



CS 602

Approximation Algorithm

Design algorithm that strictly runs in polynomial time ($n^{O(1)}$)
 Output is allowed to be a "provable" factor away from
 the optimal solution.

Maximization Problems

Ind Let
 Variable $\alpha > 1$ set of vertices such that no two of them are connected

α -approx if we output a solution that is $(\frac{1}{\alpha})$ to the optimal solution

Minimization problems

Hamiltonian Cycle
 Cycle that visits every vertex of G exactly once and returns back α -opt if we output a solution that is at most α of the optimal solution

Polynomial-time approximation solution

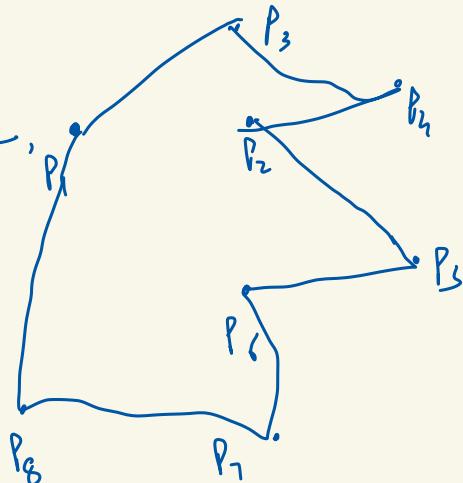
algorithm (of with some parameter $\epsilon > 0$) for any input, output a solⁿ within a factor $(1 + \epsilon)$ of the optimal solution.
 that runs in $n^{f(\frac{1}{\epsilon})}$ for some comparable f^n .
 (Running time is polynomial in $n, \text{some } \epsilon$)

Traveling Salesman Problem (TSP)

Given a list of cities ($P \subseteq \mathbb{R}^2$), and distances between each pair of cities, goal is to compute the shortest possible route that visits every city exactly once.

Decision Version

Given a length L ,
is it possible
to find a solution
of length L .



Graph:

A set of vertices, edges, weights $G = (V, E, w)$
 Visit all the vertices without repetition (minimize the sum
 of edge weights)

↓
Hamiltonian cycle problem

- * Hamiltonian cycle is NP-complete (Richard Karp)
- * No constant factor abs! is possible

Symmetric

Some edge weights
 on both
 directions

Asymmetric



Metric Space :

- $d(u, v) \geq 0$
- $d(x, y) = d(y, x)$

- Triangle inequality

$$d(x, y) + d(y, z) \geq d(x, z)$$

x . y
 . z

$\text{cost}(S) = \text{sum of the weights of the edges (union)}$

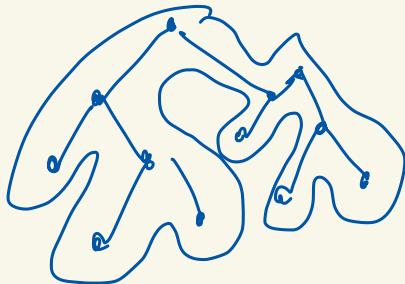
$S = \text{set of edges}$

same as finding the \min of Hamiltonian path
path cycle

Base Structure

- Min Spanning Tree [kruskal]

Due to triangular inequality, we can remove duplicates and cost is reduced



Do the DFS traversal and delete duplicates

Analysis

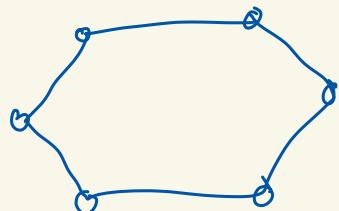
Every edge of the MST is traversed twice

$$\text{cost}(C) \leq 2 \times \text{cost}(\text{MST})$$

valid cycle

$$\text{cost}(\text{MST}) \leq \text{cost}(\text{opt})$$

$$\text{cost}(C) \leq 2 \times \text{cost}(\text{opt})$$



Q Can we do better?

$$I/O = G = (V, E) \xrightarrow{\quad} OPT_G$$

(i) Take a subset

Induced subgroup

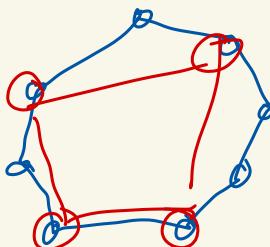
$$S \subseteq V$$

$$G_i(S)$$

$$\xrightarrow{\quad} OPT_S$$

$$OPT_S \leq OPT_G$$

Triangle
Inequality



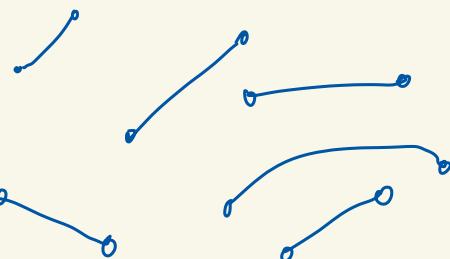
Property-2

Perfect Matching (can be computed in polynomial time)

Min cost AM

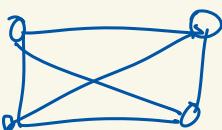
Perfect Matching with
smallest cost

(Polynomial Time
 $\sim O(n^2)$)



Eulerian Tour (Circuit)

vertices can be repeated



each node has
even degree than
this is possible

$$\sum d(v_i) = 2 \times [\epsilon]$$

↑
every edge connected twice

Q How many odd degree vertices we have?
even

We will add another edge for perfect matching

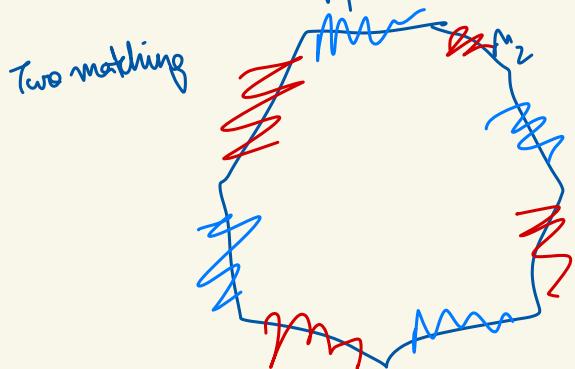
Compute Eulerian circuit

→ Vertex repetition is allowed, but we can delete the duplicates.

Analysis

$$\begin{aligned} \text{Cost } (C) &= \underbrace{\text{Cost } (\text{MST})}_{\leq \text{Cost } (\text{optimal})} + \text{Cost } (\text{Matching}) \\ \text{Cost } (C') &\leq \frac{1}{2} \text{Cost } (\text{optimal}) \end{aligned}$$

$\text{Cost } (C') \leq \frac{3}{2} \text{Cost } (\text{optimal})$



$$\begin{aligned} \text{Cost } (M_1) &\leq \text{Cost } (\text{optimal}) \\ \text{Cost } (M_2) &\leq \text{Cost } (\text{optimal}) \\ \text{Cost } (M') &\leq \frac{1}{2} \text{Cost } (\text{optimal}) \\ \text{Matching we choose because it is minimum} \end{aligned}$$

Metric TSP

- 2- α_{PR} [MST doubling]
- 1.5- α_{PR} (Christofides algorithm, 1976)

$$\downarrow \\ 1.5 - \varepsilon$$

$$\varepsilon = 10^{-30} \quad (2021)$$

No α_{PR} is possible if the distances are arbitrary.

$$G = (V, E, W)$$

- Determine if there is a hamiltonian cycle & some length t .

G' - complete graph

TSP in G' of wt n .

Does there exist in PTAS $(1+\varepsilon)-\alpha_{\text{PR}}$ for metric TSP
No $n^{o(1)}$ time

Theorem: There can't be a PTAS $(220/219) - \alpha_{\text{PR}}$, unless $P = NP$.

Restrict the metric

$\overline{\mathcal{I}}$

Euclidean metric

A set of points in \mathbb{R}^2 , with euclidean distances

$$d(x, y) = \|x - y\|_2$$

Find the shortest route that covers all the points.

The Traveling Salesman Problem

$$\text{Cities} = \{1, 2, \dots, n\}$$

$C(n \times n)$ matrix \rightarrow Cost of traveling between pairs of cities

\downarrow
symmetric

$\underbrace{\quad}_{\text{If we view this as undirected complete graph}}$
then the problem is $\underbrace{\text{Hamiltonian cycle}}_{\text{problem.}}$

Approximation algorithms for the
TSP can be used to solve the Hamiltonian cycle problem

NP-complete

$$G_1 = (V, E)$$

$$C_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ n+2 & \text{otherwise} \end{cases}$$

If Hamiltonian cycle then
tour = n

$$\text{otherwise } \geq (n+2) + (n-1) = 2n+1$$

$\underbrace{\quad}_{\text{Input to TSP-algo}}$

we can detect
hamiltonian cycle $\leftarrow \begin{cases} 2\text{-apx can increase the cost to } 2n \\ \text{for hamiltonian cycle} \end{cases}$

\downarrow
Contradiction! (because HC is NP-complete)

Assumption: Restrict attention to metric space (metric TSP)

Algo(1): A spanning tree of a connected graph $G_1 = (V, E)$ is a minimal subset of edges $F \subseteq E$ such that each pair of nodes in G is connected by a path using edges only in F .

minimum spanning tree: Total edge cost minimized.

* Cost (optimal tour of TSP) \geq cost (MST)

Take this tour and remove one edge

(We will get a spanning tree whose cost $>$ cost (MST))

Algo(1) = nearest addition algorithm \rightarrow 2-apx algo

\downarrow

$F = \{(i_2, j_2), \dots, (i_n, j_n)\}$ \rightarrow edges obtained

$OPT > \sum_{l=2}^n c_{i_l j_l}$

\downarrow
Minimum spanning tree

Cost of the first
two nodes (i_2, j_2)

\downarrow
 $2c_{i_2 j_2}$ (traversed
two times)

j is inserted between (i, k)

marked

$$c_{ij} + \underbrace{c_{jk} - c_{ik}}_{\leq c_{ij}} \leq 2c_j$$

$$\text{cost (nearest-addition algo)} \leq 2 \sum_{l=2}^n c_{i_l j_l} \leq 2(OPT)$$

* Eulerian graph \rightarrow traversal of edges (each edge exactly once)

A graph is eulerian iff it is connected and each node has even degree

Algo(II) \rightarrow Double Tree Algorithm

MST compute \rightarrow replace each edge by two copies of itself

↓
resulting graph is Eulerian and has cost $\leq 2(\text{OPT})$

Eulerian Traversal \rightarrow sequence of edges (but vertices might repeat)

i_0, i_1, \dots, i_k remove all but the first occurrence of each city in this sequence.

↳ Tour of each city once

two consecutive cities (i_1, i_m)

we have removed i_{l+1}, \dots, i_{m-1}

By triangle inequality, cost is decreased,

↳ In total cost is at most the total cost of all the edges in the Eulerian graph
 $\leq 2(\text{OPT})$

double-tree = 2-apx algo.

Christofides Algorithm : MST Comput

↳ $O = \text{set of odd-degree vertices}$

For a tree, sum of degrees = $2 \times |E| = \text{even}$

↳ number of odd degree vertices = $|O| = \text{odd}$

$|O| = 2k$

→ perfect matching $(i_1, i_2), \dots, (i_{2k-1}, i_{2k})$

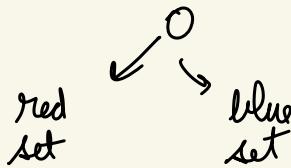
perfect-matching of minimum cost = $O(n^2)$

* Christofides = $\frac{3}{2}$ -apx algo

MST has cost $\leq \text{OPT}$

tour on nodes of $\underbrace{\mathcal{O}}$ has cost $\leq \text{OPT}$
↓
subset of original graph

Consider the shortest tour on the node set \mathcal{O} . Colour edges
red and blue



$$\text{cost(red)} + \text{cost(blue)} \leq \text{OPT}$$

$$\min(\text{cost(red)}, \text{cost(blue)}) \leq \frac{\text{OPT}}{2}$$

Perfect matching cost \leq Perfect matching + MST $\leq \frac{3}{2} \text{OPT}$

* For any constant $\alpha < \frac{220}{219}$ no α -apx for the metric TSP.

Euclidean TSP

Given n points in \mathbb{R}^2 with Euclidean distances i.e.

$$d(x_i, y) = \|x - y\|_2 \quad \text{shortest tour that visits all points?}$$

Euclidean TSP = NP-hard (do not know NP) length might be irrational

ϵ -nice instance

- (1) Every point has integral coordinates in the interval $[0, O(\frac{m}{\epsilon})]^2$
- (2) Any two different points have distances at least 4.

Consider the smallest bounding box around the points of the input instance. L = longer side of the box

$$L = \left\lceil \frac{8n}{\epsilon} \right\rceil$$

* OPT_I = length of the optimum tour in I . We can transform I into an ϵ -nice instance I' such that $OPT_{I'} \leq (1+\epsilon)OPT_I$

Proof: $I \rightarrow$ smallest bounding box \rightarrow length longer = L

optimal tour $\geq 2L$ \leftarrow $\begin{cases} \text{two points opposite in} \\ \text{the box have distance} \\ \geq L \end{cases}$

Now to obtain $I' \rightarrow$ draw a fine grid with spacing $\frac{\epsilon L}{2n}$ and map each point to the closest grid point.

$$L = \left\lceil \frac{8n}{\epsilon} \right\rceil = O\left(\frac{n}{\epsilon}\right) \quad \frac{\epsilon L}{2n} > \frac{\epsilon}{2n} \times \frac{8n}{\epsilon} = 4$$

$\Rightarrow I' = \epsilon$ -nice (integer coordinates and $d_{ij} > 4$)

By mapping points of I' to the points in I , we moved each point at most by $\frac{\varepsilon L}{2n}$

→ Edge changed by $\frac{\varepsilon L}{n}$ n edges $\rightarrow \varepsilon L$
 $\leq \text{opt}$

$$\text{OPT}_{I'} \leq \text{OPT}_I + \varepsilon L \leq (1+\varepsilon) \text{OPT}_I$$

VC dimension

Range space $S = (X, R)$

elements of $X \rightarrow$ points
elements of $R \rightarrow$ ranges

↓
ground set
(finite or infinite)

family of subsets of X
(finite or infinite)

$x =$ finite subset of X

measure of a range $\bar{m}(x) = \frac{|x \cap X|}{|x|}$

subset N (might be a multi-set)
of x

estimate of the measure $\bar{m}(x)$
is $\bar{s}(x) = \frac{|x \cap N|}{|N|}$

$Y \subseteq X$ $R_{1Y} = \{x \cap Y \mid x \in R\}$

projections of R on Y . The range space
S projected to Y is $S_{1Y} = (Y, R_{1Y})$

If R_{1Y} contains all subsets of Y ($\text{if } Y = \text{finite}, |R_{1Y}| = 2^{|Y|}$)

then Y is shattered by R

VC dimension ($\dim_{VC}(S)$) maximum cardinality of a
shattered subset of X .

Interval $\rightarrow VC = 2$

Disks $\rightarrow VC = 3$

Convex sets $\rightarrow VC = \infty$

Complement : range space $S = (X, R)$ $\dim_{VC}(S) = \bar{S}$

$$\bar{S} = (X, \bar{R})$$

$$\bar{R} = \{X \setminus \sigma \mid \sigma \in R\}$$

If S shatters B , then for any $Z \subseteq B$, $(B \setminus Z) \in R_{|B}$

$$Z = B \setminus (B \setminus Z) \in \bar{R}_{|B}$$

$\Rightarrow \bar{R}_{|B}$ contains all the subsets of B .

$\Rightarrow \bar{S}$ shatters $B \Rightarrow \dim_{VC}(\bar{S}) = \dim_{VC}(S)$

* Let $P = \{p_1, \dots, p_{d+2}\}$ be a set of $d+2$ points in \mathbb{R}^d . There are real numbers $\beta_1, \dots, \beta_{d+2}$ not all of them zero such that $\sum_i \beta_i p_i = 0$ and $\sum_i \beta_i = 0$

Proof: $q_i = (p_i, 1)$ $q_1, \dots, q_{d+2} \in \mathbb{R}^{d+1}$ are linearly dependent.

There are coefficients $\beta_1, \dots, \beta_{d+2}$ not all of them zero such that $\sum_{i=1}^{d+2} \beta_i q_i = 0$ considering only the first d -coordinates $\sum_{i=1}^{d+2} \beta_i p_i = 0$ $(d+1)^{th}$ coordinate $\sum_{i=1}^{d+2} \beta_i = 0$

Rado's Thm: $P = \{p_1, \dots, p_{d+2}\} \subset \mathbb{R}^d$ Then, there exist two disjoint subsets C and D of P , such that $CH(C) \cap CH(D) = \emptyset$

$$C \cup D = P$$

Proof: By previous thm, $\sum_i \beta_i p_i = 0$ and $\sum_i \beta_i = 0$

$$\mu = \sum_{i=1}^k \beta_i = - \sum_{i=k+1}^{d+2} \beta_i$$

$$\sum_{i=1}^k \beta_i p_i = - \sum_{i=k+1}^n \beta_i p_i$$

$v = \sum_{i=1}^n (\beta_i / \mu) p_i$ is a point in Convex Hull($p_1 \dots p_n$)

$$v = \sum_{i=k+1}^{d+2} -(\beta_i / \mu) p_i \in \text{CH}(p_{k+1}, \dots, p_{d+2})$$

v = intersection of the two convex hulls

* $P \subseteq \mathbb{R}^d$ = finite set s = point in $\text{CH}(P)$ h^+ = halfspace containing s . Then there exists a point of P contained inside h^+ .

Proof: $h^+ = \{t \in \mathbb{R}^d \mid \langle t, v \rangle \leq c\}$

$$\sum_i \alpha_i = 1 \quad \text{and} \quad \sum_i \alpha_i p_i = s$$

$$\langle s, v \rangle \leq c \Rightarrow \left\langle \sum_{i=1}^m \alpha_i p_i, v \right\rangle \leq c \Rightarrow \beta = \sum_{i=1}^m \alpha_i \langle p_i, v \rangle \leq c$$

$\beta_i = \langle p_i, v \rangle$ β is a weighted average of $\beta_1 \dots \beta_m$

\Rightarrow there must be a β_i which is no larger than the average $\Rightarrow \beta_i \leq c \Rightarrow \langle p_i, v \rangle \leq c \Rightarrow p_i \in h^+$.

