Advanced Algorithmics

Thomas Nowak

March 8, 2026

These are partial lecture notes from the undergraduate course Advanced Algorithmics, given by Serge Haddad and Thomas Nowak at ENS Paris-Saclay in the academic year 2025–2026. Good references for the material covered here include Williamson and Shmoys [1] for approximation algorithms, Motwani and Raghavan [2] for randomized algorithms, and the lecture notes by Ghaffari [3] and the textbook by Suomela [4] for distributed algorithms.

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,
Spring Term: 2025–2026,
Last Update: March 8, 2026,
ENS Paris-Saclay

Contents

February 19, 2026

Lecture 6: 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. The key insight is that for many practical problems, we don't actually need the perfect solution: we just need a solution that's provably close to optimal and can be computed quickly.

6.1 Foundations and 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 6.1

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, which is not perfect but often good enough in practice.

The landscape of approximable problems is characterized by several complexity classes. The class APX contains problems that admit constant-factor 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 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. Understanding which problems fall into which category is a central question in approximation algorithms.

6.2 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 6.1. 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:

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

Taking logarithms:

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

Using the inequality $\log(1 - x) \leq -x$ for $0 < x < 1$:

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

Therefore:

$$t < \mathrm{OPT} \cdot \log|E_0| \leq 2 \cdot \mathrm{OPT} \cdot \log n$$

Since the algorithm selects t vertices, we have $|C| = t = O(\mathrm{OPT} \cdot \log n)$. □

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

Theorem 6.2. 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 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 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:

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

□

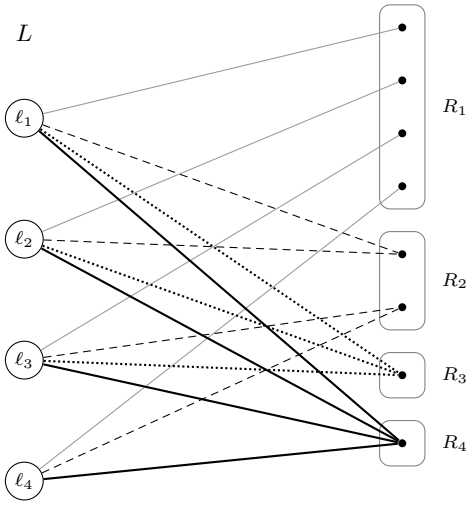

Figure 1: The graph G_4 with $r = 4$. Each vertex in R_i has degree exactly i. Edge styles distinguish groups: R_1 thin gray, R_2 dashed, R_3 dotted, R_4 thick solid.

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 $|C| \leq c \cdot \text{OPT}$. 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 $\text{OPT} \geq |C|/c$.

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 $\text{OPT} \geq 1$; two edges certify $\text{OPT} \geq 2$—but only if they don't share an endpoint, since otherwise a single vertex could satisfy both demands simultaneously. More generally, to certify $\text{OPT} \geq k$, 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 $C = \emptyset$ and $E' = E$.
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 6.3. 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 $\text{OPT} \geq |M|$. Therefore:

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

□

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. This turns out to be a deep question connected to fundamental conjectures in complexity theory.

On the hardness side, Dinur and Safra [5] 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 [7]. 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 [8] 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.

6.3 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 6.1. 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 $\mathcal{C} = \emptyset$ and $R = U$.
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.

Theorem 6.4. The greedy algorithm for set cover is an H_n-approximation, where $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the n-th harmonic number. Since $H_n \leq \ln n + 1$, the algorithm achieves 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 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 10.1, we have $H_n \leq 1 + \ln n$, 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 [6] proved that set cover cannot be approximated within a factor of $(1-\varepsilon)\ln n$ 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.

This is a rare and beautiful case where we fully understand the approximability of a problem: the greedy algorithm achieves $\ln n$ approximation, and this is the best possible unless unusual complexity-theoretic collapses occur.

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 6.4, 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 $K \ll n$, but still logarithmic.

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 $|\mathcal{C}^*| = \mathrm{OPT}$. For each element $e \in U$, assign it a weight of $1/f$. Since e belongs to at most f sets, 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, since $\mathcal{C}^*$ covers all elements, summing the weights over sets in $\mathcal{C}^*$ gives at least n/f, so $\sum_{S \in \mathcal{C}^*} |S|/f \geq n/f$, which means $\sum_{S \in \mathcal{C}^*} |S| \geq n$. Now consider the algorithm that simply selects every set containing at least one uncovered element, greedily avoiding redundant sets. More directly: 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* subcollection that covers U uses at most $f \cdot \mathrm{OPT}$ 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 6.1) 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 6.3 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 6.4 gives an alternative $O(\log n)$ proof of Theorem 6.1.

The difference in approximability between vertex cover and general set cover thus comes down to structure. For vertex cover, we have a 2-approximation and suspect (via UGC) that this is optimal, but without UGC we only know a hardness factor of 1.3606. For set cover, we have matching upper and lower bounds of $\Theta(\log n)$, giving a complete picture of the problem's approximability.

6.4 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 $\mathrm{DP}[i][s]$ which is true if we can achieve sum exactly s using a subset of the first i items. The recurrence is:

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

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

Theorem 6.5. For any $\varepsilon > 0$, the above algorithm is a $(1+\varepsilon)$-approximation for subset sum, running in time $O(n^3/\varepsilon)$.

Proof. We first bound the size of each list. We claim that $|L_i| \leq \frac{2n}{\varepsilon} \log T$ 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:

$$(1+\delta)^{k-1} \leq nT$$

Taking logarithms:

$$(k-1)\log(1+\delta) \leq \log(nT)$$

Using $\log(1+\delta) \geq \delta/2$ for small δ:

$$k \leq 1 + \frac{\log(nT)}{\delta/2} = 1 + \frac{2\log(nT)}{\delta} = 1 + \frac{2\log(nT)}{\varepsilon/(2n)} = 1 + \frac{4n\log(nT)}{\varepsilon}$$

Therefore $|L_i| = O(n \log T/\varepsilon)$.

We now 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:

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

Base case: $L_0 = \{0\}$ and $\mathrm{OPT}_0 = 0$, so the claim holds.

Inductive step: 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 $\varepsilon/2 \leq \ln(1+\varepsilon)$ (since $\ln(1+x) \geq x - x^2/2 \geq x/2$ for $x \in [0,1]$), so $e^{\varepsilon/2} \leq 1+\varepsilon$. Therefore, 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.

For the running time, each iteration i processes L_{i-1}, which has size $O(n \log T/\varepsilon)$. Computing L'_i takes $O(|L_{i-1}|)$ time and trimming takes $O(|L'_i|)$ time. Sorting takes $O(|L'_i| \log |L'_i|) = O((n \log T/\varepsilon) \log n)$ time.

Over n iterations, the total time is:

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

If we assume $T \leq 2^{poly(n)}$ (polynomial number of bits), then $\log T = O(n^c)$ for some constant c, giving total time $O(n^3/\varepsilon)$. □

This example illustrates that some NP-hard problems admit FPTAS: for any $\varepsilon > 0$, we achieve approximation ratio $1+\varepsilon$ in time polynomial in both n and $1/\varepsilon$. The key technique is controlled loss of precision: by trimming values that are very close to each other, we reduce the state space while losing only a small multiplicative factor in solution quality.

March 9, 2026

Lecture 7: 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 0/1 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.

7.1 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 [14] showed that no finite approximation ratio is achievable.

Theorem 7.1. Unless P = NP, for any polynomial-time computable function $\rho(n)$, there is no polynomial-time $\rho(n)$-approximation algorithm 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). For approximation, it 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 7.1. 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 7.2. 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 7.1. 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 [13] and independently Serdyukov [21], 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 [24] 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 7.2. The minimum-weight perfect matching M on S satisfies $c(M) \leq \mathrm{OPT}/2$.

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 $c(M_1) + c(M_2) \leq \mathrm{OPT}$. Therefore, $\min(c(M_1), c(M_2)) \leq \mathrm{OPT}/2$. Since M is a minimum-weight perfect matching, $c(M) \leq \mathrm{OPT}/2$. □

Theorem 7.3. 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 7.1, $c(T) \leq \mathrm{OPT}$. By Lemma 7.2, $c(M) \leq \mathrm{OPT}/2$. Therefore:

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

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 [22] achieved a breakthrough by obtaining a $(\frac{3}{2} - \varepsilon_0)$-approximation for some small constant $\varepsilon_0 > 0$, the first improvement over $\frac{3}{2}$. On

the hardness side, metric TSP is known to be APX-hard: there is no PTAS unless P = NP.

7.2 Knapsack

The 0/1 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 $S \subseteq \{1, \dots, n\}$ maximizing $\sum_{i \in S} p_i$ subject to $\sum_{i \in S} w_i \leq W$. We may assume without loss of generality that $w_i \leq W$ 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 remarkably 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 the table by weight, giving running time $O(nW)$. For our purposes, we index by *profit* instead. Define $A[i][p]$ as the minimum total weight needed to achieve profit exactly p using a subset of items $\{1, \dots, i\}$. 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 $A[n][p] \leq W$. 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 $nK \leq \varepsilon \cdot \mathrm{OPT}$. We don't know OPT, but we do know that $\mathrm{OPT} \geq p_{\max}$ (since every item fits individually). Setting $K = \varepsilon \cdot p_{\max}/n$ guarantees $nK = \varepsilon \cdot p_{\max} \leq \varepsilon \cdot \mathrm{OPT}$.

Algorithm (FPTAS for Knapsack):

1. Let $p_{\max} = \max_i p_i$.
2. Set the scaling factor $K = \frac{\varepsilon \cdot p_{\max}}{n}$.
3. For each item i, define the scaled profit $p'_i = \left\lfloor \frac{p_i}{K} \right\rfloor$.
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 7.4. 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:

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

The total scaled profit is $P' = \sum_{i=1}^{n} p'_i \leq n^2/\varepsilon$. 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 $\sum_{i \in S} p'_i \geq \sum_{i \in S^*} p'_i$.

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

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

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

The profit-scaling technique, introduced by Ibarra and Kim [15], 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.

7.3 Load Balancing

We now turn to a scheduling problem, which is NP-hard (by reduction from the Partition 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:

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

As before, we start by asking: what are good lower bounds on the optimal makespan? Two bounds are immediate.

Lemma 7.3. The optimal makespan satisfies:

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

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 $\max_i t_i$. □

Neither bound alone is tight in general, but together they capture the two fundamentally different sources of difficulty: too much total work, or a single job that is too large.

