - \cos\theta} \geq 0.878.$$

Proof. By linearity of expectation and Lemma 3.1:

$$\mathbb{E}[w(S)] = \sum_{\{i,j\} \in E} w_{ij} \cdot \frac{\arccos(v_i^* \cdot v_j^*)}{\pi}$$

We compare this to the SDP value, which uses $(1 - v_i^* \cdot v_j^*)/2$ per edge. For each edge $\{i, j\}$, let $\theta_{ij} = \arccos(v_i^* \cdot v_j^*)$, so $v_i^* \cdot v_j^* = \cos\theta_{ij}$. We need to show $\frac{\theta}{\pi} \geq \alpha \cdot \frac{1-\cos\theta}{2}$ for all $\theta \in [0, \pi]$, i.e., $\frac{2\theta}{\pi(1-\cos\theta)} \geq \alpha$.

The function $g(\theta) = \frac{2\theta}{\pi(1-\cos\theta)}$ is continuous on $(0, \pi]$, with $g(\theta) \to \infty$ as $\theta \to 0^+$ and $g(\pi) = 1$. Since g is continuous and tends to $+\infty$ at 0, it attains its minimum α on $(0, \pi]$. Numerical computation gives $\alpha \approx 0.878$.

Therefore $\mathbb{E}[w(S)] \geq \alpha \sum_{\{i,j\} \in E} w_{ij} \cdot \frac{1 - v_i^* \cdot v_j^*}{2} = \alpha \cdot \mathrm{OPT}_{\mathrm{SDP}} \geq \alpha \cdot \mathrm{OPT}$. □

The improvement from 0.5 to 0.878 comes entirely from the stronger relaxation: the SDP bound $\mathrm{OPT}_{\mathrm{SDP}}$ is much tighter than the LP bound $\mathrm{OPT}_{\mathrm{LP}} = W$. The SDP relaxation has an integrality gap of exactly $\alpha \approx 0.878$.

The Goemans–Williamson result was the first to show that semidefinite programming gives strictly better approximation ratios than linear programming. Assuming the Unique Games Conjecture [4], the ratio $\alpha \approx 0.878$ is the best possible for MAX-CUT in polynomial time.

September 21, 2026

Lecture 4: Approximation Algorithms IV

This last lecture on approximation algorithms treats two topics. The first is clustering: given n points and a number k, group the points into k clusters so that each point is close to the center of its cluster. Clustering problems are among the most common optimization problems in practice, and the k-center problem admits a particularly clean greedy approximation with a matching hardness bound. The second topic is a general design paradigm we have not yet formalized: *local search*, which starts from an arbitrary solution and repeatedly applies small improving modifications. We analyze local search on two problems: MAX-CUT, where it derandomizes the random cut of Lecture 3, and k-median, where a careful accounting of swaps yields a constant-factor approximation.

4.1 k-Center

In the *k-center* problem, we are given the metric space (X, d) of $|X| = n$ points and an integer $k \geq 1$, and we must choose a set $S \subseteq X$ of at most k *centers* minimizing the covering radius

$$r(S) = \max_{x \in X} d(x, S) \ .$$

Equivalently, we look for the smallest radius r such that k balls of radius r cover all of X.

The following greedy algorithm is due to Gonzalez [13] and is often called *farthest-first traversal*.

Algorithm: Farthest-First Traversal

1. Pick an arbitrary point $c_1 \in X$.
2. For $i = 2$ to k: pick $c_i \in X$ maximizing $d\big(c_i, \{c_1, \dots, c_{i-1}\}\big)$.
3. Return $S = \{c_1, \dots, c_k\}$.

The algorithm greedily attacks the currently worst-served point. Its analysis rests on a simple dichotomy: either the returned radius is already small, or the algorithm has found $k+1$ points that are pairwise far apart, which is impossible if a good solution exists.

Theorem 4.1. Farthest-first traversal is a 2-approximation algorithm for k-center and runs in $O(nk)$ time.

Proof. Let $r = r(S)$ be the radius of the returned solution, attained by some point v with $d(v, S) = r$, and let r^* denote the optimal radius. Consider the $k+1$ points $c_1, \dots, c_k, v$. We claim that they are pairwise at distance at least r. For $j < i \leq k$:

$$d(c_i, c_j) \ \geq \ d\big(c_i, \{c_1, \dots, c_{i-1}\}\big) \ \geq \ d\big(v, \{c_1, \dots, c_{i-1}\}\big) \ \geq \ d(v, S) \ = \ r \ ,$$

where the second inequality holds because c_i was chosen to maximize the distance to the first $i-1$ centers, and the third because distances to a set can only decrease when the set grows. Moreover, $d(v, c_i) \geq d(v, S) = r$ for every i.

Now fix an optimal solution with centers $s_1^*, \dots, s_k^*$; its balls of radius r^* cover X. By the pigeonhole principle, two of our $k+1$ points, say x and y, lie in the ball of the same optimal center s^*. By the triangle inequality,

$$r \ \leq \ d(x, y) \ \leq \ d(x, s^*) + d(s^*, y) \ \leq \ 2r^* \ .$$

For the running time, maintain for every point its distance to the current center set; each of the k

iterations updates these n values with one distance evaluation each. □

The analysis is tight already for $k = 1$. Let $X = \{0, 1, \ldots, 2r\}$ with $d(i, j) = |i-j|$, the vertices of a path with unit edge lengths. The optimal center is the midpoint r, with radius $r^* = r$, whereas the algorithm may pick the endpoint $c_1 = 0$, with radius $2r$. For general k, place k copies of this path at mutual distance much larger than r: the algorithm can pick an endpoint in each copy, since after choosing one center in a copy, the farthest points lie in the copies not yet visited.

Theorem 4.2 (Hsu, Nemhauser [14]). For every $\varepsilon > 0$, there is no polynomial-time $(2-\varepsilon)$-approximation algorithm for k-center unless P = NP.