* Growth Function = $G_S(n) = \sum_{i=0}^n \binom{n}{i} \leq \sum_{i=0}^n \frac{n^i}{i!} \leq n^S$

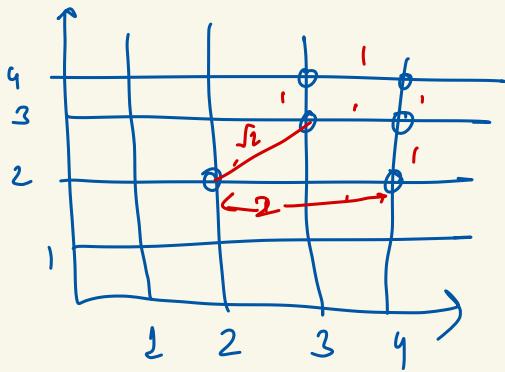
* Sauer's Lemma: If (X, R) is a range space of VC dimension S with $|X| = n$ then $|R| \leq G_S(n)$

Proof: holds for $n=0$ and $S=0$

$$R_x = \{\sigma \setminus \{x\} \mid \sigma \cup \{x\} \in R \text{ and } \sigma \setminus \{x\} \in R\}$$

$$R \setminus x = \{\sigma \setminus \{x\} \mid \sigma \in R\}$$

$$|R| = |R_x| + |R \setminus x| \leq G_{S-1}(n-1) + G_S(n-1) = G_S(n)$$



Euclidean TSP is NP-hard
but not known to be
NP

Sum of square roots (SRS)

Given a set of positive integers $\{a_1, \dots, a_k\}$ decide if $\sum_{i=1}^k a_i \leq t$

$a_1, a_2, \dots, a_k \in \mathbb{Z}^+$

$\sum_{i=1}^k a_i \leq t$

$\{b_1, \dots, b_k\}$

$\sum_{i=1}^k a_i \leq \sum_{i=1}^k b_i$

NP-hard
not in NP

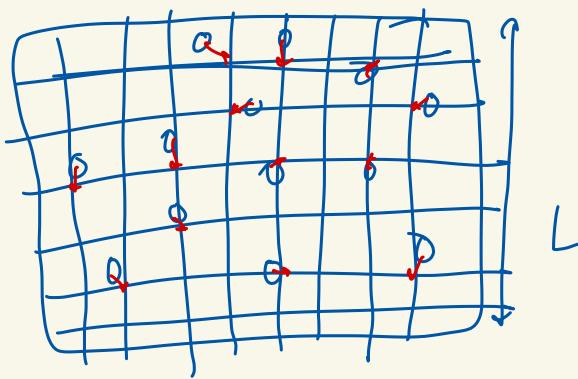
PTAS - polynomial time approximation scheme.

for Euclidean TSP

→ Rounding the instance

→ Partitioning (Exploit the structure of the instance by breaking it into more instances)

→ Dynamic programming



Map each point
to the closest
grid point
(To make the
coordinates
rational)

Given - E

ϵ - "nice" instance

Defⁿ - An instance of Euclidean TSP is ϵ -nice if

1. Every point has integral co-ordinates in the interval $[0, O(\frac{n}{\epsilon})^2]$
2. Any two diff points have dist at least 4.

- Take a small bounding box (axis-parallel)

longer side - L.

s.t. rooted not origin

$$\text{Scale } L = \sqrt{\frac{8n}{\epsilon}}$$

Lemma - I is slp & OPT_I is optimal tour.
I' is ϵ -nice instance $OPT_{I'}$ is optimal tour

$$OPT_{I'} \leq (1 + \epsilon) OPT_I$$

OPT is at least $2L$

- Draw a fine grid with spacing $\frac{\epsilon \times L}{2n}$
- Map every pt to its closest grid point (multiple pts could be mapped to the same grid pt).
- All pts have integer coordinates

$$L = \left\lceil \frac{8n}{\epsilon} \right\rceil \text{ or } O\left(\frac{n}{\epsilon}\right)$$

$$\text{Grid spacing } \frac{\epsilon L}{2n} \Rightarrow \frac{\epsilon L}{2n} \times \frac{8n}{\epsilon} = 4$$

→ Mapping each point in I has moved $\frac{\epsilon L}{2n}$

Every edge in the sol'^m changes by at most

$$\frac{\epsilon L}{2n} \text{ edges in OPT}$$

$$\text{Cost} = \epsilon \times L$$

$$\hookrightarrow \text{OPT}_I + \epsilon \times L$$

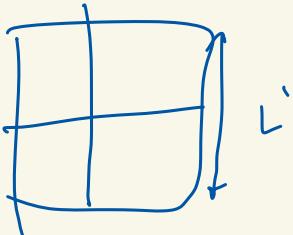
$$\text{OPT}_I + \epsilon \times \text{OPT}_I$$

$$\text{OPT}_{I'} \leq (1+\epsilon) \times \text{OPT}_I$$

$$L \leq \text{OPT}_I$$

Partition the space

- Extend the bonding box to square with new side length L'
 L' is the smallest power 2



* Recursively partition the box (square) into four equal sized squares until the side length is 1
 ↴ (L' is power of 2)

- each pt is separated
- one pt in each "non-empty" square

Partitioning terminate after $O(\log L')$ steps

Height of the quadtree

- $O(\log L')$
- $O(\lg(\frac{n}{\epsilon}))$

Apply dynamic programming to the quad tree

Solve for each square that are leaves

↓
Bottom-up combine

Portals

Limits the # of interactions

of portals
 Accuracy improves
 Running Time

Select $m = \text{power of } 2$

$$m \in \left[\frac{k}{\epsilon}, \frac{2k}{\epsilon} \right]$$

for each square \rightarrow put portals in corners
 put $(m-1)$ portals equally spaced

Portal-respecting tour (p-tour)

Defn: p-tour enters/exits through portals

$$-\text{length of p-tour} \leq (1+\epsilon) \times \text{OPT}$$

Detours can add much more cost

Sol^m → Randomize

(i) Translate the grid by a random offset at most

$$\frac{1}{2} \text{ in each coordinate}$$

(ii) points remain grid points.

(iii) with high probability, the pts are nicely concentrated.

(iv) higher levels in the partition (quad tree)
have more portals → fine-grained tree

Defn
(a,b) dissection : origin of the grid is translated by (-a,-b)

Theorem: (a,b) picked up uniformly at random $\left[0, \frac{1}{2}\right]$ with prob at least $\left(\frac{1}{2}\right)$

p-tour such that cost (p-tour) $\leq (1+4\epsilon) \times \text{OPT}$.

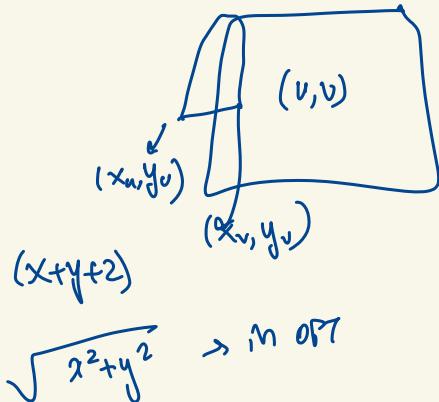
Extend non-ptour to a ptour

Proof: For each vertical/horizontal line l
 $+ (l) = \# \text{ of times intersects } l$

$$T_L = \sum_L t(L)$$

Claim : $T \leq 2 \times \text{opt}$

- e crosses $(x+1)$ vertical lines
- " " $(y+1)$ horizontal "
- total contribution $(x+y+2)$



$$\begin{aligned} \sqrt{2(a^2+b^2)} &\geq a+b \\ \forall x,y \quad d(x,y) &\geq 0 \\ x+y+2 &\leq \sqrt{2(x^2+y^2)} + 2 \\ &\leq 2 \underbrace{\sqrt{x^2+y^2}}_{\text{OPT}} \end{aligned}$$

Bound - expected length of the detours

Detours might occur

$$|x_u - x_v| + |y_u - y_v| + 2$$

i of the quad-tree

$$\frac{L'}{2^{i-m}}$$

if l is in level i

$$\leq \frac{L'}{2^{i-m}}$$

Q: what is the prob that after random shift l crosses
l at level-i

- l could be mapped to $\frac{l'}{2}$ many times [translated by $(0, \frac{l'}{2})$]

- 2^{i-1} many lines of level i :

$$\frac{2^{i-1}}{l'/2} = \frac{2^i}{l'}$$

Expected length $\sum_{i=1}^k \frac{2^i}{l'} \times \frac{l'}{2^i m} \leq \epsilon$

By linearity of expectation $2\epsilon \times \text{OPT}$

Markov Inequality

Pr (total length

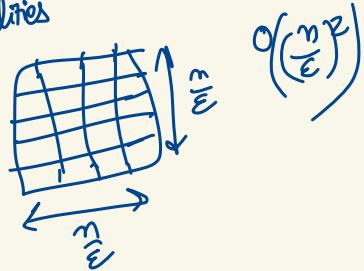
increase of detours) $> 4\epsilon \text{OPT}$)

$$\leq \frac{2\epsilon \text{OPT}}{4\epsilon \text{OPT}} = \frac{1}{2}$$

De-randomize

- fixed ϵ

- grid shifting by trying all possibilities



Final Step (DP)

Given (a, b) dissection, get p-tours

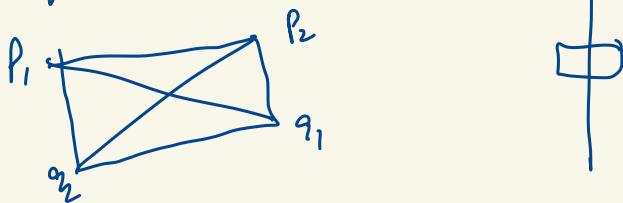
Introduce state

- Square

- any set of possible ways of entering/exiting the squares

$$\# \text{ of states} - (1 + 4 + 4^2 + \dots + L^2) = O\left(\frac{n^2}{\epsilon^2}\right)$$

Lemma: w.l.o.g. a portal is well-behaved 2-light



- 4 portals
- use one portal $m = O\left(\frac{n}{\epsilon}\right) = O\left(\log \frac{L}{\epsilon}\right)$

$$\text{Catalan number} = \frac{1}{2n+1} \binom{2n}{n} = O(2^{2n}) = O(2^{kn})$$

Algorithm = try all parenthesis

- translate them into paths

- Discard anything that intersects

$$m = O\left(\log \frac{n}{\epsilon}\right) \quad \# \text{ of entry exist} = O(n^{1/\epsilon})$$

Computation of values

A $\left[(s_1, t_1) \dots (s_\ell, t_\ell) \right]$ - Compute the whole table.

Clustering

- Learning, searching, data mining
- Given data, find an interesting structure
- Represented as points in \mathbb{R}^d

General metric space (X, d) where X is a set
 $d : X \times X \rightarrow [0, \infty)$

is a metric it satisfies -

- (i) $x=y \rightarrow d_\mu(x, y) = 0$
- (ii) $\forall x, y \quad d_\mu(x, y) = d_\mu(y, x)$
- (iii) $\forall x, y, z \quad d_\mu(x, y) + d_\mu(y, z) \geq d_\mu(x, z)$

Assumption

$(x, y) \quad d_\mu(x, y) \quad$ in $O(1)$ time

Norm

↳ norm defines distances between pts
 $p, q \in \mathbb{R}^d \quad \|p-q\|_p = \left(\sum_{i=1}^d |p_i - q_i|^p \right)^{\frac{1}{p}}$ for $p \geq 1$

$p=2$: Euclidean norm

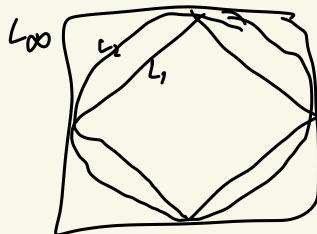
$p=1$: Manhattan distance (L_1 norm)

ℓ_∞ norm

$$\|p - q\|_\infty \leq \lim_{p \rightarrow \infty} \|p - q\|_p$$

max $|p_i - q_i|$

Triangle inequality holds for ℓ_∞ too, it's called Minkowski inequality



for any $p \in \mathbb{R}^d$

$$\|p\|_p \leq \|p\|_2 \quad \text{if } p \geq 0$$

Lemma — For any $p \in \mathbb{R}^d$

$$\|p\|_1 / \sqrt{d} \leq \|p\|_2$$

Proof: $p = (p_1, \dots, p_d)$ $p_i \geq 0 \ \forall i$

Const. α $f(x) = x^2 + (\alpha - x)^2$ minimized if $x = \frac{\alpha}{2}$

$$\text{Let } \alpha = \|p\|_1 = \sum_{i=1}^d |p_i|$$

By symmetry obs on $f(x) = \sum_{i=1}^d x_i^2$

$$\|p\|_2 \geq \sqrt{d(\frac{\alpha}{d})^2} = \|p\|_1 / \sqrt{d}$$

Metric space (X, d)

I/P : A set of points P , $|P|=n$

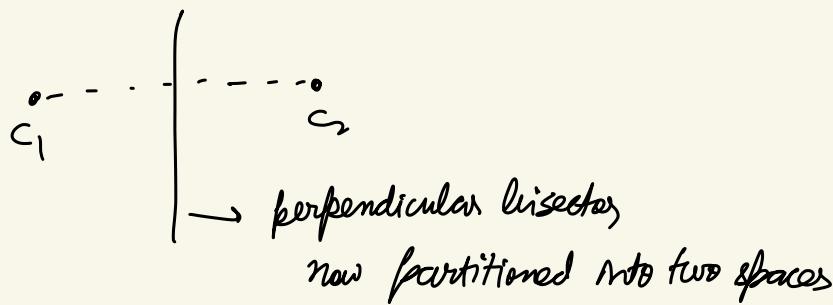
O/P : Find a clustering (set of centers) such that each pt is assigned to its nearest center

Set of clusters C

$$\text{Clusters}(C, \bar{c}) = \{p \in P \mid d_N(p, \bar{c}) = \underbrace{d_N(p, c)}_{\text{J}}$$

Voronoi Partition

minimum
across all the
pts



$$p_c = (d(p_1, c), d(p_2, c), \dots, d(p_n, c))$$

i^{th} coordinate $d(p_i, c)$ dist to p_i to its closest center

O/P = Find a set of k -centers $C \subseteq P$

such that the maximum distance of a point in P to its closest center is minimized

Defⁿ: Given a set of k -centers C ,
 $\|p_c\|_\infty = \max_{p \in P} d(p, C)$

Find C , s.t $\|p_c\|_\infty$ is minimized

$$\text{opt}_\infty(p, k) = \min_C \|p_c\|_\infty \quad C \subseteq P \quad k = |C|$$

- C_{opt}
- NP-hard
- Hard to approximate beyond 1.86
- 2-approx in the Euclidean space in \mathbb{R}^2

Greedy Algo

- Start by picking an arbitrary pt. \bar{c}_1

$$C_1 = \{\bar{c}_1\}$$

- Compute the distances for each $p \in P$ from \bar{c}_1

- Take the pt with worst distance

$$(x_1 = \max_{p \in P} d_1(p))$$

say \bar{c}_2

$$C_2 = C_1 \cup \{\bar{c}_2\}$$

$$C_i = C_{i-1} \cup \{\bar{c}_i\}$$

$O(nk)$
space

$\underbrace{O(nk)}_{\text{data from previous iterations}}$

For each pt $p \in P$, a single variable $d[p]$ with its current dist to the closest pt.

$$d[p] \leftarrow \min(d[p], d_N(p, \bar{c}_i))$$

Defⁿ: A ball of radius σ around a pt $p \in P$ is a set of pts in P with dist at most σ from p

$$b(p, \sigma) = \{q \in P \mid d_N(p, q) \leq \sigma\}$$

Remark: k -center is essentially covering P with k -balls of minimum radius.

Thm: Greedy Algo computes a set K of k -center such that k is 2 -apx $\|p_k\|_1 \leq 2 \|p_k\|_\infty$ takes $O(n \times k)$ time.

Proof: Running Time ✓

$$\text{Def}^n \quad r_k = \|p_k\|_\infty$$

Let \bar{c}_{k+1} is the point realising

$$r_k = \max_{p \in P} d(p, k)$$

$$C = K \cup \{\bar{c}_{k+1}\}$$

By the defⁿ of r :

$$r_1 \geq r_2 \geq \dots \geq r_k$$

$$i < j < k+1$$

$$d_N(\bar{c}_i, \bar{c}_j) \geq d_N(\bar{c}_i, \bar{c}_{i-1})$$

$$r_{i-1} \geq r_k$$

— the dist. between any pair of pts. in C is at least τ_k

opt — covers P by using k balls

by triangle inequality any two points within such a ball are with a dist at most $2 \times \text{opt}$.

↓

None of the balls contain two points from
Contradiction!

$C \subseteq P$

Greedy permutation

Let this run till we exhaust all pts

$$\langle P \rangle = \langle \bar{c}_1, \bar{c}_2, \dots, \bar{c}_n \rangle$$

↓

$$\langle \tau_1, \tau_2, \dots, \tau_n \rangle$$

Defⁿ : τ -packing : A set $S \subseteq P$ for P

(i) covering property : all the pts in P are within dist of atmost τ from S .

(ii) separation property : $\forall p, q \quad d_M(p, q) \geq \tau$

τ -packing gives compact representation

* Greedy permutation gives such a rep.

Thm: $\langle \overline{c_1}, \overline{c_2}, \dots, \overline{c_n} \rangle < \infty$
 for any i , we have $c_i = \langle \overline{c_1} \dots \overline{c_i} \rangle$
 is an ∞_i -packing of P

Proof: By contradiction

$$\infty_{k-1} = d(\overline{c_k}, \overline{c_{k-1}}) \forall k = 2, \dots, n$$

$$\text{for } j < k \leq i \leq n$$

$$d_\mu(\overline{c_j}, \overline{c_n}) = \infty_{k-1} \geq \infty_i$$

K-medians clustering

A set $P \subseteq X$ ($|P|=n$), a parameter k . Find a set of k -points $C \subseteq P$ s.t. the sum of distances of the pts in P to its closest center is minimized.

Clustering price: $\|P_C\| = \sum_{p \in P} d(p, C)$

Objective f^* : $\text{opt}_p(p, k) = \min_{\substack{C \subseteq P \\ |C|=k}} \|P_C\|$

Optimal set of centres - C_{opt} .

Local search: move sol^n to sol^n in the space of candidate sol^n (the search space) by applying local changes

Continue until, end up on optimal or we exhaust the running time.