The simplest scheduling heuristic is list scheduling, introduced by Graham [16]: 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 7.5. 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 $\sum_{i<\ell} t_i \leq \sum_i t_i - t_\ell$, and since M_j has the smallest load, $m \cdot L \leq \sum_i t_i - t_\ell$. Therefore:

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

by Lemma 7.3. Also, $t_\ell \leq \max_i t_i \leq \mathrm{OPT}$ by the same lemma. Therefore:

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

□

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 [17] observed that this simple modification leads to a significantly improved guarantee.

Algorithm (Longest Processing Time first, LPT):

1. Sort jobs so that $t_1 \geq t_2 \geq \cdots \geq t_n$.

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 7.6. 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. Since the jobs are sorted in decreasing order and $n > m$, we have $\ell \ge m+1$, so $t_\ell \le t_{m+1}$.

We distinguish two cases.

Case 1: $t_\ell \le \mathrm{OPT}/3$. Using the same bound on L as in Theorem 7.5:

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

Case 2: $t_\ell > \mathrm{OPT}/3$. Since jobs are sorted in decreasing order and $\ell \ge m+1$, we have $t_i \ge t_{m+1} \ge t_\ell > \mathrm{OPT}/3$ for all $i \in \{1, \dots, m+1\}$. In any schedule, no machine can hold three or more of these $m+1$ large jobs (their total processing time would exceed OPT). By pigeonhole ($m+1$ large jobs, m machines, at most 2 per machine), exactly one machine holds two large jobs and the remaining $m-1$ machines each hold one.

We claim that LPT is optimal. Consider any optimal schedule. The makespan is at least the load of the machine holding two large jobs, say $t_a + t_b$ with $a < b$ (so $t_a \ge t_b$). Now suppose in some other schedule, t_1 is paired with t_k where $k \le m$. Then $t_k \ge t_{m+1}$, so $t_1 + t_k \ge t_1 + t_{m+1}$: pairing t_1 with any job other than t_{m+1} does not decrease the load of the heavy machine. The optimal pairing is therefore to place $t_1, \dots, t_m$ on separate machines and pair t_{m+1} with the lightest, which is t_m. This is exactly what LPT does: it assigns the first m jobs round-robin, then t_{m+1} joins the least loaded machine (holding t_m). The remaining small jobs $t_{m+2}, \dots, t_n$ are each at most $t_{m+1} \le \mathrm{OPT}$, and LPT assigns each to the current least loaded machine, which cannot exceed the optimal makespan. Therefore $\mathrm{ALG} = \mathrm{OPT}$.

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

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 [23] 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.

March 30, 2026

Lecture 10: Algorithms and Probability I

We start our discussion of randomization in algorithms by going through a number of representative examples that can be studied either by using only elementary probability-theoretic facts or by looking solely at the expected value of certain quantities. The most important technical fact about the expected value that we need is that it is linear: $\mathbb{E}(X+Y) = \mathbb{E}X + \mathbb{E}Y$. This equality holds even if X and Y are not independent.

10.1 Expected Runtime of Quicksort

We recall the quicksort algorithm to sort the list S of distinct elements from a totally ordered universe:

1. If S has at most one element, return S.
2. Choose a pivot element $s \in S$.
3. Compare each element of S to s and construct lists S_1 and S_2 with the elements that are less than, respectively greater than, s.
4. Recursively sort S_1 and S_2.
5. Return S_1, x, S_2.

The deterministic version that chooses the first element of the list as the pivot element has the worst-case running time $\Omega(n^2)$ for lists of length n. If we make the choice in Step 2 an independent uniformly random one, this yields a randomized algorithm. To upper-bound the runtime of randomized quicksort, we first note that it is asymptotically dominated by the number of comparisons. We can then bound the expected number of comparisons:

Theorem 10.1. The expected number of comparisons of randomized quicksort on a list of length n is $2n \log n + O(n)$.

Proof. Let $x_1, \dots, x_n$ be the input values and $y_1, \dots, y_n$ the same values in increasing order. Let X be the number of comparisons, and let X_{ij} be the indicator variable for whether y_i and y_j are ever compared (for $i < j$).

Set $Y_{ij} = \{y_i, \dots, y_j\}$. We have $X_{ij} = 1$ if and only if y_i or y_j is the first pivot element selected from Y_{ij}. We thus have:

$$\begin{aligned}\mathbb{E}X &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}X_{ij} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{k=2}^{n} \sum_{i=1}^{n+1-k} \frac{2}{k} \\ &= \sum_{k=2}^{n} (n+1-k)\frac{2}{k} = (2n+2)\sum_{k=1}^{n} \frac{1}{k} - 4n = 2n \log n + O(n)\end{aligned}$$

Here, we used Lemma 10.1 in the last step. □

The following lemma is useful in many situations. 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 10.1. 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. □

10.2 The Secretary Problem

In the secretary problem, we seek to hire a new secretary. We must decide in an online fashion: we interview candidates sequentially, and once we pass on a candidate, we cannot return to that decision. We suppose each of the n candidates has a score, which we can perfectly assess during the interview. We do not assume any *a priori* knowledge of the distribution of the scores. Our goal is to pick the candidate with the highest score. We assume all candidate scores $x_1, x_2, \ldots, x_n \in \mathbb{R}$ are distinct, and that the order of the candidates is uniformly random, *i.e.*, all permutations are equally likely.

At the i^{th} step, the algorithm has seen the scores $x_1, \ldots, x_i$ of the first i candidates. It turns out that the following class of algorithms is optimal: Reject the first m candidates, but keep track of their maximum score. Then, for each of the candidates $m+1, \ldots, n$, hire the first candidate whose score exceeds all previously seen scores.

For this class of algorithms, we now want to determine the best parameter m, *i.e.*, the one that yields the highest probability of hiring the best candidate. Denote by E_i the event that the i^{th} candidate has the highest score and that we hire them. For $i \leq m$, we have $\mathbb{P}(E_i) = 0$, since we do not hire any of the first m candidates. For $i = m+1$, we have $\mathbb{P}(E_{m+1}) = 1/n$, since we hire the highest-scoring candidate in position $m+1$ only if they happen to be there. More generally, for positions $i > m$, we have

$$\mathbb{P}(E_i) = \mathbb{P}(H_i \mid B_i) \cdot \mathbb{P}(B_i)$$

by Bayes' theorem, where H_i is the event that we hire the i^{th} candidate and B_i is the event that the best candidate is in position i. Due to the uniformity assumption, we have $\mathbb{P}(B_i) = 1/n$. Now, to calculate $\mathbb{P}(H_i \mid B_i)$, we note that we hire candidate i if and only if we did not hire anyone before them. This happens precisely when the highest score among the first $i-1$ candidates occurred within the first block of m. We thus have $\mathbb{P}(H_i \mid B_i) = m/(i-1)$. In summary:

$$\mathbb{P}(E_i) = \begin{cases} 0 & \text{if } i \leq m \\ \frac{m}{i-1} \cdot \frac{1}{n} & \text{otherwise} \end{cases}$$

Hence, letting E denote the event that we hire the best candidate, we obtain:

$$\mathbb{P}(E) = \frac{m}{n} \sum_{i=m+1}^{n} \frac{1}{i-1} = \frac{m}{n} (H_{n-1} - H_{m-1})$$

As in the proof of Lemma 10.1, we can show that $H_{n-1} - H_{m-1} \geq \log n - \log m$, so

$$\mathbb{P}(E) \geq \frac{m}{n} (\log n - \log m) \quad . \tag{1}$$

Differentiating this lower bound gives

$$\frac{d}{dm} \frac{m}{n} (\log n - \log m) = \frac{1}{n} (\log n - \log m) - \frac{1}{n} = \frac{\log n - \log m - 1}{n} \quad ,$$

which is zero if and only if $\log m = \log n - 1$, *i.e.*, $m = n/e$. Plugging this into (1) gives $\mathbb{P}(E) \geq 1/e \approx 37\%$.

We did make some approximations here: For one, we optimized a lower bound on $\mathbb{P}(E)$ rather than the probability itself. Also, we cannot choose $m = n/e$ exactly since this is not an integer, so we must round, *e.g.*, to $m = \lfloor n/e \rfloor$. In both cases, we introduce small errors, but asymptotically we remain optimal: The lower bound is tight since we also have the upper bound $\mathbb{P}(E) \leq \frac{m}{n}(\log(n-1) - \log(m-1)) \sim 1/e$ as $n \to \infty$. As for rounding, note:

$$\frac{\lfloor n/e \rfloor}{n} \log \frac{n}{\lfloor n/e \rfloor} \geq \frac{(n-e)/e}{n} \log \frac{n}{n/e} = \left(1 - \frac{e}{n}\right) \frac{1}{e} \sim \frac{1}{e}$$

as $n \to \infty$.

We also left open why this class of algorithms is optimal. Note that whenever an optimal algorithm chooses to hire candidate i, that candidate must be the best seen so far. Otherwise, we forgo a chance to hire the best candidate. Moreover, since we lack information about future scores, we always have $\mathbb{P}(B_i \mid H_i) = i/n$ for optimal algorithms. Since this increases with i, once we begin considering hiring, we must continue to do so. This implies that the choice of m depends only on n.

Thus, we conclude:

Theorem 10.2. An optimal algorithm for the secretary problem has an asymptotic success probability of $1/e$.

10.3 The Online Paging Problem

In the paging problem, we are tasked with managing a cache of size k. We receive a request sequence $\rho_1, \rho_2, \ldots, \rho_n$ of pages in a much larger (virtually infinite) main memory. At each request, if the requested page is not currently in the cache, we can choose which of the pages to evict to make space for the newly requested one. The cache is initially populated with some set of pages. Popular deterministic online algorithms for this problem are LRU (last recently used), FIFO(first-in first-out), and LFU (least frequently used). In contrast to online algorithms, offline algorithms can look at the complete request sequence—in the past as well as into the future—for each decision.

We assume that each cache miss incurs a constant cost, which we normalize to being 1. Formally, for any paging algorithm A and each request sequence $\rho_1, \ldots, \rho_n$,

we define $f_A(\rho_1, \dots, \rho_n)$ to be the number of cache misses when executing A with the request sequence $\rho_1, \dots, \rho_n$. For a randomized algorithm A, this is a random variable.

Since we can always choose a request sequence with $f_A(\rho_1, \dots, \rho_n) = n$ by requesting n different pages, it does not make much sense to look at f_A in the absolute. Rather, we will compare to the theoretical optimum:

Definition 10.1

Let O be an optimal offline algorithm and let $c \in \mathbb{R}_+$. A (randomized) algorithm A is c-competitive if there is some $b \in \mathbb{R}_+$ such that

$$\mathbb{E} f_A(\rho_1, \dots, \rho_n) \leq c f_O(\rho_1, \dots, \rho_n) + b$$

for all n and all request sequences $\rho_1, \dots, \rho_n$.