Proof. We reduce from the NP-complete *dominating set* problem: given a graph $G = (V, E)$ and an integer k, decide whether there is a set $D \subseteq V$ with $|D| \leq k$ such that every vertex is in D or adjacent to a vertex of D.

Given (G, k), define a metric on $X = V$ by $d(u, v) = 1$ if $\{u, v\} \in E$ and $d(u, v) = 2$ if $u \neq v$ and $\{u, v\} \notin E$. The triangle inequality holds because all distances are 1 or 2. If G has a dominating set D of size at most k, then $r(D) \leq 1$, so the optimal k-center radius is 1. If G has no such dominating set, then every set of k centers leaves some point at distance 2, so the optimal radius is 2.

A $(2-\varepsilon)$-approximation algorithm returns, in the first case, a solution of radius at most $(2-\varepsilon)\cdot 1 < 2$, hence of radius 1, which is a dominating set of size at most k. In the second case, every solution has radius 2. Comparing the returned radius to 2 therefore decides dominating set in polynomial time. □

4.2 Local Search

A *local search* algorithm is specified by a set of feasible solutions and, for each solution, a *neighborhood* of solutions reachable by a small modification. The algorithm starts at an arbitrary feasible solution and repeatedly moves to a neighbor of better objective value, until no neighbor improves on the current solution; such a solution is called a *local optimum*. Two questions decide whether this is a good algorithm: how good are local optima compared to global ones, and how many improving moves can occur?

We first revisit MAX-CUT in the notation of Lecture 3: given an undirected graph $G = (V, E)$ with edge weights $w_{ij} \geq 0$ and total weight W, find a cut $(S, \bar{S})$ maximizing the weight $w(S)$ of the crossing edges. In Lecture 3, we showed that a uniformly random cut has expected weight at least $W/2$ (Theorem 3.5), derandomized this by conditional expectations, and remarked that every locally optimal cut has weight at least $W/2$ as well. We now prove this remark as a first instance of the paradigm, and, more importantly for local search, bound the number of improving moves. Take as neighborhood the *flip* of a single vertex to the other side.

Theorem 4.3. Every local optimum $(S, \bar{S})$ of flip-based local search for MAX-CUT satisfies $w(S) \geq W/2$. On unweighted graphs, a local optimum is reached after at most $|E|$ flips.

Proof. Let $(S, \bar{S})$ be a local optimum. For a vertex v, let $\mathrm{cut}(v)$ and $\mathrm{uncut}(v)$ denote the total weight of the edges at v that do and do not cross the cut. Flipping v changes the cut weight by $\mathrm{uncut}(v) - \mathrm{cut}(v)$, so local optimality means $\mathrm{cut}(v) \geq \mathrm{uncut}(v)$ for every v. Summing over all vertices counts every edge

twice:

$$2\,w(S) = \sum_{v \in V} \mathrm{cut}(v) \geq \sum_{v \in V} \mathrm{uncut}(v) = 2\,\big(W - w(S)\big) \ ,$$

so $w(S) \geq W/2$. On unweighted graphs, every flip increases the number of cut edges by at least 1, and this number is bounded by $|E|$, so at most $|E|$ flips occur. □

The guarantee of Theorem 4.3 concerns the quality of *every* local optimum; nothing needed to be known about how the local search reaches one. This separation of concerns is typical for local search analyses, including the one we do next.

The running-time argument, on the other hand, breaks down for weighted graphs: each flip improves the cut weight, but possibly by a tiny amount, and there are weighted instances on which local search performs an exponential number of flips. The standard remedy is to accept a move only if it improves the objective by a fixed factor. Then the objective moves geometrically, and the number of moves is logarithmic in the ratio of the largest to the smallest possible objective value.

Lemma 4.1. Let $0 < \delta \leq 1$. Consider a local search that moves from S to a neighbor S' only if $\mathrm{cost}(S') \leq (1-\delta)\,\mathrm{cost}(S)$, for a minimization problem, or only if $w(S') \geq (1+\delta)\,w(S)$, for a maximization problem. If the objective is at most C initially and every nonzero objective value is at least c, then the number of moves is at most $2\ln(C/c)/\delta + 1$.

Proof. For minimization, after t moves the cost is at most $(1-\delta)^t C \leq e^{-\delta t} C$. If $t > \ln(C/c)/\delta$, this is less than c, so the cost is zero and no further move is possible. For maximization, start the count at the first solution with nonzero objective, at least c; after t moves the objective is at least $(1+\delta)^t c$ and at most C, so $t \leq \ln(C/c)/\ln(1+\delta) \leq 2\ln(C/c)/\delta$, using $\ln(1+\delta) \geq \delta/2$ for $\delta \leq 1$. □

For MAX-CUT, start from a cut separating the endpoints of a heaviest edge, so that $c = w_{\max}$ and $C = W \leq |E|\,w_{\max}$, and flip only if the cut weight grows by a factor at least $1+\delta$ with $\delta = \varepsilon/n$. By Lemma 4.1, at most $O\big(\frac{n}{\varepsilon}\log|E|\big)$ flips occur. The price is paid in the guarantee. At the end, $\mathrm{uncut}(v) - \mathrm{cut}(v) \leq \delta\,w(S)$ for every v, and summing over v as in the proof of Theorem 4.3 gives $2W - 4w(S) \leq n\delta\,w(S) = \varepsilon\,w(S)$, that is,

$$w(S) \ \geq \ \frac{W}{2} \cdot \frac{1}{1+\varepsilon/4} \ \geq \ \Big(\frac{1}{2} - \frac{\varepsilon}{8}\Big) W \ .$$

The same device will make the k-median local search below run in polynomial time.