Notations:

$$\text{A set } U = \{P_c \mid C \in P^k\}$$

$$\text{opt}_{\infty}(P, k) = \min_{\substack{q \in U \\ \text{k-center}}} \|q\|_{\infty} \quad \left| \begin{array}{l} \text{opt}(P, k) = \min_{q \in U} \|q\|_1, \\ \text{k-median} \end{array} \right.$$

1.86 Apx X

2 Apx ✓ (Greedy)

Claim: For any set P , $|P| = n$, k

$$\text{opt}_{\infty}(P, k) \leq \text{opt}_1(P, k) \leq n \times \text{opt}_{\infty}(P, k)$$

$$\begin{aligned} \text{Proof: } P &\in \mathbb{R}^m & \|P\|_{\infty} &= \max_{i=1}^n |P_i| \\ && \leq \sum_{i=1}^n \|P_i\|_1 &= \|P\|_1 \end{aligned}$$

$$\|P\|_1 \leq \sum_{i=1}^n |P_i| \leq \sum_{i=1}^n \max |P_i| \leq n \times \|P\|_{\infty}$$

C -set of centers $|C| = k$ realising $\text{opt}_1(P, k)$ i.e.

$$\text{opt}_c(P, k) = \|P_c\|_1$$

$$\begin{aligned} \text{opt}_{\infty}(P, k) &\leq \|P_c\|_{\infty} \\ &\leq \|P_c\|_1 = \text{opt}_c(P, k) \end{aligned}$$

Similarly, k realizing $\text{opt}_{\infty}(P, n)$

$$\begin{aligned} \text{opt}_k(P, k) &= \|P_k\|_1 \leq \|P_k\|_1 \\ &\leq n \times \|P_k\|_{\infty} \\ &= n \times \text{opt}_{\infty}(P, k) \end{aligned}$$

($2n$ -factor for median)

$2n$ -apx

use this as a first step for local search

L - is $2n$ -apx

Improve : parameter $0 < s < 1$
 $\forall i \in [n] L_{\text{curr}}$

Local search

- Set $L_{\text{curr}} \leftarrow L$
- At each iteration



We will check if the current "set" L_{curr} can be improved

by replacing one of the centers

by one center from outside (non-centers)



Swap

$$K \leftarrow (L_{\text{curr}} \setminus \{\bar{c}\}) \cup \{\bar{c}\}$$

if $\|P_k\|_1 \leq (1-s) \|P_{L_{\text{curr}}}\|_1$

- continue swap as long as it satisfies the constraint
- return L_{curr}

Running time : An iteration takes $O(m \times k)$ swaps

$(n-k)$ candidates to be swapped in k -candidates
to be out)

implementing swap (naively $O(n^k)$)
overall $O(n^{2k})$

Since

$$\frac{1}{1-s} \geq (1+s)$$

$$O((n^k)^2 \log \frac{1}{1-s} \frac{\|P_k\|_1}{\epsilon_{\text{pt}_1}})$$

$$= O(n^k)^2 \cdot \log(1+s)^{2n} = O((n^k)^2 \log \frac{n}{s})$$

K-means

Set $P \subseteq X$, K , find K pts $C \subseteq P$ $|C|=k$

$$\|P_C\|_2^2 = \sum_{p \in P} (d_{\mu_p}(p, C))^2$$

Obj : s.t. $\|P_C\|_2^2$ is minimized

$$\text{Opt}_2(P, k) = \min_{C, |C|=k} \|P_C\|_2^2$$

$O(n)$ -factor for k -means as well

Thm: $0 < \varepsilon < 1$

$(25 + \varepsilon)$ -approx

VC-dim

- A range space $(X, R) = S$

X = ground set (finite / infinite)

R = family of subsets of X .

Consider finite subset of X as the estimating ground set.

Dfⁿ (Measure): fixed subset of X . For a range $\tau \in R$

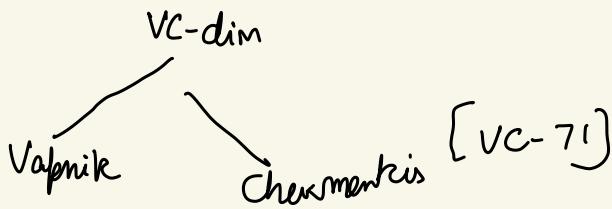
$$m(\tau) = \frac{|\tau \cap X|}{|X|}$$

For a subset N (multi-set) of X , the estimate of the measure of $m(\tau)$, for $\tau \in R$

$$\hat{s}(\tau) = \frac{|\tau \cap N|}{|N|}$$

Q2 How we get methods to generate N s.t.

$$\overline{S}(x) = \overline{m}(x) \quad \forall x \in \mathbb{R}$$



Dfm: $S = (X, R)$ For $Y \subseteq X$

$$R_{S,Y} = \{\tau \cap Y \mid \tau \in R\}$$

be the projection of R on Y

$\binom{|Y|}{2}$

If this is the cardinality then it is called
shattered by R

The orange
space S_Y
is projected
to $S_{Y'} = \{Y, R_{Y'}\}$

Complement

$$S = (X, R) \quad S = \dim_{VC}(S)$$

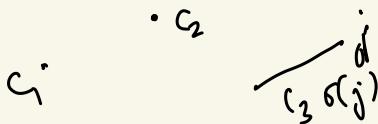
$$\overline{S} = (X, \overline{R}) \quad \text{where } \overline{R} = \{X \setminus \tau \mid \tau \in R\}$$

Q2 what is the VC-dim of \overline{S} ?

A subset $B \subseteq X$, is shattered in \overline{S} iff it is shattered in S
for any $Z \subseteq B$ $(B \setminus Z) \in \overline{R}_{IB} \Rightarrow Z = B \setminus (B \setminus Z) \in \overline{R}_{IB}$

Local search

- X be the set of arbitrary subset of k -centers
while true do:
(Swap) if i.e X and $i' \in F \setminus X$
 $\text{cost}(X - i + i') < \text{cost}(X)$
- (greedy solution of k -centers)



$\text{opt} = X^* - \text{optimal set of } X \leftarrow X - i + i'$
k-center otherwise break

Nearest

$$i^* \in X^*$$

$$i \notin X$$

for each center chosen in X ,
nearest center in X^* $\min_{i^*}(\|i\|_j)$

Inverse

Ties the centers in X^*

inverse is the
nearest map

Bijection

Claim: for any $j \in X$,
 $d_{\text{nearest}}(\sigma^*(i), j) \leq d_j + 2d_j^*$

Half-spaces

Let \mathcal{R} be the set of closed half spaces in \mathbb{R}^d

Claim: $P = \{P_1, \dots, P_{d+2}\}$ set of points in \mathbb{R}^d

Real numbers $\beta_1, \beta_2, \dots, \beta_{d+2}$ (not all are zero)

$$\text{s.t. } \sum_i \beta_i p_i = 0 \quad \& \quad \sum_i \beta_i = 0.$$

Proof: $a_i = (p_i, \beta_i)$ for $i=1 \dots d+2$

pts are linearly dependent . and these are
 $a_1, a_2, \dots, a_{d+2} \in \mathbb{R}^{d+2}$ coefficients $\beta_1, \dots, \beta_{d+2}$
 s.t. $\sum_{i=1}^{d+2} \beta_i p_i = 0$

- Considering first d -coordinates of these pts implies

$$\sum_{i=1}^{d+2} \beta_i p_i = 0$$

$$\text{Similarly } (d+1) \text{ coordinates } \sum_{i=1}^{d+2} \beta_i = 0$$

Radon's Thm: $P = \{P_1 \dots P_{d+2}\}$ \exists disjoint subsets
 $C \& D$ of P . $H(C) \cap H(D) = \emptyset$ then
 $C \cup D = P$

Shattering Dim:

Property : A range space (\mathcal{R}) with $VC\text{-dim}(S)$
 means # of ranges given polynomially on (n)
 (Generally this is \exp^n)

$$\underline{\text{Growth function}}: \quad G_{\delta}(n) = \sum_{i=0}^{\delta} \binom{n}{i} \leq \sum_{i=0}^{\delta} \frac{n^i}{i!} \leq n^{\delta} \quad \text{for } \delta > 1$$

$$\underline{\text{Sauer's Lemma}}: \quad S = (X, R) \\ \text{VC}(S) = \delta \quad |X| = n \quad |R| \leq G_{\delta}(n)$$

Proof: $n=0 \quad \delta=0 \quad \rightarrow \text{done!}$

$$x \in X$$

$$\text{contains}_x \{R_x\} = \left\{ \tau \setminus \{x\} \mid \tau \cup \{x\} \in R \text{ and } \tau \supset \{x\} \in R \right\}$$

$$\text{does not contain}_x \{R \setminus x\} = \left\{ \tau \setminus \{x\} \mid \tau \in R \right\}$$

$$\text{Observation: } |R_x| + |R \setminus x| = |R|$$

Shatter function: $S = (X, R)$ shatter f^n
 $\pi_S(m)$ is the maximum # of sets that might be created by S , when restricted to the subsets of size m .

$$\pi_S(m) = \max_{\substack{B \subseteq X \\ |B|=m}} |R_{|B|}|$$

Shattering dim: The smallest d such that $\pi_S(m) = O(m^d)$ $\forall m$

Then $S = (X, R)$ has shattering dim d , then the VC-dim is bounded by $O(d \log d)$

Proof: $N \subseteq X$ be the largest subset of X shattered by S and s is the cardinality

$$2^s = |R_{|N|}| \leq \pi_S(m)$$

$$s \leq \log c + d \log s \quad (s \geq \max(2, \frac{2}{c}))$$

$$\Rightarrow \frac{s \cdot \log c}{\log s} \leq d$$

$$\frac{s}{2 \log s} \leq d \Rightarrow \frac{s}{\log s} \leq O(1) \times d$$

$$f(x) = \frac{x}{\log x} \rightarrow \text{non-increasing } x > c$$

$c > \sqrt{e}$ if $f(x) \geq e$ $\Leftrightarrow x > 1$
 $f(x) \leq x$ then $x \leq \log x$

ε -net and ε -sampling

$S = (X, R)$ x is a finite subset of X
 $0 \leq \varepsilon \leq 1$

Informally, ε -sampling captures R , upto some ε -error

a subset $c \subseteq x$ is an ε -sample for x if for any range $r \in R$

$$|\bar{m}(r) - \xi(r)| \leq \varepsilon$$

\downarrow measure \curvearrowright estimate

Thm: (ε -sample, VCT₁) — There is a free constant C s.t. if (X, R) is any range space with VC dim S .

$x \subseteq X$ finite subset of X and $\forall \varepsilon, \phi > 0$
 \exists a random subset $C \subseteq X$ of

(with probability $= \phi$) cardinality $S = \frac{C}{\varepsilon^2} \left(\delta \log \frac{\delta}{\varepsilon} + \log \frac{1}{\phi} \right)$

ε -net : A set $N \subseteq X$ is an ε -net for x if for any range space $r \in R$ if $\bar{m}(r) \geq \varepsilon$
 then r contains at least one pt. of N (i.e. $r \cap N \neq \emptyset$)

(Intuitively: hit all heavy subsets)

ϵ -net Thm (HWF7)

$S = (x, R)$ has $\text{VC dim}(S) \leq S$

x is a finite subset of X .

Suppose $0 \leq \epsilon \leq 1$ & $\phi < 1$

- N a set obtained by random independent draws.
- $m \geq \max\left(\frac{4}{\epsilon} \log \frac{4}{\phi}, \frac{8S}{\epsilon} \log \frac{16}{\epsilon}\right)$
- Then, N is a ϵ -net with prob $(1-\phi)$.

* Remark: Both of the thms hold for spaces with shattering dim S . ($O\left(\frac{1}{\epsilon} \log \frac{1}{\phi} + \frac{S}{\epsilon} \log \frac{S}{\epsilon}\right)$)

Range Searching $p \in \mathbb{R}^d$ we have a database
Given a hyper rectangle, we want to report the points that lie inside

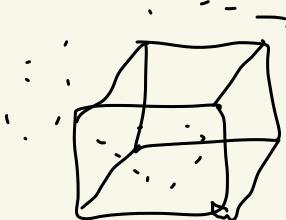
Allow 1% error,

ϵ -sample (Thm) says there is
a subset of const. size (which depends on ϵ)

Use this to perform an estimation.

Rectangle has bounded VC-dim

Random sample with probability $(1-\phi)$



Learning Concepts

Assume we know a f^* that returns 1 if inside,
0 otherwise

Query
Oracle

There is a distribution D defined over the space. We pick points from D .

Growth function $G_d(n) = \sum_{i=0}^d \binom{n}{i} \leq \sum_{i=0}^d \frac{n^i}{i!} \leq n^d$

Sauer's Lemma $S = (X, R)$

Suppose $\text{VC dim}(S) \leq d$

$$|R| \leq G_d(n) \leq \sum_{i=0}^d \binom{n}{i}$$

$T_F(n)$

Proof: By induction on n, d
 $n=d=1$ holds

Assume that it holds for $n-1 \& d$
and as well as for $n-1 \& d-1$

We prove for $n \& d$
define $f^* : \sum_{i=0}^d \binom{n}{i} = h(n, d)$

Our induction hypothesis is for F with $\text{VC-dim} \leq d$

$$T_F(n) \leq d$$

$$\binom{n}{d} = \binom{n-1}{d} + \binom{n-1}{d-1}$$

$h(n, d) = h(n-1, d) + h(n-1, d-1)$

recurrence

Now let's fix a class F

$$VC\text{-dim}(F) = d \quad \text{and a set}$$

$$X_1 = \{x_1, \dots, x_m\} \subseteq X$$

$$f_1 = f_{1X}$$

$$f_2 = f_{2X}$$

$$F_3 = \{f_{1X} | f \in F \text{ & } f' \in F \text{ s.t. } \forall x \in X_2, f'(x) = f(x) \text{ & } f'(x_1) = -f(x_1)\}$$

$$VC\text{-dim}(F')$$

$$\leq VC\text{-dim}(F) \leq d$$

$$|F_1| = |F_2| + |F_3| \leq d \leq d-1$$

Induction hypothesis

$$\begin{cases} |F_2| \leq h(n-1, d) \\ |F_3| \leq h(n-1, d-1) \end{cases} \rightarrow |F_1| \leq h(n-1, d) + h(n-1, d-1) \leq h(n, d)$$

Ex. Let F be s.t. $VC\text{-dim}(F) \leq d$ for $n \geq d$

$$\pi_F(n) \leq \left(\frac{mc}{d}\right)^d$$

Set-cover / Hitting set (piercing)

U = universe of elements

X = set of subsets

$S = (U, X)$ - set system

choose a subset $X' \subseteq X$
which is a cover

- NP hard

Greedy approximation - (1) sort all the sets based on cardinality
(2) choose the set with max cardinality.

log factor

↳ \exists a lower bound shows that we can't get better than log factor.

Wish: for "nice" set families, can we beat greedy (log factor)?

$S = (X, R)$

↓
set of elements → set of ranges

Goal: choose a subset $R' \subseteq R$ that covers X .

class of objects \rightarrow has bounded VC-dim

ε -net: Sample that "wits" all the heavy sets ($> \varepsilon_n$)

A set $N \subseteq X$ is an ε -net for a finite subset x if
for any range $r \in R$, $m(r) = \frac{m(x)}{|x|} > \varepsilon$

then r contains at least one pt.

Construction of ε -net

- choose a random sample if it is large enough its ε -net
- small hitting set " (which is also a net)

ε -net thm

We can get a subset N by m ind. draws for a finite subset x .
(uniformly chosen)

$$N \geq \max \left\{ \frac{9}{\varepsilon} \log \frac{a}{\phi}, \frac{8\delta}{\varepsilon} \log \frac{16}{\varepsilon} \right\}$$

with prob $> 1 - \phi$.

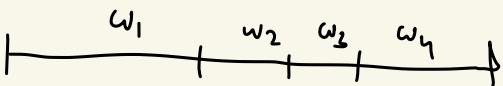
Suppose the shattering dim is d .

$$\text{sample size} \geq O\left(\frac{d}{\varepsilon} \log \frac{d}{\varepsilon}\right)$$

Weighted net: suppose the elements are wt,
($W: x \rightarrow \mathbb{R}^+$)

r -subset
 $w(r) = \sum_{j \in r} w(j)$

Goal : all the r 's with wt. $> \epsilon x_w$



Algorithm for set cover

(1) Repeatedly select an ϵ -net (for some ϵ)

$$S = (X, R) \text{ dual} - S^* = (X^*, R^*)$$

\downarrow
shattering dim S^*

$$\text{size of the net} = O\left(\frac{\xi^*}{\epsilon} \log \frac{\xi^*}{\epsilon}\right)$$

Verify if it is a net. If not - discard.

\downarrow
if it is a net check if it is a setcover
if yes - done

Let $R_p = \{\infty \in R \mid p \in \infty\}$ all ranges that contain p .

Double the weight of the elements in R_p

Observation: every time we double we are increasing not more than $(1+\epsilon)$ multiplicative function

$$w_i \leq (1+\epsilon)^i w_0$$

$i \rightarrow$ iteration

Q1 What is the min wt. of elements ($K = \text{optt}$) in opt?

$$K \times 2^{\frac{i}{k}} \leq w_i = (1+\varepsilon)^i w_0 = (1+\varepsilon)^i x_m$$

$$\leq \varepsilon^{\frac{i}{k}} x_m$$

$$K \times 2^{\frac{i}{k}} \leq w_i$$

$$i = k \times g$$

$$k \times 2^g \leq e^{\varepsilon i} x_m$$

$$\log(k + g) \leq \log m + \sum i$$

$$g(1-\varepsilon k) \leq \log m - \log k = \log\left(\frac{m}{k}\right)$$

Suppose, we take $\varepsilon = \frac{1}{2k}$

$$\Rightarrow g \leq O\left(\log \frac{m}{k}\right)$$

$$\# \text{ of iteration } O\left(k \log \frac{m}{k}\right)$$

Size of our cover $O(S^*k \log S^*k)$ $k = \text{opt. cover}$

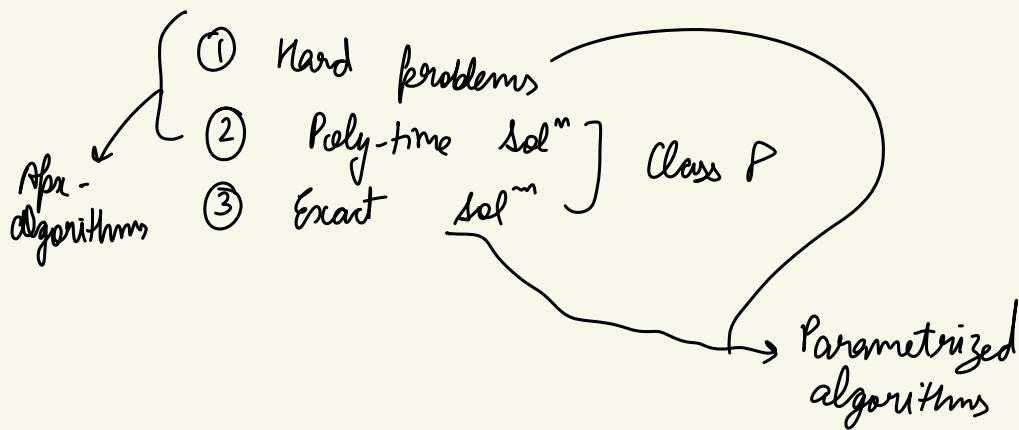
Q2 How to choose ε ?

ε is independent on k

Suppose $\varepsilon = \frac{1}{uk}$ instead of $\frac{1}{2k}$

Guess the value of $\frac{\varepsilon}{k_i}$

$O(k_i \log \frac{m}{k_i})$ iterations



Idea: Aim is to get exact algo

But we want to isolate \exp^n terms (parameters)

\Rightarrow obtain very fast solⁿ when the parameter small.

(Note: parameters are small in practice).

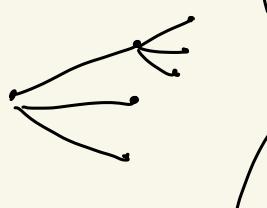
parameters - non-negative integer $k(x)$ (comes with prob i/p)
 - denote by k
 - Not necessarily efficiently computable.

Parametrized Problem

problem + parameter (k)
 (w.r.t. k)

Goal: poly. complexity on n
 Expⁿ complexity on k

Example:



I/p : $G = (V, E)$ $k \in \mathbb{N}$

o/p : Does there exist a
 k -size vertex cover

↓
output a set ($\subseteq V$) s.t. $\forall e \in E$
 $\exists v \in S$

Brute force solution

(1) Try all $\binom{n}{k} + \binom{n}{k-1} + \dots + \binom{n}{0}$
 ↓
 All sets of k vertices

— Test valid VC takes $O(E)$ time

— Total = $O(V^k E)$ kely for fixed k .

slow for large n and
reasonable k .