It turns out that randomization enables an exponential improvement in the competitivity ratio c. To show this, we first start with the lower bound for for deterministic algorithms:

Theorem 10.3. No deterministic online algorithm is c-competitive for $c < k$.

Proof. We construct an infinite request sequence of $k+1$ pages in the main memory, *i.e.*, $\rho_i \in \{1, \dots, k+1\}$ for all $i \in \mathbb{N}$. We do this in an inductive fashion: Initially, we populate the cache with pages $1, \dots, k$. Then, we request the unique page ρ_{i+1} that is not in the cache when executing A on the request sequence $\rho_1, \dots, \rho_i$.

For analysis purposes, we introduce the notion of a round in the request sequence. A round is a maximal subsequence in which at most k different pages are requested. The first round is $\rho_1, \dots, \rho_k$.

An optimal offline algorithm misses at most once in a round: Since at most k different pages are requested, there is one that is not requested. The optimal algorithm can thus choose to evict the non-requested page at the first cache miss. The online algorithm A, on the other hand, misses at least k times (since it misses at every request).

To conclude, we distinguish two cases. If there are infinitely many rounds, then

$$\frac{f_A(\rho_1, \dots, \rho_{n_r})}{f_O(\rho_1, \dots, \rho_{n_r})} \geq \frac{rk}{r} = k$$

where n_r is the last index of round number r. This is a contradiction to A being c-competitive with $c < k$. In the second case, there is only a finite number $R \in \mathbb{N}$ of rounds. Since $f_A(\rho_1, \dots, \rho_n) = n$ by construction and $f_O(\rho_1, \dots, \rho_n) \leq R$ for all $n \in \mathbb{N}$, we have

$$\lim_{n \to \infty} \frac{f_A(\rho_1, \dots, \rho_n)}{f_O(\rho_1, \dots, \rho_n)} \geq \lim_{n \to \infty} \frac{n}{R} = \infty \ ,$$

which is in contradiction to A being c-competitive for any c. □

When considering competitivity of randomized algorithms, multiple different definitions are possible. They differ in the amount of information that can be accessed to construct the request sequence. The procedure that constructs this sequence if often

referred to as an *adversary*. For now, we restrict ourselves to *oblivious* adversaries, who have access to the source code of the algorithm, but not to any of the random choices made during execution. Adversaries that can do that are usually called adaptive.

In the Marker algorithm, each cache location has a marker bit associated with it. It proceeds in rounds. At the start of each round, all marker bits are set to zero. When a request results in a cache hit, the marker bit of that location is set to one. When a request results in a cache miss, a random unmarked location is evicted, the requested page is placed in that location, and the location's marker bit is set to one. The current round ends when all location's marker bits are set to one.

Theorem 10.4. There exists a randomized online algorithm that is $2(1 + H_k)$-competitive against oblivious adversaries.

Proof. We use the Marker algorithm. Let $\rho_1, \dots, \rho_n$ be a request sequence.

Let R be the number of rounds of the Marker algorithm with this request sequence. Call a page *fresh* in round r if it was newly added to the cache by the Marker algorithm in round r. Let f_r be the number of fresh cache locations in round r.

If $r \geq 2$, during rounds $r-1$ and r, there are at least $k + f_r$ different page requests. Of these, at least f_r are cache misses in any algorithm, in particular the optimal one. Setting $\Delta_O(0) = 0$ and $\Delta_O(r) = f_O(\rho_1, \dots, \rho_{n_r}) - f_O(\rho_1, \dots, \rho_{n_{r-1}})$ where n_r denotes the last index of round $r \geq 1$, we have $\Delta_O(r-1) + \Delta_O(r) \geq f_r$ for all $r \geq 1$. We showed this for $r \geq 2$, but it is also true for $r = 1$ since both O and A start with the same pages in the cache. Hence:

$$\begin{aligned} 2f_O(\rho_1, \dots, \rho_n) &\geq 2\sum_{r=1}^{R} \Delta_O(r) \geq \sum_{r=0}^{R-1} \Delta_O(r) + \sum_{r=1}^{R} \Delta_O(r) \\ &= \sum_{r=1}^{R} (\Delta_O(r-1) + \Delta_O(r)) \geq \sum_{r=1}^{R} f_r \end{aligned} \tag{2}$$

We now turn to upper-bounding the expected number of cache misses of algorithm A. Let us focus on a given round r and write $\Delta_A(r) = f_A(\rho_1, \dots, \rho_{n_r}) - f_A(\rho_1, \dots, \rho_{n_{r-1}})$ for the number of cache misses in that round. There are exactly f_r cache misses due to requests for fresh pages. Let $i_1, \dots, i_{k-f_r}$ be the requested non-fresh pages, in the order of the first request to them. Let X_ℓ be the indicator variable that is 1 if and only if the first access to i_ℓ is a cache miss. The expected number of cache misses is:

$$\mathbb{E}\Delta_A(r) = f_r + \sum_{\ell=1}^{k-f_r} \mathbb{E}X_\ell = f_r + \sum_{\ell=1}^{k-f_r} \mathbb{P}(X_\ell = 1) \tag{3}$$

To bound $\mathbb{P}(X_\ell = 1)$, denote by γ the number of fresh pages in the cache before the first access to i_ℓ. The page i_ℓ might have been replaced by a previously accessed page. Compared to the start of the round, exactly γ pages are different. We know that $\ell - 1$ of the pages have to be the same, namely $i_1, \dots, i_{\ell-1}$, but

we don't know the fate of i_ℓ. We thus have:

$$\mathbb{P}(X_\ell = 1) = \frac{\gamma}{k-\ell+1} \leq \frac{f_r}{k-\ell+1} \tag{4}$$

Combining (3) and (4), the expected number of cache misses in round r is at most

$$\mathbb{E}\Delta_A(r) = f_r \left(1 + \sum_{\ell=1}^{k-f_r} \frac{1}{k-\ell+1}\right) \leq f_r(1+H_k) \ . \tag{5}$$

Now, combining (2) and (5) gives

$$\mathbb{E}f_A(\rho_1,\ldots,\rho_n) \leq (1+H_k)\sum_{r=1}^{R} f_r \leq 2(1+H_k)f_O(\rho_1,\ldots,\rho_n)$$

and concludes the proof. □

The Marker algorithm is almost optimal. It can be proved that no randomized online algorithm is c-competitive against oblivious adversaries if $c < H_k$.

April 9, 2026

Lecture 11: Algorithms and Probability II

We now turn to more involved analysis techniques than elementary probability and expected values. In terms of systems under consideration, we focus on randomized data structures.

11.1 Skip Lists

A skip list is a collection of ordered doubly linked lists $L_1, L_2, \ldots, L_r$ such that $\text{Vals}(L_r) = \emptyset$ and $\text{Vals}(L_i) \subseteq \text{Vals}(L_{i-1})$ for all $1 < i \leq r$ where $\text{Vals}(L)$ denotes the set of values in the linked list L. In addition to the horizontal links inside of its linked list L_i, each node also has a vertical link to node in L_{i-1} that contains the same value. Each linked list L_i additionally contains a start node with value $-\infty$ and an end node with value $+\infty$. The skip list encodes the set $\text{Vals}(L_1)$ of values. The linked list L_i is called the i^{th} level of the skip list.

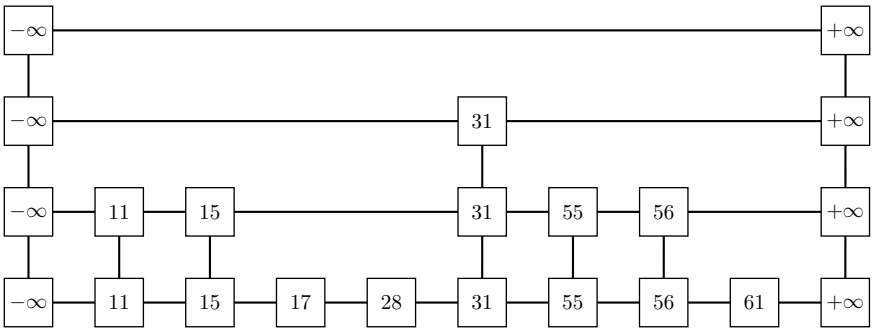

Figure 2: A skip list with $r = 4$ levels containing the values $\text{Vals}(L_1) = \{11, 15, 17, 28, 31, 55, 56, 61\}$.

For a value x in the skip list, we denote by $h(x) = \max\{i \mid x \in \text{Vals}(L_i)\}$ its height, *i.e.*, highest level that contains x. Since $\text{Vals}(L_r) = \emptyset$, we have $h(x) \leq r - 1$ for all values x of the skip list. On the other hand, we have the expression $r = 1 + \max\{h(x) \mid x \in \text{Vals}(L_1)\}$ for the number of levels.

An empty skip list has a single level. Starting from an initially empty skip list, this data structure supports three operations:

- `search`(x): returns true if x is a value in the skip list and false otherwise. The search starts at the $-\infty$ node of L_r. In each iteration, we look at the next node in the current level. If its value is equal to x, we return true. If its value is strictly smaller than x, then we advance to the next node in current level. If its value is strictly larger than x, then we drop down one level if possible. If it's not possible to drop down a level, then we return false.
- `insert`(x): inserts value x into the skip list if it's not already in the list. We first use `search`(x) to find the greatest lower bound and the least upper bound on x in each level. If x is already in the list, we do nothing more and return. Otherwise, we choose a random height $h(x)$ and insert x into the lists $L_1, \ldots, L_{h(x)}$ at the

correct location and add the vertical links. If necessary, *i.e.*, if $h(x) \geq r$, we create new empty levels before inserting x into them.

- `delete`(x): deletes value x if it's in the list. We first use `search`(x) to find look for x in each level. If x is not in any level, we do nothing more and return. Otherwise, we delete x from each level in which we found it. We then delete all empty levels except the top level.

In the description of the operations, we left one thing unspecified: how to choose the random height $h(x)$ in the insertion operation? The most often employed method is to have $h(x)$ be a geometric random variable with parameter $p = 1/2$. That is, the number of fair coin flips until we get one tails. During the analysis of the runtime of the operations, we will see why this is a good choice.

As a warm-up to analyzing the asymptotic performance of the operations, we show that the number of levels of a skip list of n values is $O(\log n)$ with high probability.

The notion "with high probability" is very common, but doesn't always have the same meaning. In its most basic meaning it says that the error probability decreases to zero with $1/n^\alpha$ for some $\alpha > 0$ as $n \to \infty$. Of course, this presupposes the parameter n to measure the size of the system under consideration in some meaningful way. The most useful version of "$X_n = O(f(n))$ with high probability" is "for all $\alpha > 0$ there exist $c > 0$ and $d > 0$ such that $\mathbb{P}\big(X_n > cf(n)\big) \leq d/n^\alpha$ for all (large enough) n". Definitions of "with high probability" that allow a free choice of the exponent α have the advantage of being composable much more easily.