4.3 k-Median

The *k-median* problem is the sum-version of k-center: choose $S \subseteq X$ with $|S| = k$ minimizing

$$\mathrm{cost}(S) = \sum_{x \in X} d(x, S) \ .$$

Compared to k-center, the objective is far more robust: a single outlier point moves the k-center optimum arbitrarily, but shifts the k-median objective by only its own distance. The price of robustness is that no analog of the clean greedy algorithm of Theorem 4.1 is known; the natural algorithms for k-median are based on linear programming or on local search. We analyze the simplest local search, due to Arya, Garg, Khandekar, Meyerson, Munagala, and Pandit [16], with the streamlined proof of Gupta and Tangwongsan.

The neighborhood is given by *swaps*: from a solution S, remove one center $s \in S$ and add one non-center $o \notin S$.

Algorithm (Local search with swaps):

1. Start with an arbitrary set $S \subseteq X$ of k centers.
2. While there are $s \in S$ and $o \notin S$ with $\mathrm{cost}(S - s + o) < \mathrm{cost}(S)$: replace S by $S - s + o$.
3. Return S.

The algorithm returns a solution that no single swap can improve. We call S *swap-optimal* if

$$\mathrm{cost}\big(S - s + o\big) \geq \mathrm{cost}(S) \qquad \text{for all } s \in S,\ o \notin S\ ,$$

where $S - s + o$ abbreviates $(S \setminus \{s\}) \cup \{o\}$.

Fix a swap-optimal solution S and an optimal solution O, both of size k. For a point x, let $d_S(x) = d(x, S)$ and $d_O(x) = d(x, O)$ be its distances to the two solutions, let $s(x) \in S$ and $o(x) \in O$ be centers attaining them, and let $N_S(s) = \{x : s(x) = s\}$ and $N_O(o) = \{x : o(x) = o\}$ be the clusters of S and O.

The proof plan is this: for suitable pairs (s, o), swap-optimality gives an inequality $0 \leq \mathrm{cost}(S - s + o) - \mathrm{cost}(S)$, and we upper-bound the right-hand side by exhibiting an explicit assignment of the points to the centers of $S - s + o$. Summing these inequalities over a well-chosen set of k swaps will produce $\mathrm{cost}(S) \leq 5 \cdot \mathrm{cost}(O)$.

Which assignment? The points of $N_O(o)$ are easy: send them to the new center o, at their optimal cost $d_O(x)$. Points that keep their center cost nothing extra. The problem is the points of $N_S(s)$ that are not in $N_O(o)$: they lose their center s, and we need a new center of S for them that is provably not much farther than s was. The optimal center $o(x)$ of such a point is close to x, but it is not available, since it is not in S. What is available is the center $\eta(o(x))$ of S closest to $o(x)$, and the detour $x \to o(x) \to \eta(o(x))$ can be charged to the optimal solution: the first leg costs $d_O(x)$, and the second leg is at most $d(o(x), s(x))$, since $s(x)$ is one of the candidates for $\eta(o(x))$, which in turn is at most $d_O(x) + d_S(x)$ by the triangle inequality. The next lemma records this.

Lemma 4.2. Let $\eta(o)$ denote a center of S closest to $o \in O$. Then, for every point x,

$$d\big(x, \eta(o(x))\big) \ \leq\ 2d_O(x) + d_S(x)\ .$$

Proof. By the triangle inequality and the definition of η,

$$d\big(o(x), \eta(o(x))\big) \ \leq\ d\big(o(x), s(x)\big) \ \leq\ d\big(o(x), x\big) + d\big(x, s(x)\big) \ =\ d_O(x) + d_S(x)\ .$$

Hence $d\big(x, \eta(o(x))\big) \leq d\big(x, o(x)\big) + d\big(o(x), \eta(o(x))\big) \leq d_O(x) + (d_O(x) + d_S(x))$. □

So each rerouted point costs at most $2d_O(x)$ more than before, an amount the optimal solution can pay for. This leaves the choice of the swaps. The obvious candidates are the k swaps $(\eta(o), o)$ for $o \in O$: each optimal center enters once, replacing the center of S that stands in for it. Two things can go wrong. First, the reroute of a point $x \in N_S(s)$ presupposes that $\eta(o(x)) \neq s$; if several optimal centers o, o' share the same $\eta(o) = \eta(o') = s$, then in the swap (s, o) the points of $N_S(s)$ that belong to $N_O(o')$ would be sent to $\eta(o') = s$, the very center being removed. Second, if the same center s is swapped out in many pairs, the charge $2d_O(x)$ on its cluster $N_S(s)$ is collected many times over, and the sum of the inequalities no longer compares to $\mathrm{cost}(O)$. The following lemma shows that both problems can be avoided at once, at the price of letting a center of S appear in two pairs rather than one. If η happens to be a bijection, the pairs $(\eta(o), o)$ satisfy all three conditions with every s appearing once, and the computation below gives $\mathrm{cost}(S) \leq 3\,\mathrm{cost}(O)$; the factor 5 is the price of the general case.

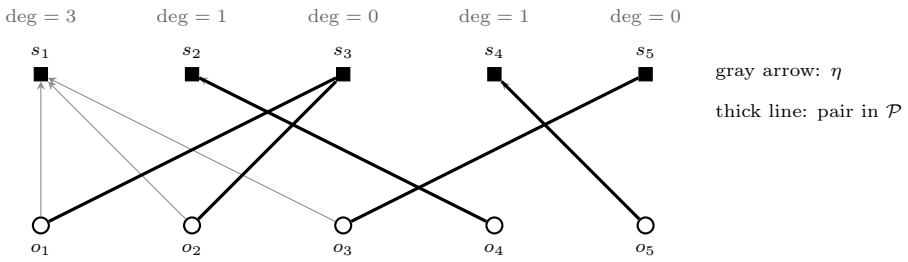

Figure 4.1: The pairing of Lemma 4.3 for $k = 5$. Squares are the centers of S, circles those of O, and the arrows show η, so the degree of a center of S is the number of arrows into it; here $n_0 = 2$, $n_1 = 2$, $n_2 = 1$. The centers s_2 and s_4 of degree 1 are paired with their unique preimages. The $k - n_1 = 3$ remaining optimal centers o_1, o_2, o_3, all mapped to s_1, are matched to the $n_0 = 2$ centers of degree 0, at most two each. No paired center of S receives an arrow from an optimal center other than its partner, which is condition 3, and s_1 is never swapped out.