Branching (Bounded search tree technique)

→ Pick an arbitrary edge $e \in E$
 ↓
 (v, v')

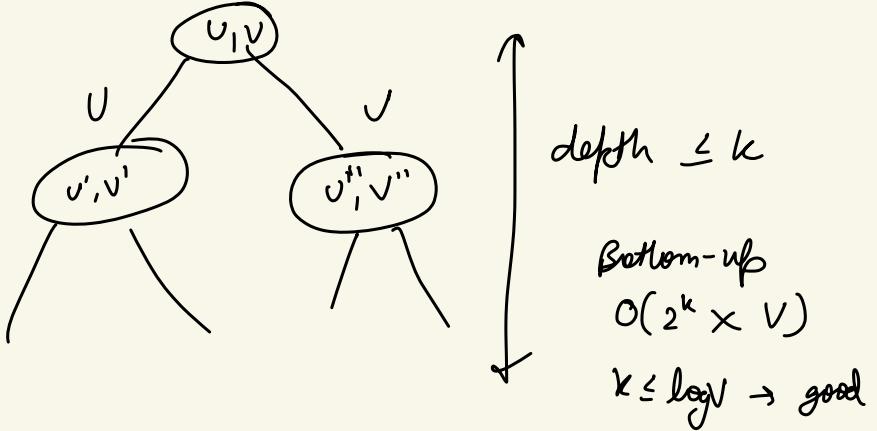
→ Know either $v \in S$ or $v' \in S$ or $\{v, v'\} \in S$

Guess — Try both

① Add v to S
 (delete v & $N(v)$ from S)
 recursive $k' = k - 1$

② same for v' .

— Return or of the outcomes



Fixed parameter-tractable (FPT)

If \exists an algo with running time $\leq f(k) \times n^{O(1)}$

$f: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ polynomial
 parameters

Q: why $f(k) \times n^{O(1)}$ and not $f(k) + n^{O(1)}$?

Thm: $f(k) \times n^c \Leftrightarrow f(k) + n^{c'}$

Proof: \Rightarrow if $n \leq f(k)$

$$f(k) \times n^c \leq f(k)^{c+1}$$

if $f(k) \leq n$

$$f(k) \times n^c \leq n^{c+1}$$

$$\text{So. } f(k) \times n^c \leq \max \left\{ f(k)^{c+1}, n^{c+1} \right\} \leq f(k)^{c+1} + n^{c+1}$$

(\Leftarrow Trivial, assuming $f(k) & n^{c'} \geq 1$)

Kernelization

simplifying
self-reduction

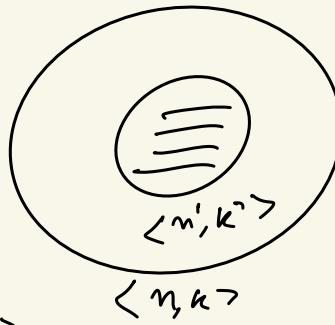
(poly-time reduction)

i/p - $\langle n, k \rangle$

converts it into $\langle n', k' \rangle$

How small? $|n'| \leq f(k)$

Equivalent — Ans($\langle n, k \rangle$) = Ans($\langle n', k' \rangle$)



Thm:

FPT \Leftrightarrow kernelization

kernelization $\Rightarrow n' \leq f(k)$

run any finite $g(n')$

$\Rightarrow n^{O(1)} + g(f(k))$ time \rightarrow FPT

\Leftrightarrow

A runs in $f(k) > n^C$

if $n \leq f(k)$ in kernelized

if $f(k) \leq n$

run A $\rightarrow f(k) \times n^C \leq n^{C+1}$

O/p, O(1) size

k is known in advance

Sunflower Lemma (Erdős Rado Cons)

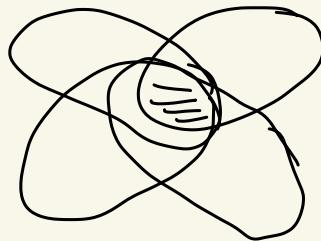
- Classical result from 1960

- Apply in kernelization

- k petals

- A core γ

(\Rightarrow None of them can be empty)



Collection of sets S_1, \dots, S_k

s.t. $S_i \cap S_j = \gamma$

$\forall i \neq j$

Petals $\forall i \in S \setminus \gamma$

Lemma F - family of sets (no duplication) over a universe U

s.t. each set has cardinality exactly d .

- If $|F| > d! (k-1)^d$, then F contains k petals

- Poly-time algorithm to compute this

w.r.t. $|F|, |U|, k$

Proof: for $d=1$, singletons

Suppose $d \geq 2$

Let $G = \{S_1, S_2, \dots, S_l\} \subseteq F$

- If $l \geq k$ then G is a sunflower

already with at least k petals

Assume $l < k$

be inclusion-wise
maximal family of
pairwise disjoint sets
in F

$$S = \bigcup_{i=1}^k S_i; \quad \text{Then } |S| \geq d \times (k-1)$$

Since G is maximal, every set $A \subseteq F$ intersects at least one set from G . $A \cap S \neq \emptyset$.

There is an element $v \in V$ which is contained in at least

$$\frac{|F|}{|S|} \text{ sets.} \quad \frac{|F|}{|S|} > \frac{d! (k-1)^d}{d(k-1)} = \underbrace{(d-1)!}_{\text{sets for } F} (k-1)^{d-1}$$

- Take all sets of F containing this element v .

Construct F' of sets union cardinality $(d-1)$ by removing v .

$$|F'| > d!. (k-1)^d$$

By induction hypothesis

F' contains a sunflower $\{S'_1 \dots S'_n\}$ with k -petals

$$\{S'_1 \cup v\} \dots \{S'_n \cup v\}$$

Poly-Time Algorithm

(1) greedily select maximal sets. If size is at least k done. Else find v and return.

Erdos-Rado - FIGO

Each set in F has cardinality d

If $|F| > d!(k-1)^d$ then there is a sunflower.

$(\log k)^d \rightarrow$ recent bound.

d -Hitting set

(Application of Sunflower Lemma)

Input: Family of sets A over V . each set has cardinality at most d .

a non-negative integer k

Output: whether there is a subset $H \subseteq V$ of size at most k , such that H contains 1 element of each of sets of A .

Proof: If A contains a sunflower, say $S = \{S_1, \dots, S_{k+1}\}$ of $\#(k+1)$ then every hitting set H of A of cardinality at most k intersects its core Y .

Reduction rule : (V, A, k)

Return (V', A', k') $A' = (A \setminus S) \cup \{x\}$
and $V' = \bigcup_{x \in A'} X$

If # of sets are larger than $d! \times k^d$ find a sunflower
— Apply green consider kernel size $O(d! k^d)$

Kernelization

A data reduction rule for a parametrized problem \mathcal{Q} is a function $\phi : \Sigma^* \times \mathbb{N} \rightarrow \Sigma^* \times \mathbb{N}$ that maps an instance (I, k) of \mathcal{Q} to an equivalent instance (I', k') of \mathcal{Q} such that ϕ is computable in time polynomial in $|I|$ and k .

$$\text{size}_A(k) = \sup \left\{ |I'| + k' : (I', k') = A(I, k), I \in \Sigma^* \right\}$$

* A kernelization algorithm for a parametrized problem \mathcal{Q} is an algorithm A that, given an instance (I, k) of \mathcal{Q} works in polynomial time and returns an equivalent instance (I', k') of \mathcal{Q} . Moreover, we require that $\text{size}_A(k) \leq g(k)$ for some computable function $g : \mathbb{N} \rightarrow \mathbb{N}$.

* If a parametrized problem \mathcal{Q} is FPT then it admits a kernelization algorithm

Proof: $\mathcal{Q} = \text{FPT} \Rightarrow \exists A (I, k) \in \mathcal{Q}$ in time $f(k)|I|^c$
 (I, k) algo runs A on (I, k) for at most $|I|^{c+1}$ steps
 If it terminates with an answer, use that for yes/no.
 If A does not terminate within $|I|^{c+1}$ steps, then return (I, k) itself

$$f(k) \cdot |I|^c > |I|^{c+1} \Rightarrow |I| < f(k)$$

$$|I| + k \leq \underbrace{f(k) + k}_{\text{computable}} \quad (\text{kernel size})$$

Sunflower Lemma

A sunflower with k petals and a core γ is a collection of sets S_1, \dots, S_k such that $S_i \cap S_j = \gamma$ for all $i \neq j$; the sets $S_i \setminus \gamma$ are petals and we require none of them to be empty (γ can be empty).

* Let A be a family of sets (without duplicates) over a universe U , such that each set in A has cardinality exactly d . If $|A| > d!(k-1)^d$, then A contains a sunflower with k petals and such a sunflower can be computed in time polynomial in $|A|, |U|$ and k .

For $d=1$, family of singletons, statement holds
 $d \geq 2$ $A = \text{family of sets of cardinality at most } d \text{ over a universe } U \text{ such that } |A| > d!(k-1)^d$.
 $G = \{S_1, \dots, S_l\} \subseteq A$ be an inclusion-wise maximal family of pairwise disjoint sets in A .
If $l \geq k$ then G is a sunflower with at least k petals.

G is maximal, every set $A \in A$ intersects at least one set from G i.e. $A \cap S \neq \emptyset$.

$$S = \bigcup_{i=1}^l S_i \quad |S| \leq d(k-1)$$

There is an element $v \in V$ contained in at least

$$\frac{|A|}{|S|} \geq \frac{d! (k-1)^d}{d(k-1)} = (d-1)(k-1)^{d-1}$$

sets from A . We take all sets of A containing such an element v , and construct a family A' of sets of cardinality $d-1$ by removing from each set the element v . Because $|A'| \geq (d-1)! (k-1)^{d-1}$, by the induction hypothesis A' contains a sunflower $\{S'_1, \dots, S'_r\}$ with k -petals. Then $\{S'_1 \cup \{v\}, \dots, S'_k \cup \{v\}\}$ is a sunflower with k -petals.

d -hitting set

Given a family A of sets over a universe U , where each set in the family has cardinality at most d , and a positive integer k . The objective is to decide whether there is a subset $H \subseteq U$ of size at most k .

such that H contains at least one element from each set in A .

* d -Hitting sets admits a kernel with at most $d!k^d$ sets and at most $d!k^d \cdot d^2$ elements.

Let (U, A, k) be an instance of d -hitting set and assume that A contains a sunflower $S = \{S_1, \dots, S_{k+1}\}$ of cardinality $k+1$ with core y . Then return (U', A', k') where $A' = (A \setminus S) \cup \{y\}$ is obtained from A by deleting all sets $\{S_1, \dots, S_{k+1}\}$ and by adding a new set y and $U' = \bigcup_{X \in A'} X$.

Additional Notes

* Voronoi Partitions: set of centers C , every point of P assigned to nearest neighbour in C

$$\Pi(C, \bar{C}) = \{ p \in P \mid d(p, \bar{C}) \leq d(p, c) \}$$

* Greedy clustering algorithm: arbitrary point \bar{c}_1 into C , for every point $p \in P$ compute $d_{\bar{c}_1}(p)$ from \bar{c}_1 . Pick point \bar{c}_2 with highest distance from \bar{c}_1 . Add this to the set of centers and denote this expanded set of centers as C_2 .

overall algorithm = $O(nk)$

→ This algorithm is 2-approx.

Proof: Case-1 Every cluster of C_{opt} contains exactly one point of k .

$$p \in P$$

$\bar{c} =$ center p belongs in C_{opt}

$\bar{k} =$ center of k that is in $\Pi(C_{opt}, \bar{c})$

$$d(p, \bar{c}) = d(p, C_{opt}) \leq r_{\infty}^{opt}(p, k)$$

$$d(\bar{k}, \bar{c}) = d(\bar{k}, C_{opt}) \leq r_{\infty}^{opt}$$

$$d(p, \bar{k}) \leq d(p, \bar{c}) + d(\bar{c}, \bar{k}) \leq 2r_{\infty}^{opt}$$

Case-2: Two centers \bar{k} and \bar{v} of k both in $\Pi(C_{opt}, \bar{c})$

σ was added later

$$r_{\infty}^k(p) \leq r_{\infty}^{C_{i-1}}(p) = d(\bar{v}, C_{i-1})$$

$$\leq d(\bar{v}, \bar{k})$$

$$\leq d(\bar{v}, \bar{c}) + d(\bar{c}, \bar{k}) \leq 2r_{\infty}^{opt}$$

* A set $S \subseteq P$ is a σ -net for P if the following two properties hold :
 (i) Covering property = All the points of P are in distance at most σ from the points of S .
 (ii) Separation property = for any pair of points $p, q \in S$
 $d(p, q) \geq \sigma$.

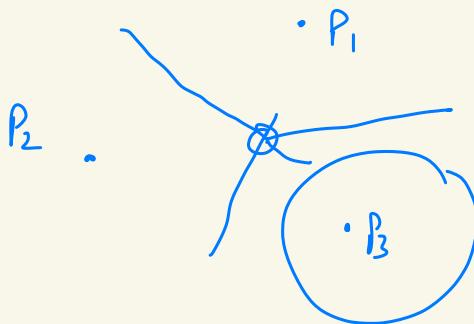
* Let P be a set of n -points in a finite metric space, and let its greedy permutation be $\langle \bar{c}_1, \dots, \bar{c}_n \rangle$ with the associated sequence of radii $\langle \bar{\sigma}_1, \dots, \bar{\sigma}_n \rangle$ for any i , $C_i = \langle \bar{c}_1, \dots, \bar{c}_i \rangle$ is a σ_i -net of P .

* $0 < p < 2 \quad \|x\|_p \geq \|x\|_2$

$$\|x\|_p \leq \sqrt{n} \|x\|_2 \quad \text{and} \quad \|x\|_2$$

Nearest Neighbour Search

- Let P be a set of n points in \mathbb{R}^d . Pre-process the pts in P .
- s.t. given a query pt. q , we can determine efficiently the closest pt to q in P .



worst possible running time $O(n^2)$
 Voronoi diagram
 $O(n \log n)$
 (through a complex algo)

- Voronoi Diagram $O(n \log n)$
- Apply point location query roughly takes $n^{[d/2]}$ time.

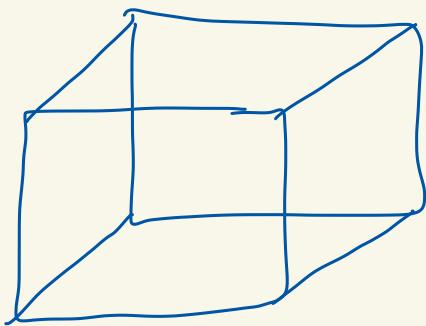
Approximate NN

- Specify a parameter $\epsilon > 0$

Build a data structure that ans. $(1+\epsilon)$ -Apx NN (ANN)

Input: P in \mathbb{R}^d ; for a query pt. q $nn(q) = \underbrace{nn(q, P)}$

$d(q, P)$ = distance of q to the closest point in P $\|q - s\| \leq (1 + \epsilon) d(q, P) \rightarrow$ Find s .



$\text{Spread} = L$ is bounded
Unit hypercube in \mathbb{R}^d

T is a quadtree of the space

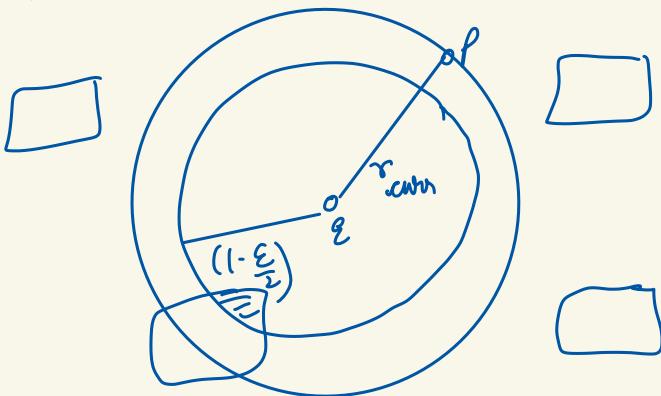
$$\text{diam}(P) = \sqrt{1}$$

T

of bits of P stored
that rooted at v .

Assume for each (internal) node m in
For representative (rep m) is one

- Idea:
1. Maintain a set of nodes of T that might contain ANN of q .
 2. Each node has a representative (compute the distance to the query q)
 3. At each (i), - Θ search gets refined by replacing a node by its children.



$$||q - \text{rep}_m|| - \text{diam}(\square_m) > (1 - \frac{\epsilon}{2}) * r_{curr} \Rightarrow \text{Abort}$$

If not then keep doing

Alg:

Let $A_0 = \{\text{root}(z)\}$ $r_{\text{curr}} = \|q - \text{rep}(\text{root}(z))\|$

- In the i^{th} iteration

for $i > 0$ Alg expands the nodes of A_{i-1}

to get to A_i

(if cond. not satisfied)

add w to A_i

Continue until all elements of A_{i-1} are considered

stop when A_i is empty.

Correctness:

Alg adds a node w to A_i

only if P_w misnot contain pts which are close to \mathcal{E}
then current best.

- say w is the last node
inspected by P Alg s.t. $\text{nn}(q) \in P_w$

if it throws away :

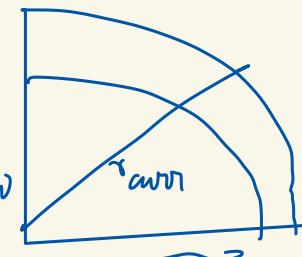
$$\|q - \text{nn}(q)\| \geq \|q - \text{rep}_w\| - \underbrace{\|\text{rep}_w - \text{nn}(q)\|}_{\geq \dfrac{1-\epsilon}{2} r_{\text{curr}}}$$

$$\geq \|q - \text{rep}_w\| - \underbrace{\text{diam}_w(\Delta_w)}_{\geq \left(1 - \frac{\epsilon}{2}\right) r_{\text{curr}}}$$

$$\text{so, } \|q - \text{nn}(q)\| / \left(1 - \frac{\epsilon}{2}\right) \geq r_{\text{curr}}$$

$$\left(1 - \frac{\epsilon}{2}\right) \leq (1 + \epsilon) \quad r_{\text{curr}} \leq (1 + \epsilon) \times d(q, P)$$

$$\geq \left(1 - \frac{\epsilon}{2}\right) r_{\text{curr}}$$



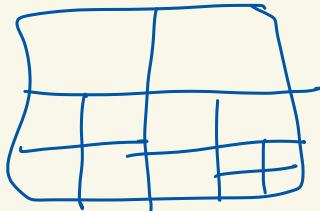
Running Time : Algorithm visits \mathcal{T} level by level

As long as level's grid cells are bigger than ANN
(# nodes visited so far)

$$\text{running time} = O\left(\varepsilon^{-1} + \log\left(\frac{1}{w}\right)\right)$$

↓
prob. to spread.