Lemma 11.1. Let $\alpha > 1$. In a skip list of n values, we have $\mathbb{P}(r > \alpha \log_2 n) \leq 4/n^{\alpha-1}$ for the number r of levels.

Proof. The random variables $h(x)$ for $x \in \mathrm{Vals}(L_1)$ are i.i.d. geometric with parameter $p = 1/2$. This is true no matter the order of the preceding operations: the height chosen during the insertion of each of the currently present values is independent of all previous and later operations.

Let us write $\{x_1, x_2, \ldots, x_n\}$ for the set of values in the skip list. For the number of levels, we already established the formula $r = 1 + \max\{h(x_j) \mid 1 \leq j \leq n\}$. We thus have

$$\begin{aligned} \mathbb{P}(r > \alpha \log n) &= \mathbb{P}\big(1 + \max\{h(x_j) \mid 1 \leq j \leq n\} > \alpha \log_2 n\big) \\ &= \mathbb{P}\big(\exists 1 \leq j \leq n\colon\ h(x_j) > \alpha \log_2 n - 1\big) \\ &\leq n \cdot \mathbb{P}\big(h(x_1) > \alpha \log_2 n - 1\big) \end{aligned} \tag{6}$$

where we used the union bound $\mathbb{P}(E_1 \cup \cdots \cup E_n) \leq \mathbb{P}(E_1) + \cdots + \mathbb{P}(E_n)$.

Now, using the specific distribution of the height $h(x_1)$, we have $\mathbb{P}\big(h(x_1) > t\big) = (1-p)^t = 1/2^t$ for every nonnegative integer t. From this, we can deduce an bound that is true for all real numbers $t \geq 1$:

$$\mathbb{P}\big(h(x_1) > t - 1\big) = \mathbb{P}\big(h(x_1) > \lfloor t \rfloor - 1\big) = 1/2^{\lfloor t \rfloor - 1} \leq 1/2^{t-2}$$

Here, we used that $n > x$ if and only if $n > \lfloor x \rfloor$ for all integers n and real numbers x, and the fact that $\lfloor t \rfloor - 1 \geq t - 2$. We also see that the bound trivially remains true for the remaining values of t, *i.e.*, for $t < 1$ since then $1/2^{t-2} > 1/2^{-1} = 2$, which is bigger than any probability.

Plugging in $t = \alpha \log_2 n$, we get:

$$\mathbb{P}\big(h(x_1) > \alpha \log_2 n - 1\big) \leq 1/2^{\alpha \log_2 n - 2} = 4/n^\alpha$$

Combining this with (6) now concludes the proof. □

For the complete analysis of the time complexity of the three operations of a skip list, we will need the arguably most important inequality for the analysis of randomized algorithms, the Chernoff bound. It is an upper bound on the probability of a sum of Bernoulli random variables being far from its expected value.

Theorem 11.1 (Chernoff bound)**.** Let $X_1, X_2, \ldots, X_m$ be mutually independent 0/1 random variables and let $X = \sum_{k=1}^m X_k$. Then

$$\mathbb{P}\big(X \leq (1-\delta)\mu\big) \leq e^{-\mu\delta^2/2}$$

for all $0 < \delta < 1$ where $\mu = \mathbb{E}\, X$.

We can now prove that the search operation takes only logarithmic time with high probability. Since the time complexity of the insertion and deletion operations are dominated by the search, we get the same result for all three operations of the skip list.

Theorem 11.2. The time complexity of `search`(x) in a skip list of n values is $O(\log n)$ with high probability.

Proof. We bound the number of iterations of the search operation, which asymptotically dominates the time complexity. Independently of whether the searched value x was found, we consider the node visited in the last iteration and work our way backwards to the initial $-\infty$ node of level L_r.

The claimed bound of $m = O(\log n)$ on the number m if iterations comes from the symmetry of following vertical or horizontal links due to the choice of the fair parameter $p = 1/2$ for the heights, and the already established height bound from Lemma 11.1. Once we reach the highest level L_r, we are back at the node of the first iteration.

We are thus done if we find a constant d such that a sequence of at least $m = d \log n$ fair coin flips (iterations) has at least $\alpha \log_2 n = c \log n$ heads (vertical moves). By Theorem 11.1, the probability of this event is upper-bounded by

$$\mathbb{P}\left(X \leq (1-\delta)\frac{m}{2}\right) \leq e^{-m\delta^2/4}$$

where $(1-\delta)\frac{m}{2} = c \log n$. Solving for δ, we get $\delta = 1 - 2c/d$. Choosing $d = 4c$, we get $\delta = 1/2$ and thus:

$$\mathbb{P}\big(\leq c \log n \text{ vertical moves}\big) \leq \mathbb{P}\left(X \leq c \log n\right) \leq e^{-d \log n} = e^{-4c \log n}$$

Now, putting things together, we have:

$$\begin{aligned}\mathbb{P}\left(\geq d\log n \text{ iterations}\right) &\leq \mathbb{P}\left(\leq c\log n \text{ vertical moves}\right) + \mathbb{P}\left(r > c\log n\right)\\ &\leq \frac{1}{n^{4c}} + \frac{4}{n^{\alpha-1}}\end{aligned}$$

For any $\alpha > 1$, we have $c = \alpha/\log 2 > \alpha$ and thus:

$$\mathbb{P}\left(\geq d\log n \text{ iterations}\right) \leq \frac{1}{n^{4\alpha}} + \frac{4}{n^{\alpha}} \leq \frac{5}{n^{\alpha}}$$

This concludes the proof. □

11.2 Universal Hashing

Consider a universe U of keys of items that we want to store in a hash table. A hash function is a function $h\colon U \to V$ where $V = \{0, 1, \ldots, m-1\}$ and $|U| > m$. In the worst case, an n-element hash table can take $\Omega(n)$ time to search for an element; for example if all elements are hashed to the same value and end up in the same linked list. This is true no matter the hash function chosen. The idea of universal hashing is to chose a random hash function to circumvent the deterministic worst case. For that, we need to look at families of hash functions from which we will make the random choice.

Definition 11.1

A family $\mathcal{H}$ of hash functions $h\colon U \to V$ is *universal* if

$$\mathbb{P}\big(h(x) = h(y)\big) \leq \frac{1}{m}$$

for all $x, y \in U$ with $x \neq y$ where the hash function h is chosen uniformly at random in $\mathcal{H}$.

The use of universal families of hash functions allows to cut down on the length of the linked lists in the hash table. Here is a result on the expected number of collisions:

Theorem 11.3. In an n-element hash table S with the hash function h chosen uniformly at random from a universal family, for every element $x \in U$ of the universe, we have:

$$\mathbb{E}\,|\{y \in S \mid h(x) = h(y)\}| \leq \begin{cases} n/m & \text{if } x \notin S \\ 1 + (n-1)/m & \text{if } x \in S \end{cases}$$

Proof. Let $y_1, y_2, \ldots, y_n$ be the elements of S and let X_i be the indicator variable of the event $h(y_i) = h(x)$. Since $\mathcal{H}$ is universal, we have $\mathbb{E}\,X_i = \mathbb{P}(X_i = 1) \leq 1/m$ if $x \neq y_i$. Now, if $x \notin S$, then:

$$\mathbb{E}\sum_{i=1}^{n} X_i = \sum_{i=1}^{n} \mathbb{P}(X_i = 1) \leq \frac{n}{m}$$

On the other hand, if $x = y_j$ for some j, then

$$\mathbb{E}\sum_{i=1}^{n} X_i = \sum_{i=1}^{n} \mathbb{P}(X_i = 1) = 1 + \sum_{\substack{1 \le i \le n \\ i \ne j}} \mathbb{P}(X_i = 1) \le 1 + \frac{n-1}{m} \ ,$$

which concludes the proof. □

Universal families of hash functions are clearly useful, but we have not yet answered the question whether any actually exists. This question is quite easily answered in the affirmative via the probabilistic method. A harder question is whether we can find a universal family that we can easily sample from and whose hash functions are simple to compute. Fortunately, the answer to this question is also yes. There are a few constructions known, here is one of them:

Let us assume that $U = \{0, 1, \dots, u-1\}$ and choose a prime number $p \ge u$. Then, for integers $0 \le a, b < p$, we define the hash function

$$h_{a,b}(x) = \big((ax + b) \bmod p\big) \bmod m$$

and define the family $\mathcal{H} = \{h_{a,b} \mid 0 \le a, b < p\}$.

Theorem 11.4. The family $\mathcal{H}$ is universal.

Proof. Let $x, y \in U$ with $x \ne y$.

For every pair $(u, v) \in \{0, \dots, p-1\}^2$ with $u \ne v$, there is a unique pair $(a, b) \in \{1, \dots, p-1\} \times \{0, \dots, p-1\}$ with $ax + b \equiv u \mod p$ and $ay + b \equiv v \mod p$. In fact, we have the formulas $a \equiv (v-u)\cdot(y-x)^{-1} \mod p$ and $b \equiv u - ax \mod p$.

It thus suffices to upper bound the number of pairs (u, v) with $u \not\equiv v \mod p$ but $u \equiv v \mod m$. If we fix a $u \in \{0, \dots, p-1\}$, then there are at most $\lceil p/m \rceil - 1 \le (p-1)/m$ possible values for v. In total, we thus have at most $p(p-1)/m$ such pairs. Since each pair corresponds to a unique hash function $h_{a,b}$ in $\mathcal{H}$, we have

$$\mathbb{P}\big(h_{a,b}(x) = h_{a,b}(y)\big) \le \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m} \ ,$$

which shows that $\mathcal{H}$ is a universal family. □

We now turn to the question whether we can completely avoid collisions. In classical hash tables, it turns out that can only be guaranteed when $m \ge n^2$, which leads to a prohibitive space complexity of $\Omega(n^2)$. It *is*, however, possible to get rid of collisions if we introduce a second level of hashing for each slot. This is a technique called *perfect hashing* and is able to guarantee a worst-case time complexity of searches of $O(1)$ while keeping an $O(n)$ space complexity. We will analyze the static variant, which does not support insertions or deletions, but dynamic variants also exist.

For the analysis of perfect hashing, we use Markov's inequality, which can be quite loose, but is often enough to prove asymptotic probability bounds. It is also the basis for more advanced bounds, including Chebyshev's inequality and the Chernoff bound.