Lemma 4.3. There is a set $\mathcal{P}$ of k swap pairs $(s, o) \in S \times O$ such that:

1. every $o \in O$ appears in exactly one pair;
2. every $s \in S$ appears in at most two pairs;
3. if $(s, o) \in \mathcal{P}$, then $\eta(o') \neq s$ for every $o' \in O \setminus \{o\}$.

Proof. For $s \in S$, let $\deg(s) = |\{o \in O : \eta(o) = s\}|$ be the number of optimal centers whose closest center of S is s, and let n_0, n_1, and n_2 be the numbers of centers $s \in S$ with $\deg(s) = 0$, with $\deg(s) = 1$, and with $\deg(s) \geq 2$, respectively. Pair every s with $\deg(s) = 1$ with the unique o such that $\eta(o) = s$; this creates n_1 pairs satisfying condition 3 (Figure 4.1 shows an example). The remaining $k - n_1$ optimal centers are matched to centers of degree 0, at most two per center; condition 3 then holds trivially. This is possible because $n_0 + n_1 + n_2 = k$ and

$$k \;=\; \sum_{s \in S} \deg(s) \;\geq\; n_1 + 2n_2 \;,$$

whence $n_2 \leq (k - n_1)/2$ and thus $n_0 = k - n_1 - n_2 \geq (k - n_1)/2$. □

Theorem 4.4 (Arya et al. [16]). Every swap-optimal solution S satisfies $\mathrm{cost}(S) \leq 5 \cdot \mathrm{cost}(O)$.

Proof. Fix a pair $(s, o) \in \mathcal{P}$ from Lemma 4.3. Then $0 \leq \mathrm{cost}(S - s + o) - \mathrm{cost}(S)$: by swap-optimality if $o \notin S$, and trivially if $o \in S$, since then $S - s + o \subseteq S$ and removing centers cannot decrease the cost. We bound $\mathrm{cost}(S - s + o)$ from above by assigning every point to some center of $S - s + o$ (an upper bound, since cost uses the closest center):

- every $x \in N_O(o)$ is assigned to o, at cost $d_O(x)$;
- every $x \in N_S(s) \setminus N_O(o)$ is assigned to $\eta(o(x))$, at cost at most $2d_O(x) + d_S(x)$ by Lemma 4.2; this center is available: $o(x) \neq o$ because $x \notin N_O(o)$, hence $\eta(o(x)) \neq s$ by condition 3 of

Lemma 4.3;

- every other point x keeps its center $s(x) \neq s$, at cost $d_S(x)$.

Subtracting $\text{cost}(S) = \sum_x d_S(x)$, the first group contributes $d_O(x) - d_S(x)$ per point, the second at most $(2d_O(x) + d_S(x)) - d_S(x) = 2d_O(x)$, and the third nothing:

$$0 \;\le\; \text{cost}(S - s + o) - \text{cost}(S) \;\le\; \sum_{x \in N_O(o)} (d_O(x) - d_S(x)) \;+\; \sum_{x \in N_S(s)} 2d_O(x) \;,$$

where we extended the second sum from $N_S(s) \setminus N_O(o)$ to all of $N_S(s)$, which is safe because $2d_O(x) \geq 0$.

Now sum this inequality over all k pairs of $\mathcal{P}$. By condition 1, the clusters $N_O(o)$ appear exactly once each and partition X, so the first sums add up to $\text{cost}(O) - \text{cost}(S)$. By condition 2, each cluster $N_S(s)$ appears at most twice, and the clusters partition X, so the second sums add up to at most $2\sum_{x \in X} 2d_O(x) = 4\,\text{cost}(O)$. Altogether $0 \leq \text{cost}(O) - \text{cost}(S) + 4\,\text{cost}(O)$, that is, $\text{cost}(S) \leq 5\,\text{cost}(O)$. □

Three concluding remarks. First, the factor 5 is tight for single swaps, but swapping p centers simultaneously improves the guarantee to $3 + 2/p$ [16]; the proof follows the same pattern with a more careful pairing. Second, as for MAX-CUT, the number of improving swaps need not be polynomial, and the remedy is the same. Accept a swap only if it decreases the cost by a factor at least $1 - \delta$ with $\delta = \varepsilon/(2k)$, for some $0 < \varepsilon \leq 1$. With integer distances at most Δ, the initial cost is at most $n\Delta$ and every nonzero cost is at least 1, so Lemma 4.1 bounds the number of swaps by $O\big(\frac{k}{\varepsilon} \log(n\Delta)\big)$, polynomial in the input size and $1/\varepsilon$. At the end, each swap inequality holds up to an additive $\delta\,\text{cost}(S)$, so summing over the k pairs of $\mathcal{P}$ gives $0 \leq 5\,\text{cost}(O) - \text{cost}(S) + k\delta\,\text{cost}(S)$, that is,

$$\text{cost}(S) \;\le\; \frac{5}{1 - \varepsilon/2}\,\text{cost}(O) \;\le\; 5(1 + \varepsilon)\,\text{cost}(O) \;.$$

Third, the analysis uses only swap-optimality, not how it was reached, so the bound holds for the output of any procedure that stops at a swap-optimal solution.