Compressed Quadtree



Alg handled nodes level by level
(keep a heap of integers in the range of
 $O(\log \phi(P))$, ..., $\log(\phi(P))$)

$$O(\varepsilon^{-d} + \log\left(\frac{\text{diam}(\mathcal{T})}{w}\right))$$

General unbounded case:

- Get rough abx
- Apply previous arguments

Embedding Finite Metric Spaces into Normed Spaces (Metric Embedding)

Recap: Metric space is a pair (X, d) , X is a set
 $d: X \times X \rightarrow [0, \infty)$

is a metric if the following happens:

$$(I) \quad d(x, y) = 0 \quad \text{if } x=y$$

$$(II) \quad d(x, y) = d(y, x)$$

$$(III) \quad \forall x, y \quad d(x, y) + d(y, z) \geq d(x, z) \quad (\text{Triangle inequality})$$

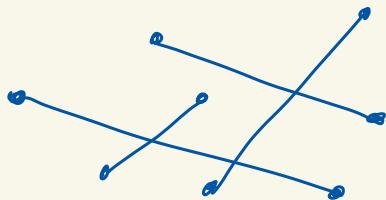
A metric on m points ($m \times n$) matrix $\binom{m}{2}$

Applications

Microbiology — X is a collection of bacterial strains.

"Nice" Representation

P in \mathbb{R}^2



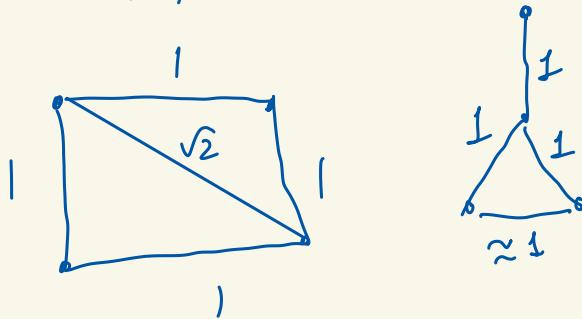
assign $x \in X$ to a point $f(x)$ in \mathbb{R}^2 s.t.,
 $f(x,y) \approx \|f(x) - f(y)\|_2$

Identify clusters, isolated entries and so on.

Geometry helps to get faster algo.

Too good to be true.

$X = \{P_1, P_2, P_3, P_4\} \rightarrow$ isometric space



Isometric Embedding

A mapping $f: X \rightarrow Y$

$X \rightarrow$ metric space with a metric P
 $Y \rightarrow$ metric space with a metric σ .

* is isometric if it preserves distances

$$\sigma(f(x), f(y)) = p(x, y) \quad \forall x, y \in X$$

Typically small "errors" (distribution) in embedding is ok

D embedding -

A mapping $f: X \rightarrow Y$ is called a D-embedding for

$$D > 1$$

real numbers

$$\text{if } \exists \tau > 0$$

$$\text{s.t. } \forall x, y \in X$$

$$\tau \times p(x, y) \leq \sigma(f(x), f(y))$$

$$\leq D \times \tau \times \sigma(x, y)$$

The infimum of number D s.t. f is a D-embedding - distortion

If X is a Euclidean space (in a normed space generally).

We can scale up/down.

Mapping with bounded D, were called bi-Lipschitz

Distortion can be defined in terms of Lipschitz constants of f and inverse of f (f^{-1})

$$\|f\|_{\text{lip}} = \sup \left\{ \frac{\sigma(f(x), f(y))}{p(x, y)} \mid \forall x, y \in X, x \neq y \right\}$$

distortion on f

$$\|f\|_{\text{lip}} \times \|f^{-1}\|_{\text{lip}}$$

$$\ell_p\text{-norm} \quad pt \quad x \in \mathbb{R}^d \quad p \in [1, \infty) \\ \|x\|_p = \left(\sum_{i=1}^d |x_i|^p \right)^{1/p}$$

Qb Does \exists some euclidean space into which a given metric space embeds isometrically

$$x = (x_1, \dots) \quad \text{of reals} \quad \|x\|_p < \infty \\ \|x\|_p = \left(\sum_{i=1}^d |x_i|^p \right)^{1/p}$$

Remark - ℓ_2 is a separable hilbert space
 \hookrightarrow has countable basis.

Known:

n dim. regular simplex has no isometric embedding in \mathbb{R}^k , $k < n$

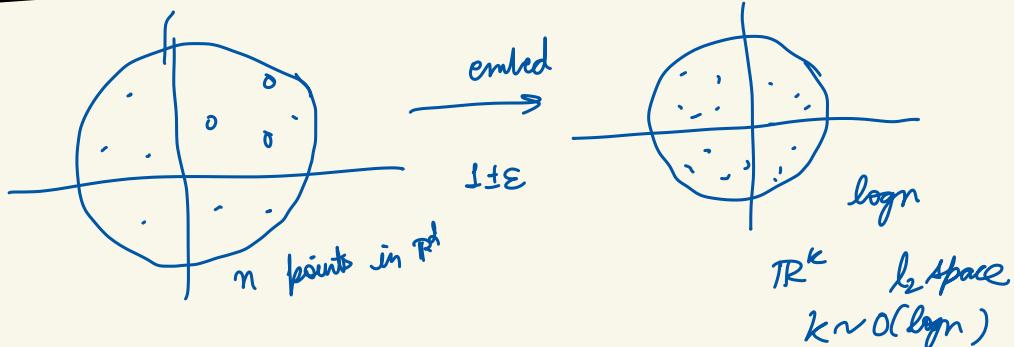
Johnson - Lindenstrauss Flattening Lemma

X be a n pt. set in \mathbb{R}^d $\epsilon \in (0, 1)$ \exists a $(1+\epsilon)$ embedding of X into ℓ_2^k $k \approx \Theta(\log n)$

Recap:

- Metric Embedding
- Isometric embedding
- Ex: isometric can't be preserved even for simple cases
- D-embedding

Johnson-Lindenstrauss Lemma



Lemma: given any $\epsilon \in (0, 1]$ and a set $X \subset \mathbb{R}^d$ with $|X| = n$
 \exists a linear map $f: \mathbb{R}^d \rightarrow \mathbb{R}^m$ with $m = O\left(\frac{\log n}{\epsilon^2}\right)$ that
extends (X, d_2) to (\mathbb{R}^m, d_2) with distortion ($\leq \epsilon$)

Idea: Use randomness to define the map f
 $A \in \mathbb{R}^{m \times d}$ be a random matrix with entries
chosen i.i.d from Gaussian.

$$f(x) = Ax$$

Hope - this exists with high probability

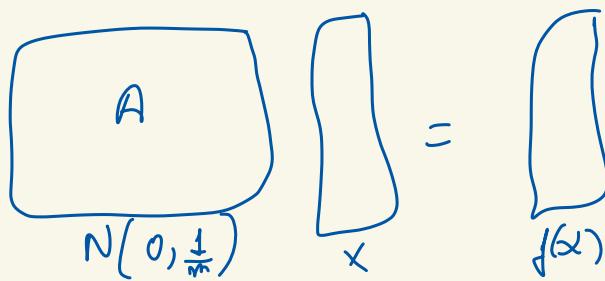
Goal - Identify such a f

$$A \begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} f(x) \end{bmatrix}$$

$N(0, \frac{1}{m})$

Geometry: If the rows of A were orthogonal then \exists orthogonal projection onto the row span of A .

Pick a random subspace given by the row span of A .
Project x onto that subspace.



Proof: Multiplying A preserves the ℓ_2 distances

Fix $x, y \in \mathbb{R}^d$ show with high probability P

$$\|A(x-y)\|_2 = (\pm \varepsilon) \|x-y\|_2$$

$$\|f(x)-f(y)\|_2$$

Then we will take the union bound of all $x, y \in X$ pairs.

Fact: Gaussian distribution = spherically symmetric

$A(x-y)$ is sort of scaled version of the first column
equivalent to show that $\|y\|_2 = 1 \pm \varepsilon$

Chernoff Style Bound

We have a set of ind. gaussian random variables

$$z_1, z_2, \dots, z_m \sim N(0, 1)$$

$$\Pr \left[\sum_i z_i^2 > (1+\varepsilon)m \right] \leq \underbrace{e^{-m\varepsilon^2/8}}_{\text{Small}}$$

Let's assume this happens.

$$\|y\|_2 = 1 \pm \varepsilon \text{ with high } p$$

y is a random vector with entries $\sim N(0, \frac{1}{m})$

$$\begin{aligned} \Pr \left[\|y\|_2 \geq 1 + \varepsilon \right] &= \Pr \left[\|y\|_2^2 \geq (1 + \varepsilon)^2 \right] \\ &\leq \Pr \left[\|y\|_2^2 \geq 1 + \varepsilon \right] \end{aligned}$$

def. of ℓ_2 -norm + eqn. 1

$$\Pr \left[\sum_i z_i^2 \geq m(1 + \varepsilon) \right] \leq e^{-m\varepsilon^2/8}$$

$$\Pr \left[\|Ax(x-y)\|_2 \geq (1 + \varepsilon) \|x-y\|_2 \right] \leq e^{-m\varepsilon^2/8}$$

Union bound $\forall x, y \in X \approx O(n^2)$

$$\Pr \left[\exists x, y \in X \text{ s.t. } \|Ax(x-y)\|_2 \geq ((1 + \varepsilon) \|x-y\|_2) \right] \leq n^2 \times e^{-m\varepsilon^2/8}$$

$$\text{Q: what is } m? \quad O\left(\frac{\log n}{\varepsilon^2}\right) \geq \frac{1}{\text{poly}(n)}$$

Fast JL transformation

- matrix multiplication is slow
- Via FFT (fast fourier transformation) we can speed up.

Motivation

Bring any metric to L_2

Bourgain's Embedding

Transformation from other metric to Euclidean.

(Bourgain 1985) [Linial, London, Rabani and BG'S]

Given any metric space (X, d) with $|X| = n$

\exists an embedding of (X, d) into (\mathbb{R}^k, d_1) where

$$k \approx O(\log^2 n)$$

$$\text{Distortion} \sim O(\log n)$$

$d_1 - L_1$ metric dist.
works for any L_p for
 $p \geq 1$

Remark:

distortion is tight because some metrics require $\Omega(\log n)$. Sometimes, it's possible to improve the dim.

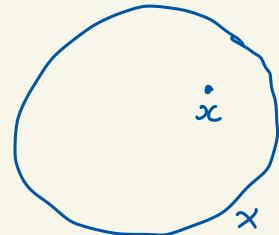
Algorithm:

- For $i = 1, 2, \dots, \log n \xrightarrow{\text{const. (will decide later)}}$
- For $j = 1, 2, \dots, c \log n$
- choose a set $S_{i,j} \subseteq X$ at random such that every $x \in X$ is contained in $S_{i,j}$ with probability 2^{-i} (independently)

Embedding (defⁿ):

$\forall x \in X, f(x) = (\underbrace{d(x, S_{1,1})}_{\text{minimum distance}}, d(x, S_{1,2}), \dots, d(x, S_{\log n, c \log n}))$

between x and any point of the set.



Q: Why this is good?

- (i) Non-expansive
- (ii) Non-shrinking

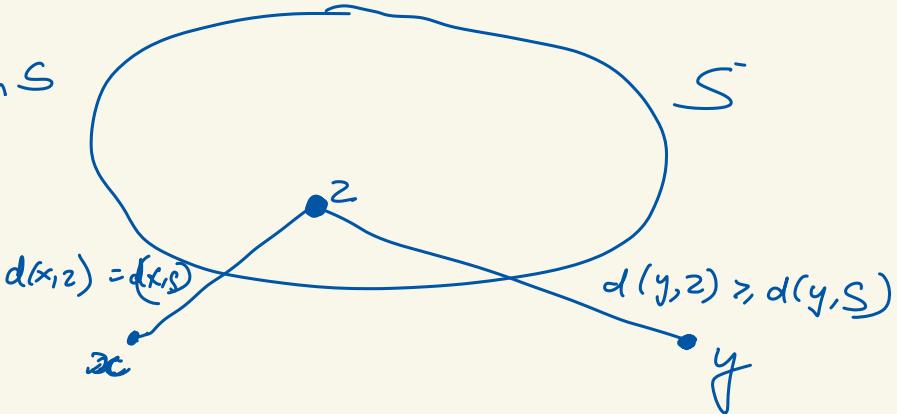
Non-expansive

This map is non-expansive
 $\forall x, y \in X$

non-expansive

$$\|f(x) - f(y)\| \leq d(x, y)$$

z is the closest point to x in S

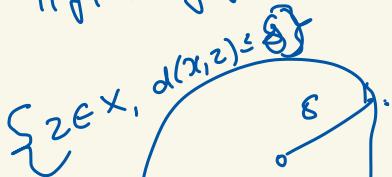


$$\begin{aligned}
 \|f(x) - f(y)\| &= d(y, S) - d(z, S) \\
 &\leq d(y, z) - d(x, z) \\
 &\leq d(x, y)
 \end{aligned}$$

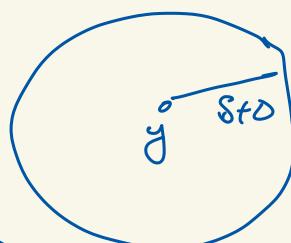
Triangle Inequality

Non-shrinking

$$\|f(x) - f(y)\| \geq \delta$$



bounding distortion above and below



$$\{z' \in X; d(y, z') \leq \delta + \epsilon\}$$

"nice instant"

$$d(y, S) - d(x, S) > (\delta + \Delta) - \delta = \Delta$$

Idea: Pick a lot of random S and our hope is that

"nice instance" happens enough number of times.
(inner loop) — this happens if S has enough intensity.

- Good choice of S depends on $d(x, y)$ (outer loop)
it varies the size of S

$$\underbrace{k d(x, y)}_{\text{log } n} \stackrel{\text{ineq-2}}{\leq} \|f(x) - f(y)\| \stackrel{\text{ineq-1}}{\leq} k \times d(x, y)$$

$$\underbrace{\|f(x) - f(y)\|}_{\text{ineq-1}} \leq k \times d(x, y)$$

enough to show $|d(x, S) - d(y, S)| \leq d(x, y)$ for every S .
 $\Rightarrow d(x, S)$ is non-expanding

$$\|f(x) - f(y)\|_1 = \sum_{i,j} d(x, s_{ij}) - d(y, s_{ij}) \leq k \times d(x, y)$$

$$\underbrace{k \times d(x, y)}_{\text{log } n} \leq \|f(x) - f(y)\|_1$$

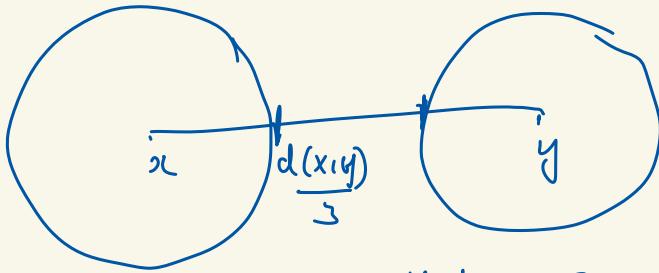
choose bunch of different deltas (S 's) A "nice" thing holds with good prob.

$$\text{fix } x, y = 0$$

$$\text{choose } 0 = \delta_0 < \delta_1 < \delta_2 < \dots < \delta_t$$

s.t. δ_i : δ_i is the smallest S

$B(x, \delta_i) \& B(y, \delta_i)$ contain $> 2^i$ pts.



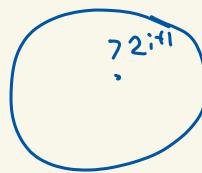
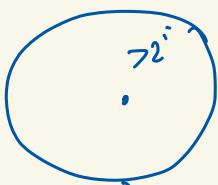
Stop at the tangent k so that $s_t < \frac{d(x,y)}{3}$

$$s_{t+1} = \frac{d(x,y)}{3}$$

nice instance holds $|d(x,s) - d(y,s)| > s_{t+1} - s_t$

Claim: It is very likely that this situation happens.

By def^m of $s_i > 2^i$



$$P[x \in S_{i,j}] = 2^{-i}$$

$$\begin{aligned} P[\text{nice event happens}] &= P[\text{non-empty intersection happens}] \times P[\text{disjointness happens}] \\ &\geq \left(1 - \left(1 - \frac{1}{2^i}\right)^{2^i}\right) \end{aligned}$$

$$\begin{aligned} &\geq \left(1 - \frac{1}{e}\right) \times \left(1 - \frac{1}{2}\right)^{2^i} \\ &\geq \frac{1}{2^{\frac{1}{2}}}, \end{aligned}$$

Claim: Fix i
 $\Pr \left[\sum_{j=1}^{c \log n} \text{function of } S_{i,j} \text{ have nice event} \right]$

1 - $\Pr \left[\sum_{j=1}^{c \log n} \text{if } \left\{ \text{nice event} \right\} \leq \frac{1}{2} \times \frac{c \log n}{2^6} \right]$

Invoke Chernoff bound

$\geq 1 - \exp \left(-\frac{c \log n}{8 \times 2^6} \right) \geq 1 - \frac{1}{n^3}$ if we choose $c \geq 3 \times 2^6$

Putting these together
By a union bound $\approx n^2$ fair and $c \log n$ choices of j
with high prob. $\forall i \forall x, y \geq \frac{c \times \log n}{2^6}$

$$\begin{aligned} & |d(x, \underset{S_{i,j}}{\cancel{s_{ij}}}) - d(y, \underset{S_{i,j}}{s_{ij}})| \geq s_{i+1} - s_i \\ & ||f(x) - f(y)|| \leq \sum_{i,j} |d(x, \underset{S_{i,j}}{s_{ij}}) - d(y, \underset{S_{i,j}}{s_{ij}})| \\ & \geq \sum_{i,j} \left(\frac{c \log n}{2^6} \right) (s_{i+1} - s_i) \\ & = \frac{c \log n}{2^6} \times s_{i+1} = \frac{c \log n}{3 \times 2^6} \times d(x, y) \end{aligned}$$

$$k = c \log n \geq \frac{k \times d(x, y)}{3 \times 2^6 \times \log n}$$

non-shrinking

Approximate Nearest Neighbour Search in Low Dimensions

- * Voronoi Diagram = $O(n^{d(d)/2})$
- * $(1+\epsilon)$ -approximate neighbours if $\|q-s\| \leq (1+\epsilon)d(q, P)$
- Alternatively $\forall t \in P \quad \|q-s\| \leq (1+\epsilon) \|q-t\|$
- * Spread of a point set $P = \phi(P) = \text{ratio of diameter and the distance of the closest pair of } P.$

* Bounded spread case

Algorithm: $A_0 = \{\text{root}(\tau)\}$ $\tau_{\text{curr}} = \|q - \text{rep}_{\text{root}(\tau)}\|$

i-th iteration $\rightarrow i > 0 \quad A_{i-1} \rightarrow A_i$
 $v \in A_{i-1} \Rightarrow C_v = \text{set of children of } v \in \bigcup_j \square_j$

$w \in C_v \quad \tau_{\text{curr}} \leftarrow \min(\tau_{\text{curr}}, \|q - \text{rep}_w\|)$ quadrants of P