Theorem 11.5 (Markov's inequality). Let X be a nonnegative random variable

and let $a > 0$. Then:

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}\, X}{a}$$

Lemma 11.2. In an n-element hash table S with hash function h chosen uniformly at random from a universal family, if $m \geq n^2$, then the probability of having no collisions is at least $1/2$.

Proof. Let $y_1, y_2, \ldots, y_n$ be the elements of S and let $X_{i,j}$ be the indicator variable of the event $h(y_i) = h(h_j)$. Defining X to be the number of collisions, we have:

$$\mathbb{E}\, X = \sum_{1 \leq i < j \leq n} \mathbb{P}(X_{i,j} = 1) \leq \frac{n(n-1)}{2} \frac{1}{m} \leq \frac{n^2}{2m}$$

Now, applying Markov's inequality (Theorem 11.5) with $a = n^2/m$ concludes the proof since $n^2/m \leq 1$ by assumption. □

The construction of a 2-level perfect hash table is as follows:

1. Pick a hash function $h_1 \in \mathcal{H}_m$ uniformly at random where $m = n$. Denote by ℓ_j the number of elements hashed to value j. If $\sum_{j=1}^{m} \ell_j^2 > cn$, then pick another hash function and check again.
2. For every $j \in \{0, \ldots, m-1\}$, pick a hash function $h_{2,j}$ uniformly at random from $\mathcal{H}_{m_j}$ where $m_j = \ell_j^2$. If there is a conflict for one of the hash functions $h_{2,j}$, pick another and check for conflicts again.

If this algorithm terminates, the perfect hash table allows searches in $O(1)$ time since there are no conflicts in the second level. Moreover, its space complexity is $O(n) + \sum_{j=1}^{m} O(\ell_j^2) = O(n)$.

Theorem 11.6. A perfect hash table can be constructed in time $O(n \log^2 n)$ with high probability.

Proof. Picking and checking a hash function in $\mathcal{H}_m$ can be done in time $O(n)$. Denoting by $X_{i,j}$ the indicator function of the event $h_1(y_i) = h_i(y_j)$, we have

$$\mathbb{E} \sum_{j=0}^{m-1} \ell_j^2 = \sum_{i=1}^{n} \sum_{j=1}^{n} \mathbb{E}\, X_{i,j} \leq n + 2\frac{n(n-1)}{2} \frac{1}{m} \leq \frac{c}{2} n$$

for some constant $c > 0$. Application of Markov's inequality (Theorem 11.5) now shows $\mathbb{P}\big(\sum_{j=0}^{m-1} \ell_j^2 > cn\big) \leq 1/2$. We can thus see the picks of h_1 as i.i.d. coin flips with success probability $p \geq 1/2$. The number of coin flips until the first success is $O(\log n)$ with high probability.

Picking and checking each $h_{2,j}$ can be done in time $O(\ell_j)$. An application of the Chernoff bound (Theorem 11.1) shows $\ell_j = O(\log n)$ with high probability. Picking one set of $h_{2,j}$ thus takes time $O(n \log n)$ with high probability. As above, Lemma 11.2 shows that we need to pick the $h_{2,j}$ at most $O(\log n)$ times with high probability. This means that constructing the second level is done in

$O(n \log^2 n)$ time with high probability. □

April 20, 2026

Lecture 12: Distributed Algorithms I

The analysis of distributed systems is of high importance due to the ubiquity of systems in which communication is necessary: from the Internet all the way to systems-on-chip. One of the most difficult tasks in this analysis is that of modeling the system. In contrast to centralized computation, no single standard model exists and a small change in the model assumptions can lead to widely different properties.

12.1 Modeling: Synchronous Message Passing

One of the simplest models of distributed computation is that of synchronous message passing. In this model, processes are assumed to repeatedly execute rounds in a lock-step fashion. The steps in a round are send–receive–compute: first all processes send a set of messages, then they receive the message just sent, and then they perform a local computation. Most commonly, the local processes are assumed to be computationally unbounded. This assumption is made since local computations are rarely very heavy in the type of algorithms normally studied. An exception is the assumption of unbreakable cryptographic primitives, for which a Turing machine model would not be very useful either.

Formally, a distributed system in this model has:

- a directed or undirected graph $G = (V, E)$
- a message alphabet M with $\perp \notin M$
- for every process $p \in V$:
  - a set States_p of local states
  - a nonempty set $\text{Start}_p \subseteq \text{States}_p$ of initial states
  - a message-generating function $\text{Msgs}_p \colon \text{States}_p \times \text{Out}_p \to M \cup \{\perp\}$
  - a state-transition function $\text{Trans}_p \colon \text{States}_p \times (M \cup \{\perp\})^{\text{In}_p} \to \text{States}_p$

We construct an *execution* as follows. We initialize the local state of every process p to some initial state in Start_p. Then, in each round $r \geq 1$:

1. Apply the message-generating functions Msgs_p to the current local state of each process p to find the message sent over every link (p, q).
2. Gather all messages sent to each process q and apply the state-transition function Trans_q to find the next local state.

The main complexity measures are time and communication complexity. Time complexity is measured in the number of rounds until the desired state. Communication complexity is most often measured in total number of messages sent. Sometimes it is also measured in the total number of bits of messages sent.

A desirable property of our algorithms is that they actually halt, *i.e.*, stop executing. Formally, a halting state is one in which no messages are sent and no state transition from the halting state is possible.

Models both with and without process identifiers exist. In models without process identifiers, the processes know their neighbors only by their port numbers. If they are assumed, process identifiers take the form of unique identifiers (UIDs), which are unique in the graph and are often assumed to be encoded in $O(\log n)$ bits.

If we allow randomized algorithms, we allow the state-transition function to return a probability distribution on the set of states instead of a single state.

12.2 Breadth-First Search

The problem of constructing a distributed breadth-first search (BFS) tree can be formalized as follows: each process maintains a parent variable, which can hold the UID of a process. A unique root process should encode itself as its parent. Any other process should encode its parent in the BFS tree rooted at the root.

For this problem, we assume a connected undirected graph G. We assume the existence of process UIDs and the existence of a unique distinguished leader process p_0, but specific knowledge of the graph G (*e.g.*, number of nodes, diameter). The assumption of the existence of a leader process might seem strong, but a simple flooding algorithm can determine a leader in $O(\mathrm{diam}(G))$ rounds.

The SynchBFS algorithm has some set of processes be marked at each round. Initially, only the leader process p_0 is marked and all parent variables are null except for the leader process, which has itself set as its parent. In every round, each process that became marked in the last round sends a search message to all its neighbors. Then, every unmarked process that received a search message from one of its neighbors chooses one of these neighbors to be its parent and becomes marked.

Theorem 12.1. The SynchBFS algorithm constructs a BFS tree in $O(\mathrm{diam}(G))$ rounds and sends $O(|E|)$ messages.

Proof. We have the following invariant at the end of every round $r \geq 1$, which can be proved by induction on r: for every $1 \leq d \leq r$, every process at distance d from p_0 is marked and has its parent set to a process at distance $d-1$ from p_0.

The correctness of the constructed tree and the bound on the time complexity then follow from this invariant at round $r = \mathrm{diam}(G)$. The message complexity bound follows from the fact that every process p sends exactly $\deg(p)$ messages. The total number of messages is thus

$$\sum_{p \in V} \deg(p) = 2|E| \ ,$$

which concludes the proof. □

Once we constructed a BFS, we can broadcast messages through the tree in $O(\mathrm{diam}(G))$ rounds. We can even add a routing table to the nodes to reduce the message complexity to $O(\mathrm{diam}(G))$ for each point-to-point message. This is obvious for messages sent by the root process, but every other process can first send the to-be-sent message to the root, which will then forward it to the right process. Termination can be implemented since we can locally detect whether a process is a leaf in the BFS tree: in the round following their search message, they don't receive another search message. A termination signal can then be propagated up the BFS tree.

12.3 Maximal Independent Set

We now turn to the problem of constructing a maximal independent set (MIS). An independent set is a set of vertices that does not contain any two neighbors. We formalize a distributed MIS construction by requiring the existence of a local Boolean variable that indicates whether the process is inside the MIS or not.

We again assume a connected undirected graph G. The algorithm that we will study does not need UIDs or any other knowledge of the graph G. It does, however,

work with real-valued variables. To achieve a finite bit complexity, one can show that using $O(\log n)$ bits of the real numbers is enough. To utilize this, one needs knowledge of n, or of an upper bound on n.

In the LubyMIS algorithm [9], each process maintains a list of remaining neighbors, initialized to all its neighbors in the graph. It proceeds in phases of three rounds each:

1. In the first round, each active process sends a uniformly chosen random value in the interval $[0, 1]$ to all neighbors. Each active process then determines whether it is a *winner*, *i.e.*, whether it has a value strictly larger than all its neighbors.
2. In the second round, each winner notifies its neighbors of the fact that they are *losers*.
3. In the third round, each winner joins the MIS. Then all winners and all losers stop executing the algorithm and become inactive. All neighbors of losers remove the losers from their list of remaining neighbors.

For analysis purposes, we define a sequence $G_0, G_1, \ldots$ of subgraphs of G. The graph $G_\phi = (V_\phi, E_\phi)$ is the subgraph of G induced by the set of active processes at the end of phase $\phi \geq 1$. Once all processes became inactive, the graphs G_ϕ are empty. Initially, we set $G_0 = G$.

Lemma 12.1. The expected number of edges removed in phase ϕ is at least $|E_{\phi-1}|/2$.

Proof. We say that process p *single-handedly* kills edge $\{q, r\}$ from q's side if p is a neighbor of q and process p's value is maximal among those of $N(p) \cup N(q)$.

Now consider any edge $\{p, q\} \in E_{\phi-1}$. The probability that process p's value is maximal among the values in $N(p) \cup N(q)$ is at least $\frac{1}{\deg(p)+\deg(q)}$. In this case, process p single-handedly kills $\deg(q)$ edges from q's side. Each edge in $E_{\phi-1}$ can be single-handedly killed at most twice; once from each side. Using $\max\{a, b\} \geq \frac{a+b}{2}$, the expected number X of edges removed in phase ϕ is:

$$\mathbb{E}\, X \geq \sum_{\{p,q\} \in E_{\phi-1}} \frac{1}{2} \left(\frac{\deg(p)}{\deg(p) + \deg(q)} + \frac{\deg(q)}{\deg(p) + \deg(q)} \right) = \frac{|E_{\phi-1}|}{2}$$

This concludes the proof. □

Theorem 12.2. If the LubyMIS algorithm terminates, it has constructed an MIS. Furthermore, the expected number of rounds until termination is $O(\log n)$.