September 28, 2026

Lecture 5: String Algorithms I

We have a long text and a short pattern, and we want to find every place the pattern appears in the text. The obvious method is to slide the pattern along and compare character by character. This takes $\Theta(nm)$ time, where n is the text length and m the pattern length. Any algorithm must read every character of the text and the pattern at least once, so $\Omega(n+m)$ is a lower bound. Can we match it with an $O(n+m)$ algorithm? The answer is yes, and the key insight is the same in all algorithms we will meet: when a comparison fails, the characters we have already read tell us something about where the next match could start. There is no need to throw that information away.

This lecture and the two following ones present a sequence of increasingly refined string-matching algorithms. In this lecture, we study the Morris–Pratt algorithm and its refinement by Knuth, which achieve $O(n+m)$ time by precomputing a failure function using $O(m)$ space. The next lecture treats the string-matching automaton of Simon and the Boyer–Moore algorithm; the last one presents the two-way algorithm of Crochemore and Perrin, which reduces the extra space to $O(1)$.

Given a string S, we write $S[k]$ for its k^{th} character, and $S[i..j]$ for the substring from position i to j inclusive. Given a text $T[1..n]$ and a pattern $P[1..m]$, the *string matching problem* asks to find all positions $i \in \{1, \ldots, n-m+1\}$ such that $T[i..i+m-1] = P$. Each such i is an *occurrence* of P in T.

5.1 Morris–Pratt

Suppose we have just found a complete occurrence of P in the text. We need to keep searching, but do we really have to start over from scratch? Look at the occurrence we just matched: its tail end is still sitting in the text, and if that tail happens to equal the *beginning* of P, then we already have a head start on the next match. We just slide the pattern forward so that the matching prefix–suffix lines up, and resume comparing from where we left off. We never re-read a single text character (see Figure 5.1). This is the central idea of the Morris–Pratt (MP) algorithm [17].

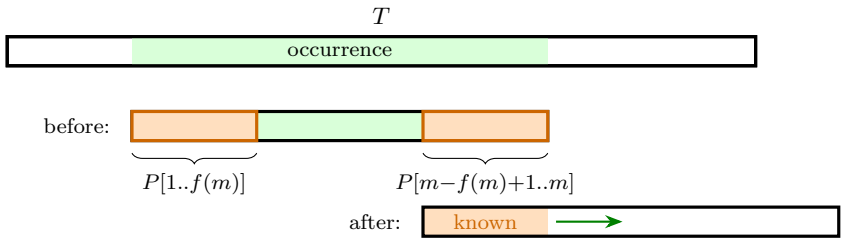

Figure 5.1: After a full match ($q = m$), MP sets $q = f(m)$. The orange suffix of the matched pattern equals the orange prefix by definition of f. These $f(m)$ characters already match the text at the shifted position, so the algorithm resumes at $P[f(m)+1]$ without re-reading.

The same idea works not only after a full match, but after any partial match. A proper prefix of a string that is also a suffix of it is called a *border* of the string. If we have matched the first q characters of P and then hit a mismatch, the longest border of $P[1..q]$ tells us exactly how far to shift. MP precomputes this for every q in the *failure function* $f\colon \{1, \ldots, m\} \to \{0, \ldots, m-1\}$:

$$f(q) = \max\{k : 0 \le k < q \text{ and } P[1..k] \text{ is a suffix of } P[1..q]\},$$

that is, $f(q)$ is the length of the longest border of $P[1..q]$.

Example. For $P =$ `ababac`:

q	1	2	3	4	5	6
$P[1..q]$	`a`	`ab`	`aba`	`abab`	`ababa`	`ababac`
$f(q)$	0	0	1	2	3	0

For instance, $f(5) = 3$ because the longest border of `ababa` is `aba`, which has length 3. Similarly, $f(4) = 2$ because `ab` is both a proper prefix and a suffix of `abab`.

How do we compute f? The trick is to match the pattern against *itself*. To see why this works, define the *failure chain* at q as the sequence $f(q), f(f(q)), \ldots, 0$ obtained by iterating f. This chain lists *every* border of $P[1..q]$, in decreasing order of length (see Figure 5.2). The reason is simple: a border of a border is itself a border, so the chain cannot skip any.

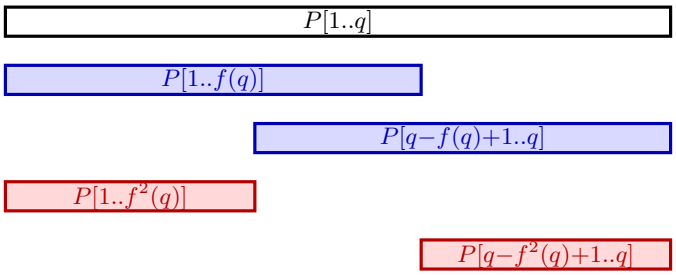

Figure 5.2: The failure chain at q. Each row shows a border of $P[1..q]$: the left-aligned bar is the prefix, the right-aligned bar is the matching suffix. A prefix of a prefix is a prefix, and a suffix of a suffix is a suffix, so every element of the chain $f(q), f^2(q), \ldots$ gives a border of $P[1..q]$.

Lemma 5.1. For $q \geq 2$:

$$f(q) = \begin{cases} k+1 & \text{if } k \text{ is the largest element of the failure chain at } q-1 \text{ such that } P[k+1] = P[q], \\ 0 & \text{if no such } k \text{ exists.} \end{cases}$$

Proof. Suppose $P[1..k{+}1]$ is a border of $P[1..q]$, with $k+1 \geq 1$. Then $P[k{+}1] = P[q]$ and $P[1..k]$ is a border of $P[1..q{-}1]$. We claim k appears in the failure chain at $q{-}1$. Indeed, $f(q{-}1) \geq k$ since $f(q{-}1)$ is the longest border; if $f(q{-}1) > k$, then $k \leq f(f(q{-}1))$ by the same reasoning applied to $P[1..f(q{-}1)]$, and by induction k is reached by iterating f. Conversely, any k in the chain with $P[k{+}1] = P[q]$ gives $P[1..k{+}1]$ as a border of $P[1..q]$. Taking the largest such k gives $f(q)$. □

In other words: to compute $f(q)$, we already know $f(q{-}1)$ from the previous iteration. We walk down the failure chain at $q{-}1$, testing whether $P[k{+}1] = P[q]$ at each step. The first k that passes gives $f(q) = k+1$; if none passes, $f(q) = 0$. This reduces the preprocessing to a single left-to-right pass over P, using the already computed values of f to guide each step. The resulting procedure takes $O(m)$ time in total, as we verify below after stating it.