Algorithm checks if $\|q - \text{rep}_w\| - \text{diam}(\square_w) < (1 - \frac{\epsilon}{2}) \tau_{\text{curr}}$
 if so add w to A_i :

$$\begin{aligned} \text{Correctness: } \|q - \text{nn}(q)\| &\geq \|q - \text{rep}_w\| - \|\text{rep}_w - \text{nn}(q)\| \\ &\geq \|q - \text{rep}_w\| - \text{diam}(\square_w) \\ &\geq \left(1 - \frac{c}{2}\right) \tau_{\text{curr}} \end{aligned}$$

$$\frac{\|q - \text{nn}(q)\|}{\left(1 - \frac{\epsilon}{2}\right)} \geq \tau_{\text{curr}} \quad \frac{1}{1 - \frac{\epsilon}{2}} \leq 1 + \epsilon$$

* Let $P = \text{set of } n \text{ points contained inside the unit hypercube in } \mathbb{R}^d$ and let $\mathcal{T} = \text{quadtree of } P$, where $\text{diam}(P) = 2(1)$.
 $q = \text{query point}$, $\varepsilon > 0 = \text{parameters } (1+\varepsilon)-\text{ANN to } \varepsilon$
 can be computed in $O(\varepsilon^{-d} + \log(\frac{1}{\varepsilon}))$ time $\bar{w} = d(q, P)$

Proof: If $w \in \mathcal{T}$ = considered by the algorithm and
 $\text{diam}(\Delta w) < (\frac{\varepsilon}{n}) \bar{w}$ then

$$\begin{aligned} \|q - \text{rep}_w\| - \text{diam}(\Delta w) &\geq \|q - \text{rep}_w\| - (\frac{\varepsilon}{n}) \bar{w} \\ &\geq \varepsilon_{\text{min}} - (\frac{\varepsilon}{n}) r_{\text{max}} \\ &\geq (1 - \frac{\varepsilon}{n}) r_{\text{max}} \end{aligned}$$

\Rightarrow no nodes of depth $\geq h = \lceil -\lg(\frac{\bar{w}\varepsilon}{n}) \rceil$ are considered

The number of relevant cells (i.e. cells that the algorithm explores and do not get immediately thrown out) = number of grid cells of $G_{2^{-h-1}}$ that intersect a box of sidelength $2h$, centered at q :

$$n_i = \left(2 \left(\frac{l_i}{2^{i-1}}\right)\right)^d = \left(1 + (2^i \bar{w})^d\right)$$

$$(a+b)^d \leq (2 \max(a, b))^d \leq 2^d (a^d + b^d)$$

$$\sum_{i=0}^h n_i = O\left(\log \frac{1}{\bar{w}} + \frac{1}{\varepsilon^d}\right)$$

Dynamic Inputs

- * Online Algorithms
- * Dynamic Algorithms
- * Streaming Algorithms

Online Algorithms

I - instance
 σ - ordering

when σ_i arrives, we have to make the decision instantly

$$\langle \sigma_1, \sigma_2, \dots, \sigma_n \rangle$$

Decisions are irrevocable

measure the optimality

$$\text{Competitive ratio} = \sup \frac{\text{Alg}(\sigma)}{\text{OPT}(\sigma)}$$

Dynamic \longrightarrow e.g.: B.S.T (Binary Search Tree)

I - instance

$$\sigma = \langle \sigma_1, \dots, \sigma_m \rangle$$

insertion and deletion both are allowed.

Goal: Build data structures which can handle insertion + deletion

optimize: Update complexity

Query complexity
 optimality

Streaming

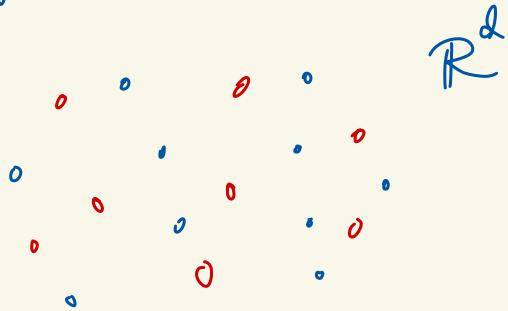
I - instance

σ - sequence

optimize: space complexity
 & Query & Time

Data Reduction (Core set)

A small subset of the input which captures all interesting features



Coreset for directional width

$|P| - |P'| = n$ pts in \mathbb{R}^d ; for any $l \in \mathbb{N}$
Projecting P onto the direction $U \in \mathbb{R}^d$ in the set

→ P onto a line l through the origin

Formally, for a vector $U \in \mathbb{R}^d$ $U \neq 0$

$$\bar{\omega}(U, P) = \max_{p \in P} \langle U, p \rangle - \min_{p \in P} \langle U, p \rangle$$

Properties: (a) Its translation invariant for any vector $s \in \mathbb{R}^d$

$$\bar{\omega}(U, s + P) = \bar{\omega}(U, P)$$

(b) Direction width scales linearly

(c) for any set $P \subseteq \mathbb{R}^d$

$$\bar{\omega}(U, P) = \bar{\omega}\left(U, \bigcup_{i=1}^k C_i(P)\right)$$

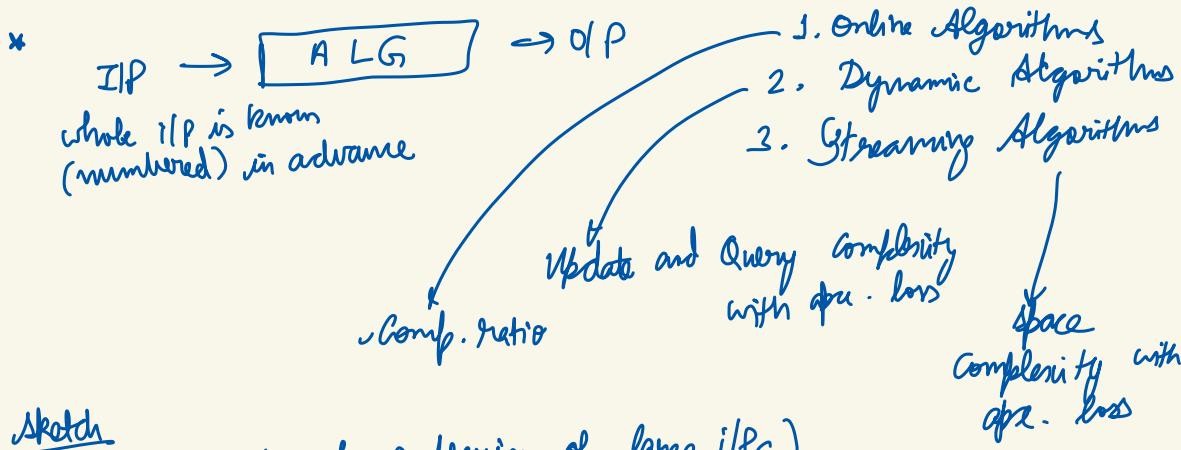
Convex Hull

(d) for any $Q \subseteq P$ and any vector U

$$\bar{\omega}(U, Q) \leq \bar{\omega}(U, P)$$

Coreset - $S \subseteq P$ is ε -coreset
 for directional width if
 sphere of directions in \mathbb{R}^d $\forall v \in S^{(d-1)}, \bar{w}(u, S) > (1-\varepsilon) \bar{w}(v, P)$
Sketch: $X \subseteq Y \subseteq P \subseteq \mathbb{R}^d$
 Y is an ε -coreset (for directional width) of P
 and X is an ε' -coreset of Y . Then X is a $(\varepsilon + \varepsilon')$ coresset of P .

Merge $X \subseteq P \subseteq \mathbb{R}^d$ and $X' \subseteq P \subseteq \mathbb{R}^d$
 then $(X \cup X')$ is also a coresset of $(P \cup P')$

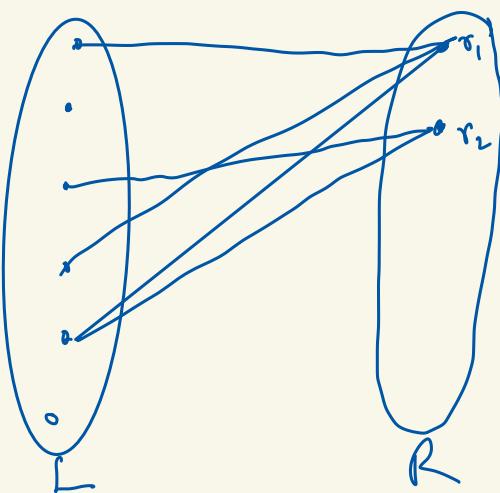


Sketch
Core set (sketch for compression of large i/Ps)

Online Algo.

I - instance
G - ordering

$\langle \sigma_1, \dots, \sigma_n \rangle$ At step i , $\sigma_i \in \{r\}$



I/P: L is known in advance

R comes one by one

a vector $v \in R$ arrives with all incident edges

- * A vertex $w \in R$ can be matched only at the time of its arrival.

Goal = maintain the maximum matching

Greedy Algo (Deterministic)

when a vertex $v \in R$ arrives look at the neighbourhood (S_v) look at the unmatched neighbours & match with an arbitrary vertex.

Thm (LB): It is not possible to get better than $\frac{1}{2}$ for any deterministic alg.

Thm: We can get $\frac{1}{2}$ by using Greedy.

Proof: via weak LP-duality.

$$L \cup R = V$$

Primal: $\max_{\mathbf{x} \in E} \sum_{e \in E} x_e$ (edge prices)

E

sub to $\sum_{e \in S(v)} x_e \leq 1$ for all $v \in L \cup R$

$x_e \geq 0$ for all $e \in E$

Dual: $\min_{v \in L \cup R} \sum_{v \in U \cup R} p_v$ (vertex prices)

sub to $p_u + p_v \geq 1 + (u, v) \in E$
 $p_v \geq 0$ & $u \in L \cup R$

for every vertex $v \in L \cup R$

$$q_v = \begin{cases} \frac{1}{2} & \text{if greedy matches } v \\ 0 & \text{otherwise} \end{cases}$$

* Comparing with the feasible solution in the dual setting -

when the algorithm adds (v, v) to its matching $q_u, q_v, -\frac{1}{2}$

$$|M| = \sum_{v \in L \cup R} q_v$$

m -matching that greedy algorthm computes

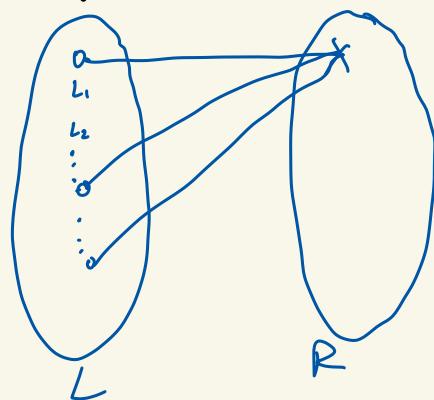
obs: Finally $(L \cup R, E)$ at least one of q_u, q_v is $\frac{1}{2}$

if not both

implies if we scale up q by a factor 2 $P = 2q$

$$|M| = \frac{1}{2} \leq P_v \geq \frac{1}{2} \times \text{opt}$$

online Bipartite Matching



I/P : left side of the bipartite matching

O/P : maximum matching maintained

$$\xrightarrow[\text{sups}]{\text{Comp. ratio}} \frac{\text{ALG}(r)}{\text{OPT}(r)}$$

Thm: There exist no deterministic online alg. with Comp. ratio better than $\frac{1}{2}$.

Thm: This is tight (e.g.: greedy algos)

Random matching

- whenever a vertex (v_i) appears in R (for any $i \in [n]$)
- Pick a random unmatched vertex (say l_j) match.

Expt. size $\frac{n}{2} + O(\log n)$

$$\text{Matrix } B_{ij} = \begin{cases} 1 & \text{if } i=j \text{ or } \begin{array}{c} \text{if } \frac{n}{2} \leq j \leq n \\ 1 \leq i \leq \frac{n}{2} \end{array} \\ 0 & \text{otherwise} \end{cases}$$

Ranking

Initialization: Pick a random permutation (say, π) of the left side.

Matching phase: When a vertex arrives (v_k) match v_k to vertex ($l_j \in L$) with highest rank.

Lower Bound claim: Thm: It is not possible to get better than

$$n\left(1 - \frac{1}{e}\right) + O(n)$$

T - $n \times n$ - complete upper-triangular matrix
Assume that columns of T arrive in the order $\langle n, n-1, \dots \rangle = \sigma$
 k^{th} column arrival - $(n-k+1)$

Def^m: $T - n \times n$
 with every permutation π on $\{1, \dots, n\}$ associated with
 $\langle T, \pi \rangle$. Let P be the uniform prob. distribution
 over $n!$ instances.

Lemma:- A deterministic algorithm "greedy" Then exp. size of the matching which A computes $\langle T, \pi \rangle$ is same as the exp. size of the matching that "random" computes.

Lemma:- The performance of some online algorithm (Alg) is upper bounded by exp. size what random achieves on $\langle T, \pi \rangle$

Brief :- For Alg A for $\langle T, \pi \rangle$ as we "random" if we have k available rows at some time t . Then there are equally likely to be $\binom{n}{k}$ rows for the first $n-t+1$ rows of T .

② for each k , the prob. that there are k eligible options at time t is same for random as it is for A .

Proof: Let $\mathbb{E}[R, \langle T, \pi \rangle]$ - exp size of matching by some randomized Alg.

$$\mathbb{E}[A] = \min_{\pi} \mathbb{E}(R, \langle T, \pi \rangle) \leq \underbrace{\max_A \{ \mathbb{E}[A(P)] \}}_{\text{Yao's min-max principle}}$$

Yao's lemma

for any randomized Alg \exists a prob. distribution over the I/P for Alg. s.t. the exp. cost of Alg on its worst case is as large as the best deterministic Alg on a random i/p from the same distribution.

Lemma :- Expected size of matching by a random alg. over T is $n\left(1 - \frac{1}{e}\right) + O(n)$

Proof : $x(t), y(t) \rightarrow$ random variables - # of columns remaining and # of rows still available at t

$$\Delta(x) = x(t+1) - x(t)$$

$$\Delta(y) = y(t+1) - y(t)$$

$$\begin{aligned} \mathbb{E}[\Delta y] &= -1 - \frac{y(t)}{x(t)} \cdot \frac{y(t)-1}{y(t)} \\ &= -1 - \frac{y(t)-1}{x(t)} \end{aligned}$$

$$\frac{\mathbb{E}[\Delta y]}{\mathbb{E}[\Delta x]} = 1 + \frac{y(t)-1}{x(t)}$$

Kartz's thm : As the prob. tends to 1 as n tends to ∞

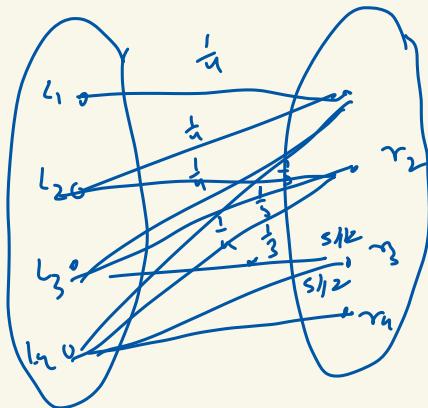
$$\frac{dy}{dx} = 1 + \frac{y-1}{x}$$

$$y = 1 + x \left(\frac{n-1}{n} - \log \frac{x}{n} \right)$$

Waterfilling Algorithm

Each vertex $v \in L$ as v contains of capacity 1.
 → when a vertex $w \in R$ arrives, it pours water of unit 1.

- Pour in the current lowest container
- Keep doing until equal



Dynamic Optimality

Binary Search Trees (BST)

- AVL
- Red - Black

operations $\rightarrow O(\log n)$

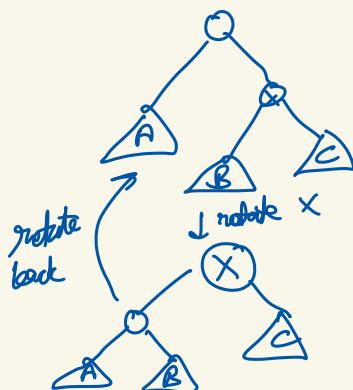
Q: Is this the best complexity that we can hope for?

BST-model (pointer-machine model)

Data is stored as keys in BST.

Operations :

- left, right, parent
- rotation



search(x)

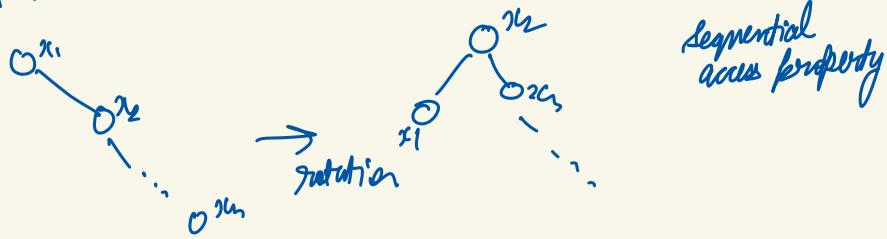
- start at root
 - must visit node with key x
- ↳ LB can't beat logn (Adversarial)

Q: Is this logn always required
 - Depends on the sequence (σ) that we have.

Keys - $\{1, 2, \dots, n\}$

Sequence σ - $\langle x_1 < x_2 < x_3 \dots < x_n \rangle$

Maintain a data structure s.t. amortized is $O(1)$

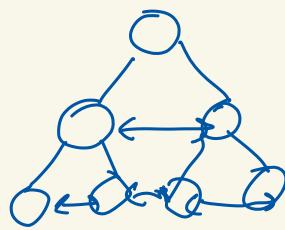


Stronger Version:

Dynamic finger property

$$|x_i - x_{i-1}| = k$$

$\Rightarrow O(\log k)$ operations



Static OPT / Entropy Bound

k appears Pn function of time

$$O\left(\sum_{i=1}^n p_i \log \frac{1}{p_i}\right)$$

amortized per operation

Working set property (roughly output sensitive)

If t_i distinct keys have been accessed so far

$$O(\log t_i) \text{ / operation}$$

Unified property

t_{ij} distinct keys accessed in x_i, \dots, x_j then x_i costs

$$O(\log \min_{i \leq j} [(x_i - x_j) + t_{ij} + 2])$$

Generalizes dynamic finger property

Conjecture - $O(1)$ complexity

Total cost of access keys = $O(OPT)$

What is OPT ? minimum over all BST Algo
(offline) over the access seq. X .

Q Does \exists online BST which $O(1)$ to the offline OPT .

$O(\log n) \rightarrow$ trivial

$O(\log \log n) \rightarrow$ Best known

Splay tree, Greedy Conj.