Proof. Let I_ϕ be the set of processes that joined the constructed MIS in phase ϕ and let $J_\phi = \bigcup_{\psi=1}^{\phi} I_\psi$ be the constructed MIS up to the end of phase ϕ. We have the following invariant at the end of every phase $\phi \geq 1$: The set J_ϕ is an independent set in G. Furthermore, the processes that have been removed in phase ϕ are those processes in $J_\phi \cup N(J_\phi)$ that have not already been removed in earlier phases, which means that $V_\phi = V \setminus (J_\phi \cup N(J_\phi))$.

If the algorithm terminates, independence of the constructed set now directly follows from the first part of the invariant. Maximality of the constructed independent set follows from the second part of the invariant.

We now turn to the time-complexity bound. Markov's inequality (Theorem 11.5) with $a = 3|E_{\phi-1}|/4$ and Lemma 12.1 gives:

$$\mathbb{P}\left(|E_\phi| \geq \frac{3}{4}|E_{\phi-1}|\right) \leq \frac{2}{3} \quad \Rightarrow \quad \mathbb{P}\left(|E_\phi| < \frac{3}{4}|E_{\phi-1}|\right) \geq \frac{1}{3}$$

Let us call phase ϕ a *success* if $|E_\phi| < \frac{3}{4}|E_{\phi-1}|$. Starting from $|E_0| \leq n^2$, we have $|E_\phi| < 1$, *i.e.*, $E_\phi = \emptyset$ after we have had at least

$$s > \frac{2}{\log \frac{4}{3}} \log n = c \log n$$

successes. If $E_\phi = \emptyset$, then the algorithm terminates in phase $\phi + 1$ at the latest. Now, using the Chernoff bound (Theorem 11.1) for $d \log n$ coin flips with success probability $1/3$ gives

$$\mathbb{P}\left(E_{\lfloor d \log n \rfloor} \neq \emptyset\right) \leq \mathbb{P}\left(s \leq c \log n\right) \leq e^{-\alpha \log n} = \frac{1}{n^\alpha}$$

where $\alpha = \delta^2/6$ and $1 - \delta = 3c/d$. In particular, for a given $\alpha > 0$ we can find an appropriate d. □

The time complexity of LubyMIS is asymptotically almost optimal, as there is a lower bound of $\Omega\left(\sqrt{\log n / \log \log n}\right)$, even if UIDs are allowed.

12.4 Coloring of Paths

We want to color a path graph with $\Delta + 1 = 3$ colors. Formally, we give every process a local color variable that can take the values 1, 2, or 3. Initially, the variables are initialized to null. The goal is to have neighboring nodes pick different colors. We assume no knowledge of the number n of processes, but do assume UIDs. We will study three different algorithms for this problem.

The first algorithm, which we call LocalLeader, has processes exchange their colors and UIDs. Undecided processes then decide whether they are a local leader, *i.e.*, whether their UID is larger than those of its undecided neighbors. A local leader then picks an arbitrary color that has not yet been picked by a neighbor.

Theorem 12.3. The LocalLeader algorithm constructs a 3-coloring of a path of length n in $O(n)$ rounds.

Proof. Correctness follows from a simple invariant. For the time complexity, we note that, before termination, there is at least one local leader. □

This bound is asymptotically tight: in the case of increasing UIDs along the path, the LocalLeader algorithm takes $\Omega(n)$ rounds.

The second algorithm, which we call RandomColor, has every undecided process pick a random color and send it to its neighbors. Then, if the received colors are locally consistent, it adopts its current color definitely.

Theorem 12.4. The RandomColor algorithm constructs a 3-coloring of a path of length n in $O(\log n)$ rounds with high probability.

Proof. The correctness of the algorithm again follows from a simple invariant. For the time-complexity bound, we note that every process has a probability of at least $1/3$ to pick a color in every round in which it is active. By an application of the Chernoff bound, we conclude that every process pick a color in $O(\log n)$ rounds with high probability. An application of the union bound now concludes the proof. □

The third algorithm, which we call IDReduction, starts with a large set of colors, which it then reduces over time until only three are left. The initial, large, set of colors is the set of UIDs. We can allow the UIDs to be very large positive integers; for simplicity we assume that they are bounded by a polynomial in n. Then, each process sends its current color to its right neighbor. (We do need to assume that our path is directed for now.) It then chooses a new color equal to $2i + b$ where i is the position of the first bit where the two IDs differ, and b is the bit of its own color at that position. This can reduce the number of colors to at most 6, since 7 is the smallest positive integer m for which $2\log_2 m + 1 < m$ We then use the LocalLeader algorithm to find a 3-coloring in $6 - 3 = 3$ more rounds. In fact, we didn't need globally unique IDs, but only locally unique ones.

Theorem 12.5. The IDReduction algorithm constructs a 3-coloring of a directed path of length n in $O(\log^* n)$ rounds.

Proof. For correctness, we prove the invariant of the first phase of the algorithm that the IDs form a $f^r(N)$ coloring at the end of round $r \geq 1$, where $f(m) = 2\log_2 m + 1$ and N is the largest UID, which is $\leq n^k$ for some k by assumption. For the induction step, we assume by contradiction that $2i + b = 2i' + b'$ for the new colors of two neighbors in the path. We then must have $b = b'$ and $i = i'$. In particular, i is the position of the first bit in which the old IDs differed. But since they did differ, we have $b \neq b'$, a contradiction.

The time complexity bound immediately follows from the invariant at round number $r = O(\log^* n)$. □

May 4, 2026

Lecture 13: Distributed Algorithms II

After having studied synchronous systems, we now turn our attention to asynchronous systems, in which we don't have a common time base. There are two main reasons for asynchronous models: when some processes or messages can be very slow, *e.g.*, when modeling the Internet, and when some processes or messages can be very quick and we don't want to wait for the slowest one.

13.1 Modeling: Asynchronous Message Passing

A configuration is a collection of one local state for each process and the state of the communication system. In our case, the state of the communication system is the multiset of messages that were sent but not yet received. A message is a pair (p, m) of a process p and a message value $m \in M$.

Computational steps of processes now happen in a possibly non-synchronous manner. Each step of a process includes: the reception of one or zero messages, a local state change, and the sending of a set of messages to its neighbors. Contrary to the synchronous model, local steps thus proceed in a receive–compute–send manner. A step of process p is triggered by an event $e = (p, m)$ where $m \in M \cup \{\perp\}$. If $m = \perp$, then no message is received by process p in the step. Otherwise, process p can read the message contents and have its state transition depend on it. The message (p, m) is then removed from the set of in-transit messages and the set of messages sent by p in the step is added to it. Given a configuration C and an event $e = (p, m)$ where either $m = \perp$ or (p, m) is an in-transit message, then we can apply the event e to configuration C and determine the successor configuration $e(C)$. Formally, for the description of the system, we need:

- a directed or undirected graph $G = (V, E)$
- a message alphabet M with $\perp \notin M$
- for every process $p \in V$:
  - a set States_p of local states
  - a nonempty set $\text{Start}_p \subseteq \text{States}_p$ of initial states
  - a state-transition function $\text{Trans}_p \colon \text{States}_p \times (M \cup \{\perp\}) \to \text{States}_p$
  - a message-generating function $\text{Msgs}_p \colon \text{States}_p \to 2^{\text{Out}_p \times M}$

An execution is a sequence of configurations such that every process starts in one of its initial states and each successor configuration is the result of the application of an applicable event. An execution is admissible if:

- No message is received by p more often than it was previously sent to p. (Safety)
- Every message is eventually received. (Communication liveness)
- Every process takes infinitely many steps. (Process liveness)

We would want our algorithms to be well-behaved in all admissible executions. In the synchronous model, there was a single execution for each initial configuration. This is now very different in the asynchronous model: the number of admissible executions is always uncountable if $|V| \geq 2$. Due to this non-determinism, people often find it useful to think about algorithm design as a two-player game: first the algorithm designer

presents the algorithm, then the adversary tries to find an admissible execution in which the algorithm misbehaves. People are also sometimes tempted to convert the non-determinism into a probabilistic choice. However, due to the difficulty of coming up with realistic probability distributions for message-delivery and scheduling choices, this only makes sense in very specific cases.

Message complexity is measured the same way as in synchronous systems: by counting the total number of messages sent. It is not immediately obvious how to define time complexity in an asynchronous system, however. The approach that is usually taken is to add real times in $\mathbb{R}_+$ to each configuration in each admissible execution in a nondecreasing fashion, with the additional constraints that the initial configuration is at time 0 and that the real-time difference between process activations, as well as a message send and its reception, is at most 1. The time complexity is then defined as the supremum of all real times of the desired event when real-time tagging according to the above rules.

13.2 Breadth-First Search

We return to the first problem that we studied for synchronous message passing—the construction of a BFS tree—and see whether and how we can solve it in asynchronous systems. As in Section 12.2, we assume an undirected connected graph G and the existence of a leader node p_0.

We can't use the SynchBFS algorithm directly in an asynchronous system because it relied on the fact that, after d rounds, all processes at distance at most d from p_0 received a search message. There is no way to make this time-based property work in an asynchronous system. We will thus need to pack some distance information into the messages that we send. One possibility is to send our current distance from p_0 to our neighbors and have them update their parent variable to us if we can offer them a shorter path than they currently have, akin to the Bellman–Ford algorithm.

This leads us to the definition of the AsynchBFS algorithm: Every process maintains a parent and a distance variable. Initially, all processes except p_0 have parent equal null and distance equal $+\infty$. The leader process p_0 initializes parent to itself and distance to 0. In the first step of p_0, it sends a message with content '0' to all its neighbors. Upon reception of a message, each process compares the received value m to its current distance d. If $m + 1 < d$, then it updates its parent variable to the process from which it received the value m and its distance to $m + 1$. It then sends this new distance to all neighbors.

Theorem 13.1. The AsynchBFS algorithm constructs a BFS tree in $O(\mathrm{diam}(G))$ time and sends $O(n \cdot |E|)$ messages.

Proof. Denote by dist_p the value of the distance variable at process p. For correctness of the constructed tree, we can show the following invariants:

1. If $\mathrm{dist}_p < +\infty$, then dist_p is the length of some path from p_0 to p.
2. Every message sent by process p is the length of some path from p_0 to p.
3. For every edge $\{p, q\} \in E$, either $\mathrm{dist}_q \leq \mathrm{dist}_p + 1$ or process p has sent a message with content dist_p to q.

The time bound follows from invariant (3) since it implies that, after time d_p,

we have $\text{dist}_p = d_p$ where d is the distance of p from p_0. The message-complexity bound follows from the fact that every process sends at most n different values, since there are at most n possible distances. □

We proved that the AsynchBFS algorithm stabilizes to a BFS tree, but we have not discussed how to make processes decide and terminate in finite time. This can be achieved by adding acknowledgment messages and propagating a termination signal to the leader process once the exploration of all neighbors is done. The bookkeeping involved in this is a bit tedious, but it can be done in the same asymptotic complexity.