Algorithm: Compute Failure Function

1. Set $f(1) = 0$ and $k = 0$.
2. For $q = 2$ to m:
   (a) While $k > 0$ and $P[k+1] \neq P[q]$, set $k = f(k)$.
   (b) If $P[k+1] = P[q]$, set $k = k+1$.
   (c) Set $f(q) = k$.

Lemma 5.2. The algorithm above correctly computes the failure function in $O(m)$ time.

Proof. We show that at the end of each outer iteration for index q, $f(q)$ is correctly set to the length of the longest border of $P[1..q]$. The invariant is that at the start of the iteration, $k = f(q-1)$. By Lemma 5.1, $f(q)$ is $k'+1$ where k' is the largest element of the failure chain at $q-1$ with $P[k'+1] = P[q]$, or 0 if none exists. The while loop walks down this chain, trying each k in decreasing order and stopping at the first match. If $P[k+1] = P[q]$, the algorithm sets $k = k+1$ and then $f(q) = k$, matching the formula. If no match is found, $k = 0$ and $f(q) = 0$. In both cases, $k = f(q)$ at the end of the iteration, maintaining the invariant for $q+1$.

For the running time: each of the $m-1$ outer iterations increments k by at most 1, so the total number of increments is at most $m-1$. Each while-loop iteration decreases k by at least 1 (since $f(k) < k$), and $k \geq 0$ always, so the total number of decrements is also at most $m-1$. Hence the while loop runs $O(m)$ times overall, giving $O(m)$ total time. □

The search phase is almost identical: we scan the text left to right, maintaining the length q of the current partial match. When a mismatch occurs, we walk down the failure chain, exactly as during preprocessing, until we find a border that can be extended, or reach 0.

Algorithm: MP Search

1. Compute the failure function f for P.
2. Set $q = 0$.
3. For $i = 1$ to n:
   (a) While $q > 0$ and $P[q+1] \neq T[i]$, set $q = f(q)$.
   (b) If $P[q+1] = T[i]$, set $q = q+1$.
   (c) If $q = m$, report an occurrence at position $i - m + 1$ and set $q = f(q)$.

Theorem 5.1. The MP algorithm finds all occurrences of P in T in $O(n+m)$ time.

Proof. We maintain the invariant that after processing $T[i]$, q equals the length of the longest prefix of P that is a suffix of $T[1..i]$. Initially $q = 0$ and the invariant holds vacuously. At step i, if $P[q+1] = T[i]$, extending the match to $q+1$ is correct. If $P[q+1] \neq T[i]$, setting $q = f(q)$ is safe: any prefix of P that is a suffix of $T[1..i-1]$ and could extend to match $T[i]$ must also be a suffix of $P[1..q]$, hence its length appears in the failure chain at q. The while loop tries each candidate in decreasing order. Thus the invariant is preserved, and occurrences (when $q = m$) are reported correctly.

Preprocessing takes $O(m)$ time by Lemma 5.2. The search phase uses the same amortized argument: q increases by at most 1 per outer iteration (at most n increments total), and each while-loop

iteration decreases q by at least 1 with $q \geq 0$ always, so at most n decrements total. The search takes $O(n)$ time, giving $O(n+m)$ overall. □

Example. Let $P = \texttt{ababac}$, with failure function $f = (0, 0, 1, 2, 3, 0)$ as computed above, and let $T = \texttt{abababac}$. The table lists, for each text position i, the comparisons made against $T[i]$ and the value of q afterwards.

i	$T[i]$	comparisons against $T[i]$	q
1	a	$P[1] = \texttt{a}$	1
2	b	$P[2] = \texttt{b}$	2
3	a	$P[3] = \texttt{a}$	3
4	b	$P[4] = \texttt{b}$	4
5	a	$P[5] = \texttt{a}$	5
6	b	$P[6] = \texttt{c} \neq \texttt{b}$, so $q \leftarrow f(5) = 3$; then $P[4] = \texttt{b}$	4
7	a	$P[5] = \texttt{a}$	5
8	c	$P[6] = \texttt{c}$; occurrence at $8 - 6 + 1 = 3$; $q \leftarrow f(6) = 0$	0

At $i = 6$, the algorithm has matched ababa and reads a b where the pattern has a c. It does not move back in the text: the border aba of ababa is still matched, and one comparison later, so is abab.

5.2 Knuth–Morris–Pratt

The bound of Theorem 5.1 is amortized: *on average* over the whole text, MP spends $O(1)$ time per text character. The *delay*, that is, the number of pattern characters compared against a single text character, can nevertheless be large. Consider $P = \texttt{a}^m$ and a text character $T[i] = \texttt{b}$ arriving after a partial match of length $q = m - 1$. The failure chain at q is $m-2, m-3, \ldots, 0$, and MP compares $P[k+1] = \texttt{a}$ against $T[i] = \texttt{b}$ for every single k on it: $m - 1$ comparisons, all of them with the same predictable outcome.