Primal Dual Method

Linear programming

$A \rightarrow m \times n$ matrix

$x \rightarrow n$ vector

$b \rightarrow m$ vector

$w \rightarrow m$ -vector

$$\begin{aligned} Ax &\geq b \\ x &\geq 0 \\ \min(w^T x) \end{aligned}$$

$$y^T A x \geq y^T b$$

$y \rightarrow m$ -vector

multiply i^{th} inequality

by $y_i \geq 0$

add up all the inequalities

Suppose this holds

$$y^T A \leq w^T$$

$$w^T x \geq y^T A x$$

$$\geq y^T A b$$

for any solution
 x to primal and
 y for the dual

$$\max(y^T b)$$

$$y^T A \leq w^T \Leftrightarrow A^T y \leq w$$

$$y \geq 0$$

→ dual of the linear program

Suppose (\bar{x}, \bar{y}) are solutions to the primal and dual that satisfy following conditions -

(i) if $\bar{x}_i > 0$ the dual constraint corresponding to x_i is satisfied exactly (inequality is an equality).

→ i^{th} row of A^T

(ii) if $\bar{y}_j > 0$ the primal constraint corresponding to j^{th} row in A is tight (an equality).

→ Complementary Slackness

If these conditions are satisfied then the cost of x^* and y^* are the same and both are optimal solutions for respective LPs.

$$w^T x^* = \sum_{i=1}^n w_i x_i^* = \sum_{i=1}^n w_i x_i^* = \sum_{i, x_i^* > 0} \left(\sum_{j=1}^m a_{ij} y_j^* \right) x_i^*$$

$x_i^* > 0$ because the corresponding inequality is tight

if $x_i^* > 0$
 $A^T y = w^T$

then the i^{th} row
is an equality

$$= \sum_{i, x_i^* > 0} \left(\sum_j a_{ij} y_j^* \right) x_i^*$$

$$= \sum_{y_j^* > 0} \left(\sum_{x_i^* > 0} a_{ij} x_i^* \right) y_j^*$$

$$= \sum_{y_j^* > 0} b_j y_j^*$$

second condition
of Complementary slackness

$$= y^T b$$

Integer Linear Programming

Linear programs with the additional restriction that the variables are required to take integer values

↳ NP Hard

* Vertex Cover in a graph

Given an undirected graph with positive weights assigned to vertices, find a minimum subset of vertices such that every edge has at least one endpoint in the subset.

$x_i \rightarrow$ for each vertex i
 $x_i = 0$ means vertex i is not in the subset.
 $x_i = 1$ means it is in the subset.

$$\text{wt. of subset} = \sum_{i=1}^n w_i x_i$$

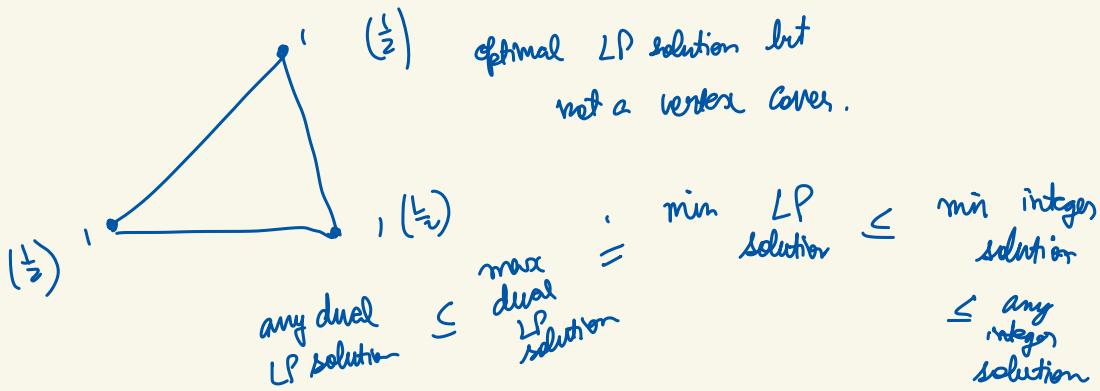
ILP formulation

$$\min \sum_{i=1}^m w_i x_i$$

for every edge $ij \quad x_i + x_j \geq 1$

for all vertices $x_i \geq 0$

LP-relaxation allows fractional values for x_i



dual vertex cover

$$\begin{aligned} & \min \sum w_i x_i \\ & x_i + x_j \geq 1 \text{ each edge} \\ & x_i \geq 0 \end{aligned}$$

y_{ij} for each edge ij

$$\begin{aligned} & y_{ij} \geq 0 \\ & \text{for every vertex } i \\ & \sum_{\substack{\text{all edges} \\ ij \text{ incident} \\ \text{with } i}} y_{ij} \leq w_i \end{aligned}$$

even this could be approximate

Approximate complementary slackness

{ if $x_i > 0$ then corresponding inequality is tight
if $y_i > 0$ then corresponding formal inequality is approximately tight.
instead of requiring equality, we allow LHS to be some constant C times RHS.

Initially start with all dual variables = 0 and $x_i = 0$

→ Pick any edge and start increasing its dual value.

↳ do this till the dual inequality corresponding to at least one of the endpoints becomes tight.

↳ Select the endpoint for which the inequality becomes tight. and
(include in vertex cover)
delete all edges incident with it.

→ Repeat until all edges are deleted.

* a vertex is selected only when the corresponding dual inequality is tight.

primal inequality is approximately satisfied with a constant factor of 2.

because for any edge $ij \quad 1 \leq x_i + x_j \leq 2$

Cost of integer solution $\leq 2 \times$ cost of dual LP solution

Set Cover

Let V and a collection of subsets of V

$\{S_1 \dots S_m\}$

each set S_i has wt w_i

Find min-wt collection $C' \subseteq C$ such that every element in V is contained in some subset in C' .

$\{e_1 \dots e_m\}$

$\{S_i$
edges incident with vertex i
 $(v_i)\}$

Vertex cover = special case of set cover

Assume that every element belongs to some set in C

$x_i \rightarrow$ let $s_i = 0$ if s_i is not selected
 $= 1$ if s_i is selected.

$$\min (\sum w_i x_i)$$

for every element a in U , $\sum_{i, \text{ such that } a \in s_i} x_i \geq 1$
 (one set has to be selected).

If every element in U belongs to at most k sets
 k -approximation algorithm.

Shortest Path with positive costs on edges

think of this as a set cover problem

directed graphs \rightarrow two vertices s and t

+ve integers wts to the edges

min-cost path from s to t .

set cover

$\{x_1, \dots, x_n\}$ and a collection of subsets of $U \rightarrow$ their union is U
 cost assigned to each subset

min wt collection of subsets whose union is U .

For edge e_i , we have a variable e_i ($\min w_i e_i$)

for every subset S of vertices that includes s but not t .

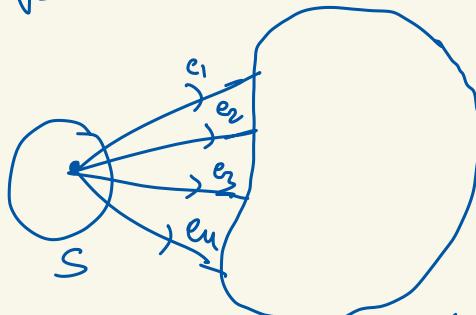
$$\sum e_i \geq 1$$

all edges e_i that join a vertex in S to a vertex not in S

\downarrow
 2^{n-1} possible constraints

$y_S \rightarrow$ increasing its value

→ till inequality
for some edge
becomes tight



↓
the min wt
edge leaving S

↓
we can include
it in the set
cover.

for an edge e the dual constraint

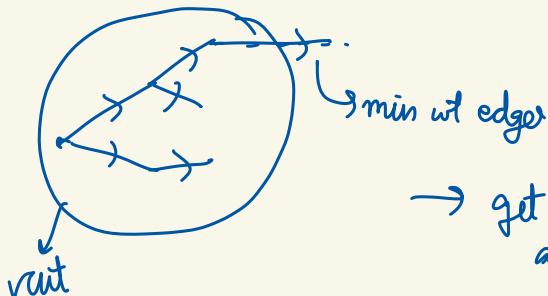
$$\sum_S y_S \leq w_e$$

such that
 e goes from
 S to S'

the weights of other edges are
reduced by w_{\min}

Consider this
as the cut

Now we will check these
edges in the next iteration



→ get a set cover as soon
as t gets added to the
tree

delete all edges not on the path from s to t .

Finally when only edge in a path are left, for every cut with a non-zero dual value, exactly one edge in the cut will be included in the solution.

If a dual variable is not zero then the corresponding primal inequality is tight.

Undirected graph with positive weights assigned to vertices. Find a min wt. subset of vertices whose removal leaves only isolated vertices or edges.

Vertex cover \rightarrow find a subset of vertices whose removal only leaves isolated vertices.

set of all edges $\vee^{\{ \text{edges incident with } v \}}$ \rightarrow Vertex cover

set of all paths of length 2 (3 vertices)

{ set of all paths of length 2 that contain v }

V

x_v

3- approximation

$\min w_v x_v$

For every path $v_1 v_2 v_3$
of length 2

$$x_{v_1} + x_{v_2} + x_{v_3} \geq 1$$



km (all weights 1)

optimal integer solution
is n-2

$$x_v = \frac{1}{3}$$

satisfies all inequalities, it is a fractional solution whose cost is $\frac{n}{3}$.
 $\frac{n}{3} \approx \frac{1}{3} (\text{Cost of OPT-integer solution})$

Integrality gap of the LP ratio between $\frac{\text{optimal integer solution}}{\text{optimal fractional solution}}$

↓
 cannot hope to prove a better bound than this for a primal dual problem.

for any subset S of 3 or more vertices that induces a connected graph

$$\sum_{v \in S} d_S(v) x_v \geq |E_S| - \frac{|S| - \tau(S)}{2}$$

↓
 no. of neighbours
 of v in S

where $\tau(S)$ denotes
 the min number of
 vertices from S to be removed
 to leave only isolated vertices.

number of
 subgraphs may
 be exponential

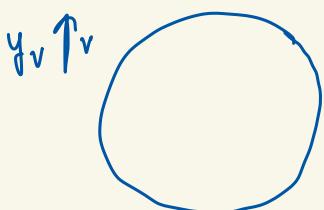
initially we have a connected graph with at least 3 vertices.

dual inequality of a vertex

$$\sum_{\substack{\text{all such} \\ \text{sets containing} \\ v}} d_S(v) y_S \leq w_v$$

Initially increase y_S for the set S of all vertices in G .

tilt it becomes tight for some vertex



↓ This one for which $\frac{w_v}{d(v)}$ is minimum.

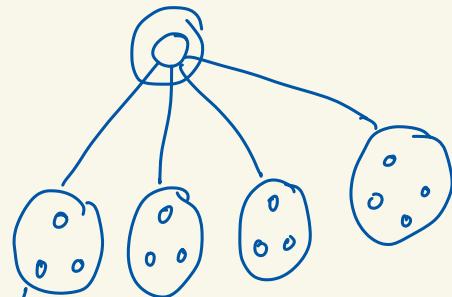
Select the vertex with minimum value $\frac{w}{\text{degree}} = S$

$$y_v = S$$

reduce the weights of all other vertices u by $S \cdot d(u)$

$$\text{new-wt of } v = \text{original-wt of } u - \frac{w(v)}{\deg(v)} \times \deg(u)$$

delete the selected vertex, all edges incident with it
and recursively work with all connected components with at least 3
vertices.

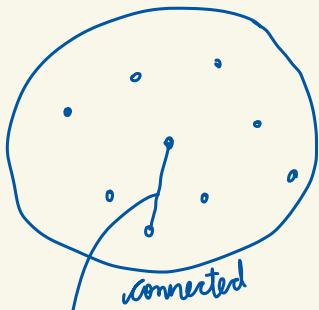


After the recursive call, check if initially selected vertex v required i.e. whether deleting it still gives a valid solⁿ and if no, delete it.

whenever a vertex is selected \Rightarrow dual inequality is tight

For any minimal subset with the required property in the graph $G(S)$ where y_S is not zero, satisfies the primal inequality within a factor of 2.

$$\sum_{x \in X} d(x) \leq 2|E| - |S| + \tau(S)$$

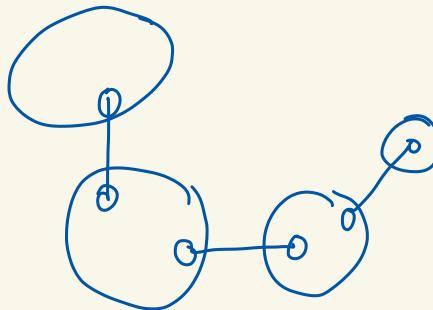


$$S = V$$

connected

Adds 2 to both LHS and RHS, hence we can reduce the graph.

$|E| \geq |S|-1$ graph is connected



Connected

k edges between vertices in X such that removing any one disconnects the graph then

$$\tau(S) \geq k+1$$

This is satisfied for any minimal subset X and connected graph G with at least 3-vertices.

* If we are allowed to leave complete subgraphs, then 3-apx can be found.

Approximate Complementary slackness

Let $x = (x_1, \dots, x_n)$ and $y = (y_1, \dots, y_m)$ be feasible solutions to the primal and dual linear programs, respectively, satisfying the following conditions:

- Primal complementary slackness: for $\alpha \geq 1$, for each i , $1 \leq i \leq n$, if $x_i > 0$ then $\frac{c_i}{\alpha} \leq \sum_{j=1}^m a_{ij} y_j \leq c_i$
 - Dual complementary slackness: for $\beta \geq 1$, for each j , $1 \leq j \leq m$, if $y_j > 0$ then $b_j \leq \sum_{i=1}^n a_{ij} x_i \leq b_j \cdot \beta$.
- Then $\sum_{i=1}^n c_i x_i \leq \alpha \cdot \beta \cdot \sum_{j=1}^m b_j \cdot y_j$

Greedy Algorithm for the set cover problem

Initially, $C = \emptyset$. Let U be the set of yet uncovered elements. As long as $U \neq \emptyset$, let $S \in S$ be the set that minimizes the ratio $\frac{c_S}{|U \cap S|}$:

- (i) Add S to C and set $x_S \leftarrow 1$
- (ii) for each $e_i \in (U \cap S)$, $y_{e_i} \leftarrow \frac{c_S}{|U \cap S|}$

The greedy algorithm is an $O(\log n)$ -approximation algorithm for the set-cover problem:

Proof: P, D = values of the objective functions of the primal and dual solutions produced.

- (1) Clearly if there exists a feasible primal solution, then the algorithm will also produce a feasible solution.
- (2) In a iteration $\rightarrow \Delta P = c_S$ (addition of set S)
 $\Delta D = c_S$ (we set the variables corresponding to $|U \cap S|$)
 $\frac{c_S}{|U \cap S|}$.
- (3) e_1, e_2, \dots, e_n be the elements belonging to the set S , sorted with respect to the order they were covered by the greedy algo.
 Element e_i : $y_{e_i} \leq \frac{c_S}{(k-i+1)}$. Algo chooses the set that

minimizes the ratio of cost to number of uncovered elements.
 At the time, e_i is covered, sets contained at least $k-i+1$
 elements still uncovered.

$$\sum_{e_i \in S} y_{e_i} \leq \sum_{i=1}^{k-i} \frac{c_s}{k-i+1} = k_k \cdot c_s = c_s \cdot O(\log n)$$

$y_k = k^{\text{th}}$ harmonic number
 $D' = \frac{D}{O(\log n)}$

$P \leq D$. By claim (3) we get that primal is at most $O(\log n)$ times a feasible dual solution.
 By weak duality, feasible dual \leq feasible primal.
 \Rightarrow Primal solution $\geq (O(\log n))$ optimal - primal solution.

Randomized Routing Algorithm:

- (1) For each set $s \in S$, choose $2lmn$ independently random variables $x(s,i)$ uniformly at random in the interval $[0,1]$.
- (2) For each set s , let $\Theta(s) = \min_{i \in s} x(s,i)$.
- (3) Solve linear program (P) .
- (4) Take set s to the cover if $\Theta(s) \leq x_s$.

The algorithm produces a solution with the following properties:

- (1) The expected cost of the solution is $O(\log n)$ times the cost of the fractional solution.
- (2) The solution is feasible with prob. $1 - \frac{1}{n} > \frac{1}{2}$.

Proof: For each i , $1 \leq i \leq 2lmn$, the prob. that $x(s,i) \leq x_s$ is exactly x_s .

s is chosen $\rightarrow \exists i$ such that $x(s,i) \leq x_s$
 $A_i = \text{event } x(s,i) \leq x_s$
 Prob that s is chosen = $\bigcup_{i=1}^{2lmn} A_i \rightarrow$ atmost the sum of individual prob which is $2x_s lmn$.

Using linearity of expectation, expected cost of the solution is at most $2lmn$ times the cost of the fractional solution.

To prove (2), pick an element e .

$$\prod_{\substack{S \in S \\ e \in S}} (1 - x_S) \leq \exp(-\sum x_S) \leq \exp(-1)$$
$$(1 - x \leq \exp(-x))$$

Probability that e is not covered due to is the probability that none of the variables $x(S_i)$ result in choosing a set S covering e . Since we choose $2mn$ random variables, the prob. that e is not covered is at most $\exp(-2mn) = \frac{1}{n^2}$. Using union bounds, prob. that there exist e which is not covered is at most $\frac{1}{n^2} \times n = \frac{1}{n}$.

Primal-Dual Schema

while there exists an uncovered element e_i :

- (1) Increase the dual variable y_{e_i} continuously.
- (2) If there exists a set S such that $\sum_{e \in S} y_e = c_S$: take S to the cover and set $x_S \leftarrow 1$.

f = maximum number of sets an element belongs to.

The algorithm = f -approx.

Both the primal and dual constraints are satisfied.

Proof: Both the primal and dual constraints are satisfied. Primal complementary slackness condition holds with $\alpha=1$ since

if $x_S > 0$ then $\sum_{e \in S} y_{e_i} = c_S$.

Dual complementary slackness holds with $\beta=f$ since if $y_{e_i} > 0$

$1 \leq \sum_{S \ni e_i} x_S \leq f \Rightarrow$ Use prev. thm.

For vertex cover, $f=2n$.

Ski-Rental Problem

Rent $\rightarrow \$1$ per day

Buying $\rightarrow \$B$

Unknown = number of skiing days remaining.

$x \rightarrow 1$ if the skier buys the skis
For each day j , $1 \leq j \leq k \rightarrow z_j \rightarrow 1$ if the skier rents on day j

$$x + z_j \geq 1$$

$$x \in \{0, 1\}$$

$$z_j \in \{0, 1\}$$

Dual: $y_j \leq 1 \quad \sum_{j=1}^k y_j \leq B \quad \text{max } \sum_{j=1}^k y_j$

optimal Deterministic Algorithm:

rent the skis for the first $B-1$ days (2-competitive,
Buy them on the B^{th} day.)

Dual-dual: On the j^{th} day, a new primal constraint $x + z_j \geq 1$ dual variable y_j

Primal is already satisfied then do nothing.
otherwise increase y_j continuously, until some dual constraint becomes tight, set the corresponding primal variable to 1.

If $y_j > 0$ then $1 \leq x + z_j \leq 2$
 if $x > 0$ then $\sum_{j=1}^k y_j = B$
 if $z_j > 0$ then $y_j = 1$

} Apply Thm.

Algorithm.