13.3 Modeling: Process Faults

One of the fundamental reasons for distribution is fault-tolerance. That is, we want to have our system function even if some of its components fail. Faults come in all kinds of varieties. For now, we will focus on crash faults, *i.e.*, having process stop taking steps during executions. These faults are difficult to tolerate in asynchronous systems, since it can be impossible to distinguish a slow from a crashed process.

To every execution, we associate a set F of faulty processes with $|F| \leq f$ such that:

- No message is received by p more often than it was previously sent to p. (Safety)
- Every message sent to a non-faulty process is eventually received. (Communication liveness)
- Every non-faulty process takes infinitely many steps. (Process liveness)

We assume that F is minimal with this property. That is, we do not flag a process faulty if it takes infinitely many steps and receives all messages sent to it.

13.4 Impossibility of Consensus in Asynchronous Systems with Process Faults

It turns out that a large class of problems are unsolvable in asynchronous systems with process faults, even for complete graphs G, which we assume in the following. A particularly striking example is the consensus problem. In the consensus problem, every process starts with an initial value $v_p \in \{0, 1\}$. We want all processes to decide on a common value. For this, we equip every process with a write-once decision variable. In every execution of a consensus algorithm, we require the following three properties:

- No two decision values are different. (Agreement)
- If all initial values are the same, then this value is the only possible decision value. (Validity)
- All non-faulty processes decide. (Termination)

We start with a general lemma about asynchronous systems. For its formulation, we extend the application of an event to sequences of events, which we call schedules. If a schedule σ is applicable to a configuration C, then we write $\sigma(C)$ for the resulting configuration if σ is finite and for the resulting execution postfix if σ is infinite.

Lemma 13.1. If σ_1 and σ_2 are applicable to configuration C and the sets of processes taking steps in σ_1 and σ_2 are disjoint, then:

1. σ_2 is applicable to $\sigma_1(C)$

2. σ_1 is applicable to $\sigma_2(C)$
3. $\sigma_2(\sigma_1(C)) = \sigma_1(\sigma_2(C))$

For the analysis of executions of consensus algorithms, we introduce the notion of valency of a configuration. A configuration C is 0-valent if all executions that contain C decide 0, and 1-valent if all execution that contain C decide 1. In this case, we say that C is univalent. A configuration that is not univalent is bivalent.

Our goal is to construct, for every algorithm, an admissible execution in which processes can't decide. For that, we construct an execution whose configurations are all bivalent. We start with the initial configuration:

Lemma 13.2. There is a bivalent initial configuration.

Proof. Without loss of generality, assume $V = \{1, 2, \ldots, n\}$. We consider the initial configurations I_k in which all processes $p \leq k$ have initial value 1 and all processes $p > k$ have initial value 0. By validity, I_0 is 0-valent and I_n is 1-valent.

Assume by contradiction that all initial configurations are either 0-valent or 1-valent. Then there must be some k such that I_{k-1} is 0-valent and I_k is 1-valent. Take any admissible infinite schedule σ that is applicable to I_{k-1} in which process k doesn't take any steps. Then, since I_{k-1} is 0-valent, the decision value in execution $\sigma(I_{k-1})$ is 0.

Now, since the only difference between I_{k-1} and I_k is the local state of process k and process k doesn't participate in schedule σ, the same schedule is also applicable to I_k. Furthermore, the only difference in the executions $\sigma(I_{k-1})$ and $\sigma(I_k)$ is the local state of process k. Hence, the decision values must be the same in both executions. But then $\sigma(I_k)$ decides 0, which is in contradiction to I_k being 1-valent. □

For the inductive construction of the bivalent execution, we need the following technical lemma:

Lemma 13.3. Let C be a bivalent configuration and let $e = (p, m)$ be an event applicable to C. If $\mathcal{C}$ is the set of configurations reachable from C without applying e and $\mathcal{D} = \{e(E) \mid E \in \mathcal{C} \text{ and } e \text{ is applicable to } E\}$, then $\mathcal{D}$ contains a bivalent configuration.

Proof. Assume by contradiction that there are no bivalent configurations in $\mathcal{D}$.

The event e is applicable to all configurations in $\mathcal{C}$. This is obvious if $m = \bot$. If $m \neq \bot$, then the message m is still in transit to process p in configuration $E \in \mathcal{C}$, and can thus be received by process p.

We next argue that there is a 0-valent and a 1-valent configuration in $\mathcal{D}$. Let E_v be a v-valent configuration reachable from C for $v \in \{0, 1\}$. Both E_0 and E_1 exist since C is bivalent. If $E_v \in \mathcal{C}$, then set $F_v = e(E_v) \in \mathcal{D}$. If $E_v \notin \mathcal{C}$, then E_v has a predecessor configuration for which e was applied, so $F_v \in \mathcal{D}$.

Now, call two configuration neighbors if one is a direct successor of the other. Since every $D \in \mathcal{D}$ is of the form $D = e(C)$ with $C \in \mathcal{C}$, $\mathcal{C}$ is connected with respect to the neighbor relation, and C is bivalent, there are two neighbors $C_0, C_1 \in \mathcal{C}$ such that $D_0 = e(C_0)$ is 0-valent and $D_1 = e(C_1)$ is 1-valent. Without loss of generality, let $C_1 = e'(C_0)$ where $e' = (p', m')$. We distinguish two cases:

1. If $p' \neq p$, then $D_1 = e(D_0)$ by Lemma 13.1. This is a contradiction to the fact that D_0 and D_1 have opposite valencies.
2. If $p' = p$, then let σ be any finite schedule in which p doesn't take any steps and for which all non-p processes have decided in configuration $A = \sigma(C_0)$. By Lemma 13.1, the schedule σ is also applicable to the configurations C_1, D_0, and D_1. Moreover, $e(A) = e(\sigma(C_0)) = \sigma(e(C_0)) = \sigma(D_0)$ and $e(e'(A)) = e(e'(\sigma(C_0))) = \sigma(e(e'(C_0))) = \sigma(e(C_1)) = \sigma(D_1)$. Since D_0 and D_1 have opposite valencies, the configuration A is a bivalent, a contradiction to the fact that some process has already decided in A. □

We can now conclude:

Theorem 13.2 (Fischer, Lynch, Paterson [10])**.** There is no consensus algorithm in asynchronous message passing with $f \geq 1$ crash faults.

Proof. Assume by contradiction that there is a consensus algorithm.

By Lemma 13.2, there is a bivalent initial configuration C_0. From this initial configuration, we construct an execution in phase. The configuration at the end of each phase will be bivalent by construction. We then show that the constructed execution is admissible, deriving the desired contradiction.

For constructing the phases, we maintain a queue of processes. Initially, all processes are in the queue in arbitrary order. A phase ends when the process at the head of the queue receives its oldest message that was in transit at the start of the phase. If no such message exists, the phase ends when it takes its first step in the phase. The process is then moved to the end of the queue. Let $e = (p, m)$ be the event that ends the current phase. By Lemma 13.3, we can end the current phase in a bivalent configuration.

Since every phase contains at least one step, this leads to an infinite execution of the algorithm. By construction, no process can ever decide, since there are bivalent configurations arbitrarily late in the execution. Furthermore, every message sent is eventually received by construction of the phases. This is a contradiction to the fact that the algorithm decides in every admissible execution. □

13.5 Asynchronous Rounds

The proof of Theorem 13.2 is kind of tricky in that we needed to make sure that the constructed execution was admissible. The hard part about that is the fact that validity of a liveness condition is not a local problem of the configurations; we can only decide it for infinite executions. In general, reasoning about liveness properties is hard, which is why there has been a push towards models that capture important aspects, like asynchrony, but which do not have liveness conditions, only safety conditions.

For asynchronous message passing with crash faults, this often comes in the form of asynchronous rounds. These are rounds constructed in an asynchronous system. Every process sends its round-r message to all other processes and proceeds to the next round $r + 1$ when it has received $n - f$ messages (including its own). Waiting for more than $n - f$ messages might block the process, since there could be f process that do not send a round-r message; so this choice of parameter is optimal.

It is not clear why one would want to construct asynchronous rounds; whether they are useful. If one looks at algorithms in asynchronous models, however, it is

exceedingly rare to find one that does not proceed in rounds.

With the above construction like this, we get a Heard-Of set $\mathrm{HO}_p(r)$ for every round r and every process p [12]. They satisfy the following two properties:

1. $p \in \mathrm{HO}_p(r)$
2. $|\mathrm{HO}_p(r)| \geq n - f$

Note, however, that no stability from one round to next exists: processes whose messages are slow in one round can be fast in the next. The Heard-Of sets of a given round describe a Heard-Of graph HO, which is the directed graph on V such that $\mathrm{In}_p = \mathrm{HO}_p$.

Since the Heard-Of model can be implemented in asynchronous message passing, the impossibility result of Theorem 13.2 also implies impossibility in the above Heard-Of model. There is, however, a much easier direct proof of this fact:

Theorem 13.3. There is no consensus algorithm in the f-receive-omission Heard-Of model if $f \geq 1$.

Proof. The existence of a bivalent initial configuration follows just like in the proof of Lemma 13.2, by silencing one process.

Let us write $\mathrm{HO} \sim \mathrm{HO}'$ if they only differ in the Heard-Of set of a single process. In this case, it is impossible to have the resulting configurations be of opposite valencies: If $\mathrm{HO}(C)$ is 0-valent and $\mathrm{HO}'(C)$ is 1-valent, then we can choose a schedule σ in which the single process that has a possibly different local state in $\mathrm{HO}(C)$ and $\mathrm{HO}'(C)$ is forever silenced. So all other processes decide the same value in both executions, so both 0 and 1, a contradiction.

Since the set of Heard-Of graphs is connected with respect to the relation $\sim$, it is impossible to have a bivalent configurations whose successors are all univalent. There hence is an infinite executions in which all configurations are bivalent, a contradiction. □

May 11, 2026

Lecture 14: Distributed Algorithms III

We have seen that consensus is impossible in asynchronous systems with process faults. We now turn to the question of where to go from here. Is it just too hard to achieve consensus? Is the model too unrealistic to capture practical systems? There is, of course a non-trivial modeling question that has no definitive answer as of yet. But it also turns out that consensus *can* be achieved if we relax the problem definition just a bit. In what follows, we assume an upper bound of $f < n/2$ on the number of crashed processes in each execution. If $f \geq n/2$ crashes are possible, then the system can be disconnected and many reasonable forms of consensus become impossible.