The wasted work has a clear structure. When the comparison of $P[q+1]$ against $T[i]$ fails and we fall back to the border $k = f(q)$, the next comparison pits $P[k+1]$ against the same $T[i]$. If $P[k+1] = P[q+1]$, this comparison is doomed: it must fail for the same reason the previous one did. The refinement of Knuth [18] skips these doomed candidates ahead of time. Call a border $P[1..k]$ of $P[1..q]$ *strict* if $P[k+1] \neq P[q+1]$, that is, if the character following the border differs from the character whose comparison just failed. The *strict failure function* is

$$f'(q) = \max\big(\{k : P[1..k] \text{ is a strict border of } P[1..q]\} \cup \{0\}\big) \qquad \text{for } 1 \leq q < m,$$

and $f'(m) = f(m)$, since there is no character $P[m+1]$ to compare. The KMP search algorithm is the MP search algorithm with f replaced by f'.

Example. For $P = \texttt{ababac}$, compare the two failure functions:

q	1	2	3	4	5	6
$f(q)$	0	0	1	2	3	0
$f'(q)$	0	0	0	0	3	0

Take $q = 4$: the longest border of $\texttt{abab}$ is $\texttt{ab}$, but $P[3] = \texttt{a} = P[5]$, so after a mismatch of $P[5]$ the candidate is doomed and f' skips it; the next border is the empty one. At $q = 5$ the border $\texttt{aba}$ survives because $P[4] = \texttt{b} \neq \texttt{c} = P[6]$.

Lemma 5.3. The KMP search algorithm reports exactly the occurrences of P in T, in $O(n+m)$ time.

Proof. We show that the KMP and MP searches end every outer iteration with the same value of q; the claim then follows from Theorem 5.1. Fix a text position i and let q be the common value at the start of the iteration. Let j be the largest element of the failure chain at q with $P[j+1] = T[i]$ (with $j = q$ itself allowed), if one exists. The MP while loop descends the chain and stops exactly at j. Now consider the KMP while loop at some current chain element $k > j$, so that $P[k+1] \neq T[i]$. The jump to $f'(k)$ skips precisely the chain elements j' with $P[j'+1] = P[k+1]$; all of them satisfy $P[j'+1] \neq T[i]$, so j is never skipped. Moreover, $P[1..j]$ is a strict border of $P[1..k]$ because $P[j+1] = T[i] \neq P[k+1]$, so $f'(k) \geq j$. Hence the KMP iteration descends through chain elements that all fail against $T[i]$, never passes j, and cannot stop above j by maximality of j; it therefore stops at j as well, and both algorithms continue with $q = j+1$. If no such j exists, both loops reach 0 and treat $T[i]$ identically. The running-time analysis of Theorem 5.1 applies verbatim, since $f'(q) \leq f(q) < q$. □

The point of the modification is the delay. By definition of strictness, two *consecutive* candidates tried against the same text character now always differ: if the comparison $P[q+1] \neq T[i]$ fails and we fall back to $k = f'(q) > 0$, then $P[k+1] \neq P[q+1]$, so the next comparison is not a repetition of the previous one. On $P = \texttt{a}^m$, the strict failure function is identically 0, and the delay drops from $m-1$ to 1. In general, the delay is logarithmic. The proof needs a second way of looking at borders. An integer $p \geq 1$ is a *period* of a string $S[1..k]$ if $S[i] = S[i+p]$ for all $1 \leq i \leq k-p$; every $p \geq k$ is a period vacuously, and $\mathrm{per}(S)$ denotes the smallest period. Borders and periods are two descriptions of the same thing: S has a border of length b if and only if it has period $k-b$, since both statements say that $S[i] = S[i+k-b]$ for $1 \leq i \leq b$. In particular, the failure chain at k lists the periods of $P[1..k]$ smaller than k, and $\mathrm{per}(P[1..k]) = k - f(k)$. A string may have several periods, but they cannot be independent.

Lemma 5.4 (Periodicity lemma). If p and q are periods of a string S of length at least $p+q$, then so is $\gcd(p,q)$.

Proof. By induction on $p+q$. The case $p = q$ is trivial, so let $p > q$. We show that $p-q$ is a period of S. Let $1 \leq i \leq |S| - (p-q)$. If $i+p \leq |S|$, then $S[i] = S[i+p] = S[i+p-q]$, using period p and then period q backwards. Otherwise $i > |S| - p \geq q$, and $S[i] = S[i-q] = S[i-q+p]$, using period q backwards and then period p. Now q and $p-q$ are periods of S with $q + (p-q) = p \leq |S|$ and $\gcd(q, p-q) = \gcd(p,q)$, so the induction hypothesis applies. □

The lemma holds under the weaker hypothesis $|S| \geq p+q-\gcd(p,q)$, which is tight; this is the theorem of Fine and Wilf [19]. The version above is all we need.

Lemma 5.5. Let $1 \leq a < m$, $b = f'(a)$, and $c = f'(b)$. If $c \geq 1$, then $a > b + c$.

Proof. Since $c \geq 1$, it is a strict border of $P[1..b]$: $P[1..c]$ is a border of $P[1..b]$ and $P[c+1] \neq P[b+1]$. A border of a border is a border, so $P[1..b]$ and $P[1..c]$ are both borders of $P[1..a]$, which therefore has the periods $a-b$ and $a-c$. Suppose $a \leq b+c$, that is, $a \geq (a-b)+(a-c)$. By Lemma 5.4, $g = \gcd(a-b, a-c)$ is a period of $P[1..a]$, and g divides $(a-c)-(a-b) = b-c$. Stepping from position $c+1$ to position $b+1$ in increments of g stays inside $P[1..a]$, because $b+1 \leq a$, and yields $P[c+1] = P[b+1]$, contradicting the strictness of c. □

Theorem 5.2 (Knuth, Morris, Pratt [18]). With the strict failure function, at most $1+\log_\varphi m$ comparisons are performed against any single text character, where $\varphi = (1+\sqrt{5})/2$ is the golden ratio.

Proof. Consider the comparisons against a text character $T[i]$, starting from the current value $q_0 \leq m-1$ of q; after an occurrence, q is reset to $f(m) < m$ before the next character is read. They follow the strict failure chain $q_0 > q_1 > \cdots > q_t = 0$, where $q_{s+1} = f'(q_s)$, and there are at most $t+1$ of them: one at each q_s, the last of which either succeeds or fails at $q_t = 0$. By Lemma 5.5, $q_s > q_{s+1} + q_{s+2}$ whenever $q_{s+2} \geq 1$, that is, for $s \leq t-3$. Let $F_1 = F_2 = 1$ and $F_k = F_{k-1} + F_{k-2}$ be the Fibonacci numbers. We claim that $q_{t-s} + 1 \geq F_{s+2}$ for $1 \leq s \leq t$. Indeed, $q_{t-1} + 1 \geq 2 = F_3$ and $q_{t-2} + 1 \geq 3 = F_4$ since the chain is strictly decreasing, and for $s \geq 3$,

$$q_{t-s} + 1 \;\geq\; (q_{t-s+1} + 1) + (q_{t-s+2} + 1) \;\geq\; F_{s+1} + F_s \;=\; F_{s+2} \;.$$

Hence $m \geq q_0 + 1 \geq F_{t+2} \geq \varphi^t$, using the bound $F_k \geq \varphi^{k-2}$, which holds for $k = 1, 2$ and propagates because $\varphi^2 = \varphi + 1$. Thus $t \leq \log_\varphi m$, and the number of comparisons is at most $1 + \log_\varphi m$. □

The bound is attained, up to an additive constant, on the Fibonacci strings `a`, `b`, `ab`, `aba`, `abaab`, ..., each of which is the concatenation of the two preceding ones [18].

It remains to compute f'. No new pass over the pattern is needed: f' is obtained from f in a single loop.

Algorithm: Compute Strict Failure Function

1. Compute the failure function f of P.
2. For $q = 1$ to $m-1$:
   (a) If $f(q) = 0$, set $f'(q) = 0$.
   (b) Else if $P[f(q)+1] \neq P[q+1]$, set $f'(q) = f(q)$.
   (c) Else set $f'(q) = f'(f(q))$.
3. Set $f'(m) = f(m)$.

Lemma 5.6. The algorithm above computes the strict failure function in $O(m)$ time.

Proof. For correctness, we argue by strong induction on q. The candidates for $f'(q)$ are exactly the elements of the failure chain at q, in decreasing order, and we must return the first one k with $P[k+1] \neq P[q+1]$, or 0. The first candidate is $k = f(q)$; if it is strict, we are done. Otherwise $P[f(q)+1] = P[q+1]$, so a border $P[1..k]$ of $P[1..q]$ with $k < f(q)$ is strict for q if and only if it is strict for the shorter prefix $P[1..f(q)]$: both conditions read $P[k+1] \neq P[q+1] = P[f(q)+1]$. The remaining candidates are the elements of the failure chain at $f(q)$, so the answer is $f'(f(q))$, which is already computed since $f(q) < q$. Each iteration takes constant time, hence $O(m)$ in total on top of the $O(m)$ of Lemma 5.2. □

Both MP and KMP thus run in $O(n+m)$ total time; the difference is the worst-case work per text character, which drops from $\Theta(m)$ to $O(\log m)$. This distinction matters when the text arrives as a stream and each character must be processed before the next one arrives. In the next lecture we push this direction to its natural limit: an automaton that spends *exactly one* transition per text character.

References

[1] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[2] Irit Dinur and Shmuel Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005.

[3] Uriel Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.

[4] Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 767–775, 2002.

[5] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within $2-\varepsilon$. *Journal of Computer and System Sciences*, 74(3):335–349, 2008.

[6] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[7] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, 1976.

[8] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[9] Ronald L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[10] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[11] Michel X. Goemans and David P. Williamson. New $\frac{3}{4}$-approximation algorithms for the maximum satisfiability problem. *SIAM Journal on Discrete Mathematics*, 7(4):656–666, 1994.

[12] Prabhakar Raghavan and Clark D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[13] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[14] Wen-Lian Hsu and George L. Nemhauser. Easy and hard bottleneck location problems. *Discrete Applied Mathematics*, 1(3):209–215, 1979.

[15] Dorit S. Hochbaum and David B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. *Journal of the ACM*, 33(3):533–550, 1986.

[16] Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. Local search heuristics for k-median and facility location problems. *SIAM Journal on Computing*, 33(3):544–562, 2004.

[17] James H. Morris, Jr. and Vaughan R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.

[18] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[19] Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

[20] Imre Simon. String matching algorithms and automata. In *Results and Trends in Theoretical Computer Science*, volume 812 of *Lecture Notes in Computer Science*, pages 386–395. Springer, Heidelberg, 1994.

[21] Robert S. Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[22] R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[23] Zvi Galil. On improving the worst case running time of the Boyer–Moore string matching algorithm. *Communications of the ACM*, 22(9):505–508, 1979.

[24] Richard Cole. Tight bounds on the complexity of the Boyer–Moore string matching algorithm. *SIAM Journal on Computing*, 23(5):1075–1091, 1994.

[25] Alberto Apostolico and Raffaele Giancarlo. The Boyer–Moore–Galil string searching strategies revisited. *SIAM Journal on Computing*, 15(1):98–105, 1986.

[26] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[27] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[28] Maxime Crochemore and Dominique Perrin. Two-way string-matching. *Journal of the ACM*, 38(3):651–675, 1991.

[29] M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.

[30] Anatoliy I. Serdyukov. On some extremal walks in graphs. *Upravlyaemye Sistemy*, 17:76–79, 1978.

[31] Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. A (slightly) improved approximation algorithm for metric TSP. In *Proceedings of the 53rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 32–45, 2021.

[32] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM*, 34(1):144–162, 1987.

[33] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[34] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.