14.1 Approximate Consensus

We first study what happens if we relax the Agreement condition by just a bit:

- For any two decision values y_p and y_q, we have $|y_p - y_q| \leq \varepsilon$. ($\varepsilon$-Agreement)

It turns out that not only this approximate consensus is solvable in asynchronous systems, it is solvable with a very easy class of algorithms: An averaging algorithm is one that updates its value to some average of the $n - f$ received values in each round. This guarantees that the new value is inside the convex hull of the set of received values.

Let $\alpha \in [0, 1]$. We call an averaging algorithm α-safe if the new value doesn't go too close to the boundary of the convex hull of received values: the new value y_p satisfies $m + \alpha\Delta \leq y_p \leq M - \alpha\Delta$ where m and M are the minimal and maximal received values, and $\Delta = M - m$. Many averaging algorithms are α-safe:

Lemma 14.1. Every averaging algorithm with weights $\geq \alpha$ is α-safe.

Proof. Let $x_1, x_2, \ldots, x_k$ be the values received by process p in round r. Assume that these values are ordered non-decreasingly, *i.e.*, $x_1 = m$ is the minimal and $x_k = M$ is the maximal value. Then:

$$y_p = \sum_{i=1}^{k} a_i \cdot x_i \leq \alpha m + (1 - \alpha)M = M - \alpha\Delta$$

We show $y_p \geq m + \alpha\Delta$ in the same way. □

Denote by Δ_r the maximum distance between values at the end of round r. Initially we have $\Delta_0 \leq 1$ and we achieved ε-Agreement when $\Delta_r \leq \varepsilon$. It would be convenient if we had a contraction ratio $\beta < 1$ by which Δ_r shrinks from round to round: $\Delta_r \leq \beta\Delta_{r-1}$ for all rounds $r \geq 1$. This is not too much to hope for, as we will see in the next lemma. It uses a fundamental fact about asynchronous rounds with $f < n/2$, the non-split property: for any pair of non-crashed processes and any round, there exists one process that both hear from in that round. The common process that both hear from can change from round to round and it need not be the same for all pairs of processes.

Lemma 14.2. Every α-safe averaging algorithm has a contraction ratio of $1 - \alpha$.

Proof. Denote by $I_p = [m_p, M_p]$ the interval spanned by the values received by process p in the current round r. Let p and q be two processes and y_p and y_q their values at the end of the round. Without loss of generality let $m_p \leq m_q$. Then, because of the non-split property, we have $m_q \leq M_p$.

For process p, we calculate

$$y_p \geq m_p + \alpha(M_p - m_p) \geq (1-\alpha)m_p + \alpha m_q \geq (1-\alpha)m + \alpha m_q$$

and

$$y_q \leq M_q - \alpha(M_q - m_q) \leq (1-\alpha)M_q + \alpha m_q \leq (1-\alpha)M + \alpha m_q \ ,$$

where m and M are the minimal and maximal values of non-crashed processes at the start of the round, from which follows that $y_q - y_p \leq (1-\alpha)(M-m) = (1-\alpha)\Delta_{r-1}$. Similarly, for process q, we get

$$y_p \leq M_p - \alpha(M_p - m_p) \leq (1-\alpha)M_p + \alpha m_p \leq (1-\alpha)M + \alpha m_q$$

and

$$y_q \geq m_q + \alpha(M_q - m_q) \geq (1-\alpha)m_q + \alpha M_q \geq (1-\alpha)m + \alpha m_q \ ,$$

from which follows that $y_p - y_q \leq (1-\alpha)(M-m) = (1-\alpha)\Delta_{r-1}$. We thus have $|y_p - y_p| \leq (1-\alpha)\Delta_{r-1}$ for all p and q, which implies that $\Delta_r \leq (1-\alpha)\Delta_{r-1}$ and concludes the proof. □

We can now conclude that all α-safe averaging algorithms solve approximate consensus:

Theorem 14.1. Every α-safe algorithm achieves ε-agreement in $\left\lceil \log_{1/(1-\alpha)} \frac{\Delta}{\varepsilon} \right\rceil$ rounds.

Proof. By Lemma 14.2, we have:

$$\Delta_r \leq (1-\alpha)^r \Delta$$

Thus, whenever $r \geq \log_{1/(1-\alpha)} \frac{\Delta}{\varepsilon}$, we have

$$\Delta_r \leq e^{\log_{1/(1-\alpha)} \frac{\Delta}{\varepsilon} \cdot \log(1-\alpha)} \Delta = e^{-\log \frac{\Delta}{\varepsilon}} \Delta = \varepsilon \ ,$$

which concludes the proof. □

We talked about classes of averaging algorithms, which all solve approximate consensus, but which specific one should we choose? A natural choice is the algorithm that chooses the unweighted average of all received values. By Lemma 14.1, this algorithm is $(1 - 1/n)$-safe. Another choice, which turns out to be time-optimal in the worst case, is the algorithm that maximizes the safety parameter α: The Midpoint algorithm chooses the midpoint of the convex hull of received values and has the maximum possible safety parameter $\alpha = 1/2$.

14.2 Randomized Consensus

Allowing randomization also allows to circumvent consensus impossibility. Multiple randomized relaxations of consensus are possible. The most useful one still requires Agreement and Validity to hold for all executions, but requires only almost sure Termination, *i.e.*, with probability 1.

The BenOr algorithm [11] is one of the simplest randomized consensus algorithms. It proceeds in phases of two rounds each:

1. Send v_p to all other processes. Then, if all received values are equal to v, then set $w_p = v$, otherwise set $w_p = \bot$.
2. Send w_p to all other processes. Then, if all received values are equal to $v \in \{0,1\}$, then set $v_p = v$ and decide v. Otherwise, if one received value is $v \neq \bot$, then set $v_p = v$. Otherwise, set v_p to a random value in $\{0,1\}$.

One of the key insights into why this algorithm works is the fact that no two different values can remain after the first round:

Lemma 14.3. In any phase, if $w_p \neq \bot$ and $w_q \neq \bot$, then $w_p = w_q$.

Proof. Assume $w_p \neq w_q$. Then there exist sets P and Q of processes with $|P| \geq n - f$ and $|Q| \geq n - f$ with $v_p = w_p$ for all $p \in P$ and $v_q = w_q$ for all $q \in Q$ at the start of the phase. But this is impossible because then the total number of processes is at least $|P \cup Q| = |P| + |Q| \geq 2(n - f) > 2\frac{n}{2} = n$, a contradiction. □

The rest of the correctness proof is now in reach:

Theorem 14.2. The BenOr algorithm solves randomized consensus in expected time $O(2^n)$.

Proof. We show the three consensus properties: Agreement, Validity, and almost sure Termination.

We start with Validity. If $v_p = v$ for all non-crashed processes at the start of the first phase, then $w_p = v$ at the end of the first round and thus all non-crashed processes decide v at the end of the second round.

To show Agreement, assume that process p decides value v in phase ϕ, and assume that ϕ is the first phase in which any process decides. No other processes can decide a different value in phase ϕ because of Lemma 14.3. Furthermore, in the second round of phase ϕ, every non-crashed process q receives at least $n - 2f \geq 1$ times the value $w_p = v$. But this means that $v_q = v$ at the end of phase ϕ. All non-crashed processes hence start phase $\phi + 1$ with the same value $v_q = v$, which leads them to decide v in the second round of phase $\phi + 1$ by Validity and the memoryless nature of the phases.

We now show almost sure Termination and the $O(2^n)$ upper bound on the expected number of phases until decision by all correct processes. At the end of any phase in which there is one non-crashed processes that has not yet decided, by Lemma 14.3, all non-crashed processes start the next phase with the same value v_p if all random choices are equal to the unique non-$\bot$ value w_p. The probability of this happening is at least $1 - 1/2^n$. The number of phases until

decision by all correct processes is thus upper-bounded by a geometric random variable with success probability $1 - 1/2^n$. This concludes the proof. □

14.3 Byzantine Processes

The case of process crashes is challenging, but one can and does consider worse than crash faults. A Byzantine process can behave arbitrarily. This includes crashing, but also sending wrong messages, sending different wrong messages to different processes, following the algorithm initially and deviating later, etc. It turns out that both approximate consensus and randomized consensus can still be solved. This of course requires adapting the Agreement and Validity conditions to only cover correct, non-Byzantine, processes. It also requires us to lower the fault-tolerance threshold to $f < n/3$. Many problems are known to be unsolvable when $f \geq n/3$.

We need to employ some tricks to get there, of course. These come in the form of reliable broadcast and the witness technique, which combine into the following guarantees in every asynchronous round: every correct process receives at least $n - f$ messages from other processes and $n - f$ of them are common to *all* correct processes. Note that receiving more than $n - f$ messages can actually be a disadvantage here, because they can be from Byzantine processes.

For approximate consensus, we will need to throw away the f smallest and the f largest received values. Otherwise, a Byzantine process can make us exit the convex hull of values of correct processes. The only critical point is the non-split property of the resulting intervals. But, after trimming, there are at least $n - f - 2f = n - 3f \geq 1$ values in common at all processes. The intervals thus intersect.

For randomized consensus, we will need to introduce some thresholds to guard the rules for setting w_p and v_p. For setting $w_p = v$ in the first round of a phase, we place the condition that all but f received values are equal to v. For deciding in the second round of a phase, a value needs to make up all but f of the received values. For adopting a value, it needs to have been received at least $f + 1$ times.

The analog of Lemma 14.3 is true: Writing V for the number of times that value v was received, we have $V = V_c + V_r$ where V_c is the number of times that v appears in the set of $k \geq n - f$ common values received by all correct processes. We have $V_r \leq m - k$ where m is the total number of received values. But then $V_c = V - V_r \geq (m - f) - (m - k) = k - f = k/2 + (k - 2f)/2 \geq k/2 + (n - 3f)/2 > k/2$, which shows that there can be only one such value v.

A value that has been decided is adopted by all other correct processes in the same phase: For the deciding process, we have shown $V_c \geq k - f$ in the above calculation. Now, for the number V' of times that any other correct process has received value v, we have $V' \geq V_c \geq k - f \geq n - 2f > f$. That process hence adopts value v in the same phase.

References

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

[2] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[3] Mohsen Ghaffari. Distributed Algorithms. Lecture notes, ETH Zürich, 2023.

[4] Jukka Suomela. *Distributed Algorithms: An Intuitive Approach*. Online textbook, 2020.

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

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

[7] 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.

[8] 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.

[9] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[11] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, 1983.

[12] Bernadette Charron-Bost and André Schiper. The Heard-Of model: Computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[13] 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.

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

[15] 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.

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

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

[18] 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.

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

[20] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

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

[22] 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.

[23] 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.

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