(1) Initially $x \leftarrow 0$ (j^{th} new constraint), if $x < 1$:

(2) Each new day (j^{th} new constraint), if $x < 1$:

$$(a) z_j \leftarrow 1-x$$

$$(b) x \leftarrow x(1+\frac{1}{B}) + \frac{1}{c \cdot B}$$

$$(c) y_j \leftarrow 1$$

Number of ski-days = k
 Since $z_j = 1 - x \rightarrow$ primal feasible
 Increments in $x = z_1, \dots, z_k$ $x = \sum_{j=1}^k z_j$

$$x_i = \frac{1}{CB} \quad q = 1 + \underbrace{\left(1 + \frac{1}{C}\right)^B}_{C} \text{ geometric}$$

$$x = \left(1 + \frac{1}{C}\right)^B - 1 \quad \Rightarrow \text{set} = \left(1 + \frac{1}{C}\right)^B - 1$$

If $x < 1$, dual objective function increases by k
 primal $\Delta P = Bx + z_j = z_j + \frac{1}{C} + 1 - x = \underbrace{1 + \frac{1}{C}}_{\text{ratio}}$

for $B \gg 1$ $1 + \frac{1}{C} \approx \frac{C}{C-1}$ use weak-duality
 Then, algo is $1 + \frac{1}{C}$ competitive.

* We need to convert the fractional solⁿ to randomized competitive algorithm. We arrange the increments of x on the $(0,1]$ interval and choose $\alpha \in [0,1]$ (before) uniformly in random. We are going to buy skis on the day corresponding to the increment of x to which α belongs.

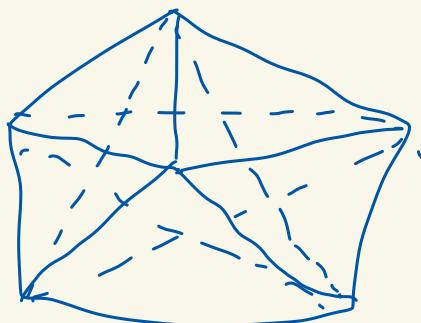
Prob. of buying skis on the j^{th} day = x_j
 Expected cost = $B \cdot \sum_{j=1}^k x_j = Bx$

$$z_j = 1 - \sum_{i=1}^{j-1} x_i > 1 - \underbrace{\sum_{i=1}^k x_i}_{\text{Prob. of not buying skis}}$$

Prob of getting is at most z_j .
 By linearity of expectation, for any numbers of ski days, the expected cost of the randomized algo is at most the cost of the fractional solution.

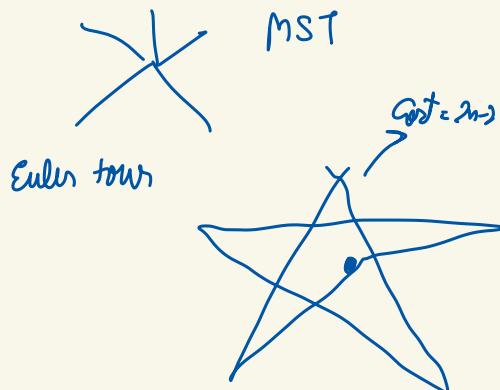
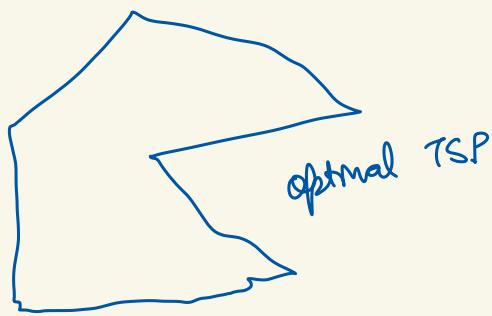
Tight Example of Eulerian Metric TSP Algo (2x MST)

Complete graph on n vertices with edges of cost 1 and 2



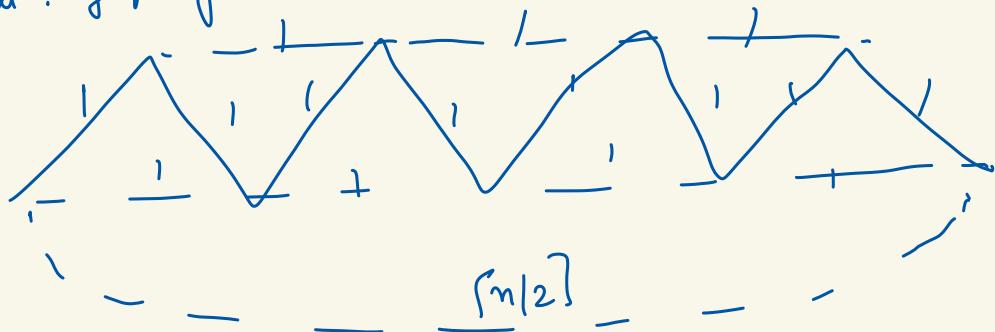
Thick edges \rightarrow cost = 1
Remaining \rightarrow cost = 2

For arbitrary n ,
graph has $2n-2$ edges of
cost 1, with these edges
forming union of a star,
and an $n-1$ cycle, all others
have cost 2.



Tight Example of Christofides Algorithm

n odd : graph of n vertices



MST = thick edge

$$\text{Christofides} = (n-1) + \lceil \frac{n}{2} \rceil$$

optimal = n

K-center clustering

- * find a set of k points, $C \subseteq P$, such that the maximum distance of a point in P to its closest point in C is minimized.

Greedy k centers

arbitrary point \bar{c}_1 served by $C_1 \rightarrow r_1 = \max_{p \in P} d_1(p)$ Add the point which is worst

$O(nk)$ time
 $\Theta(n)$ space

This algo is 2-approx to optimal k-center clustering.

$$r_k = \|p_k\|_\infty$$

$$\bar{c}_{k+1} = \text{point in } P \text{ realizing } r_k = \max_{p \in P} d(p, C) \quad C = k \cup \{\bar{c}_{k+1}\}$$

$$r_1 > \dots > r_k$$

$$d_m(\bar{c}_i, \bar{c}_j) > d_m(\bar{c}_j, c_{j-1}) = r_{j-1} \geq r_k$$

$$; c_j \leq k+1$$

distance between any pair of points in C is at least r_k .

assume $r_k > 2 \text{opt}_{\infty}(P, k)$

optimal solution that covers P with k balls of radius opt_{∞} .

By Δ -inq, any two points are within a distance of at most 2opt_{∞} .
thus none of these balls can cover two such points of $C \subseteq P$.
The optimal covering by k balls of radius opt_{∞} cannot cover C (and thus P) as $|C| = k+1 \rightarrow$ contradiction.

* Spread = Ratio between diameter and the distance of the closest pair of P .

ANN for bounded spread

* Let $A_0 = \{\text{root}(\mathcal{T})\}$ $r_{\text{curr}} = ||q - \text{rep}_{\text{root}}(T)||$
 In the i^{th} iteration, for $i > 0$, expand the nodes of A_{i-1} to get A_i
 for $v \in A_{i-1}$, $C_v = \text{set of children of } v \in \mathcal{T}$
 $\square_v = \text{denote the cell}$

for every node $w \in C_v$:
 $r_{\text{curr}} + \min(r_{\text{curr}}, ||q - \text{rep}_w||)$

$$\text{if } ||q - \text{rep}_w|| - \text{diam}(\square_w) < (1 - \frac{\epsilon}{2}) r_{\text{curr}}$$

stop when $A_i = \text{empty}$

* The algorithm adds a node w to A_i only if the set P_w might contain points which are closer to q than the (best) current nearest neighbour of the algorithm. Start node w inspected by the algo such that $\text{nm}(q) \in P_w$. Algorithm decided to throw it away:
 $||q - \text{nm}(q)|| \geq ||q - \text{rep}_w|| - ||\text{rep}_w - \text{nm}(q)||$
 $\geq ||q - \text{rep}_w|| - \text{diam}(\square_w) \geq (1 - \frac{\epsilon}{2}) r_{\text{curr}}$

$$\text{Thus } \frac{||q - \text{nm}(q)||}{1 - \frac{\epsilon}{2}} \geq r_{\text{curr}} \rightarrow \frac{1}{(1 - \frac{\epsilon}{2})} \leq 1 + \epsilon$$

$$\rightarrow r_{\text{curr}} \leq (1 + \epsilon) d(q, P)$$

Lemma: Let P = set of n points inside the unit hypercube in \mathbb{R}^d \mathcal{T} = quadtree of P , where $\text{diam}(P) = \Omega(1)$. q = query point, $\epsilon > 0$
 A $(1 + \epsilon)$ ANN to q can be computed in $O(\epsilon^{-d} + \log(\frac{1}{\epsilon}))$
 time, where $\bar{w} = d(q, P)$.

If a node $w \in \mathcal{T}$ is considered by the algo and $\text{diam}(\square_w) < (\frac{\epsilon}{n}) \bar{w}$ then

$$\|q - \text{ref}_w\| - \text{diam}(\square_w) \geq \|q - \text{ref}_w\| - \left(\frac{\varepsilon}{\bar{w}}\right)\bar{w} \geq \text{curr} - \left(\frac{\varepsilon}{\bar{w}}\right)\text{curr}$$

No nodes of depth $\geq h = \lceil -\log \left(\frac{\bar{w}\varepsilon}{\bar{w}} \right) \rceil$ are being considered. $\geq (1 - \frac{\varepsilon}{\bar{w}})\text{curr}$

node w of T of depth i containing $m(q, p)$ distance between q and ref_w is at most $l_i = \bar{w} + \text{diam}_w = \bar{w} + \sqrt{d} 2^i$
at the end of i^{th} iteration, $\text{curr} \leq l_i$

The number of relevant cells (i.e. cells that the algorithm deems and do not get immediately thrown out) is the number of grid cells of $G_{2^{-i-1}}$ that intersect a box of sidelength $2l_i$ centered at q i.e.

$$n_i = \left(2 \left\lceil \frac{l_i}{2^{-i-1}} \right\rceil \right)^d = O \left((1 + (2^i \bar{w}))^d \right)$$

$$(a+b)^d \leq (2 \max(a, b))^d \leq 2^d (a^d + b^d)$$

$$\sum_{i=0}^n n_i = O \left(\log \frac{1}{\bar{w}} + \frac{1}{\varepsilon^d} \right)$$

METRIC EMBEDDING

Let (X, d_X) and (Y, d_Y) be metric spaces. A mapping $f: X \rightarrow Y$ is an embedding if it is C -lipschitz if $d_Y(f(x), f(y)) \leq C \cdot d_X(x, y)$ for all $x, y \in X$. The mapping f is called κ -bi-lipschitz if there exists

a $C > 0$ such that :

$$C^{-1} d_X(x, y) \leq d_Y(f(x), f(y)) \leq C \cdot d_X(x, y)$$

$\forall x, y \in X.$

least $\kappa = \text{distortion of } f$

CHAPTER 4

Clustering – Definitions and Basic Algorithms

In this chapter, we will initiate our discussion of **clustering**. Clustering is one of the most fundamental computational tasks but, frustratingly, one of the fuzziest. It can be stated informally as: “Given data, find an interesting structure in the data. Go!”

The fuzziness arises naturally from the requirement that the clustering should be “interesting”, which is not a well-defined concept and depends on human perception and hence is impossible to quantify clearly. Similarly, the meaning of “structure” is also open to debate. Nevertheless, clustering is inherent to many computational tasks like learning, searching, and data-mining.

Empirical study of clustering concentrates on trying various measures for the clustering and trying out various algorithms and heuristics to compute these clusterings. See the bibliographical notes in this chapter for some relevant references.

Here we will concentrate on some well-defined clustering tasks, including k -center clustering, k -median clustering, and k -means clustering, and some basic algorithms for these problems.

4.1. Preliminaries

A clustering problem is usually defined by a set of items, and a distance function between the items in this set. While these items might be points in \mathbb{R}^d and the distance function just the regular Euclidean distance, it is sometimes beneficial to consider the more abstract setting of a general metric space.

4.1.1. Metric spaces.

Definition 4.1. A **metric space** is a pair $(\mathcal{X}, \mathbf{d})$ where \mathcal{X} is a set and $\mathbf{d} : \mathcal{X} \times \mathcal{X} \rightarrow [0, \infty)$ is a **metric** satisfying the following axioms: (i) $\mathbf{d}_M(x, y) = 0$ if and only if $x = y$, (ii) $\mathbf{d}_M(x, y) = \mathbf{d}_M(y, x)$, and (iii) $\mathbf{d}_M(x, y) + \mathbf{d}_M(y, z) \geq \mathbf{d}_M(x, z)$ (triangle inequality).

For example, \mathbb{R}^2 with the regular Euclidean distance is a metric space. In the following, we assume that we are given **black-box access** to \mathbf{d}_M . Namely, given two points $p, q \in \mathcal{X}$, we assume that $\mathbf{d}_M(p, q)$ can be computed in constant time.

Another standard example for a finite metric space is a graph G with non-negative weights $\omega(\cdot)$ defined on its edges. Let $\mathbf{d}_G(x, y)$ denote the shortest path (under the given weights) between any $x, y \in V(G)$. It is easy to verify that $\mathbf{d}_G(\cdot, \cdot)$ is a metric. In fact, any **finite metric** (i.e., a metric defined over a finite set) can be represented by such a weighted graph.

The **L_p -norm** defines the distance between two points $p, q \in \mathbb{R}^d$ as

$$\|p - q\|_p = \left(\sum_{i=1}^d |p_i - q_i|^p \right)^{1/p},$$

for $p \geq 1$. The **L_2 -norm** is the regular Euclidean distance.

The L_1 -norm, also known as the **Manhattan distance** or **taxicab distance**, is

$$\|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^d |\mathbf{p}_i - \mathbf{q}_i|.$$

The L_1 -norm distance between two points is the minimum path length that is axis parallel and connects the two points. For a uniform grid, it is the minimum number of grid edges (i.e., blocks in Manhattan) one has to travel between two grid points. In particular, the shortest path between two points is no longer unique; see the picture on the right. Of course, in the L_2 -norm the shortest path between two points is the segment connecting the two points.

The L_∞ -norm is

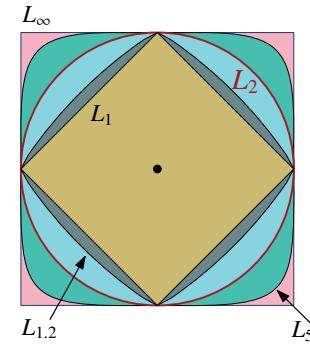
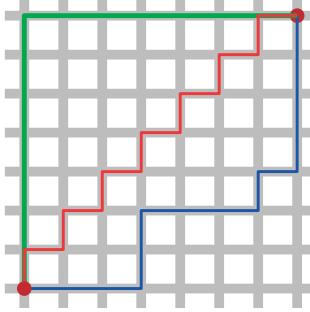
$$\|\mathbf{p} - \mathbf{q}\|_\infty = \lim_{p \rightarrow \infty} \|\mathbf{p} - \mathbf{q}\|_p = \max_{i=1}^d |\mathbf{p}_i - \mathbf{q}_i|.$$

The triangle inequality holds for the L_p -norm, for any $p \geq 1$ (it is called the **Minkowski inequality** in this case). In particular, L_p is a metric for $p \geq 1$. Specifically, \mathbb{R}^d with any L_p -norm (i.e., $p \geq 1$) is another example of a metric space.

It is useful to consider the different unit balls of L_p for different value of p ; see the figure on the right. The figure implies (and one can prove it formally) that for any point $\mathbf{p} \in \mathbb{R}^d$, we have that $\|\mathbf{p}\|_p \leq \|\mathbf{p}\|_q$ if $p > q$.

Lemma 4.2. For any $\mathbf{p} \in \mathbb{R}^d$, we have that $\|\mathbf{p}\|_1 / \sqrt{d} \leq \|\mathbf{p}\|_2 \leq \|\mathbf{p}\|_1$.

PROOF. Indeed, let $\mathbf{p} = (\mathbf{p}_1, \dots, \mathbf{p}_d)$, and assume that $\mathbf{p}_i \geq 0$, for all i . It is easy to verify that for a constant α , the function $f(x) = x^2 + (\alpha - x)^2$ is minimized when $x = \alpha/2$. As such, setting $\alpha = \|\mathbf{p}\|_1 = \sum_{i=1}^d |\mathbf{p}_i|$, we have, by symmetry and by the above observation on $f(x)$, that $\sum_{i=1}^d \mathbf{p}_i^2$ is minimized under the condition $\|\mathbf{p}\|_1 = \alpha$, when all the coordinates of \mathbf{p} are equal. As such, we have that $\|\mathbf{p}\|_2 \geq \sqrt{d(\alpha/d)^2} = \|\mathbf{p}\|_1 / \sqrt{d}$, implying the claim. ■



4.1.2. The clustering problem. There is a metric space $(\mathcal{X}, \mathbf{d})$ and the input is a set of n points $\mathbf{P} \subseteq \mathcal{X}$. Given a set of centers \mathbf{C} , every point of \mathbf{P} is assigned to its nearest neighbor in \mathbf{C} . All the points of \mathbf{P} that are assigned to a center $\bar{\mathbf{c}}$ form the **cluster** of $\bar{\mathbf{c}}$, denoted by

$$(4.1) \quad \text{cluster}(\mathbf{C}, \bar{\mathbf{c}}) = \left\{ \mathbf{p} \in \mathbf{P} \mid \mathbf{d}_{\mathcal{M}}(\mathbf{p}, \bar{\mathbf{c}}) = \mathbf{d}(\mathbf{p}, \mathbf{C}) \right\},$$

where

$$\mathbf{d}(\mathbf{p}, \mathbf{C}) = \min_{\bar{\mathbf{c}} \in \mathbf{C}} \mathbf{d}_{\mathcal{M}}(\mathbf{p}, \bar{\mathbf{c}})$$

denotes the **distance** of \mathbf{p} to the set \mathbf{C} . Namely, the center set \mathbf{C} partition \mathbf{P} into clusters. This specific scheme of partitioning points by assigning them to their closest center (in a given set of centers) is known as a **Voronoi partition**.

In particular, let $P = \{p_1, \dots, p_n\}$, and consider the n -dimensional point

$$P_C = (\mathbf{d}(p_1, C), \mathbf{d}(p_2, C), \dots, \mathbf{d}(p_n, C)).$$

The i th coordinate of the point P_C is the distance (i.e., cost of assigning) of p_i to its closest center in C .

4.2. On k -center clustering

In the k -center clustering problem, a set $P \subseteq \mathcal{X}$ of n points is provided together with a parameter k . We would like to find a set of k points, $C \subseteq P$, such that the maximum distance of a point in P to its closest point in C is minimized.

As a concrete example, consider the set of points to be a set of cities. Distances between points represent the time it takes to travel between the corresponding cities. We would like to build k hospitals and minimize the maximum time it takes a patient to arrive at her closest hospital. Naturally, we want to build the hospitals in the cities and not in the middle of nowhere.^①

Formally, given a set of centers C , the k -center clustering **price** of P by C is denoted by

$$\|P_C\|_\infty = \max_{p \in P} \mathbf{d}(p, C).$$

Note that every point in a cluster is within a distance at most $\|P_C\|_\infty$ from its respective center.

Formally, the **k -center problem** is to find a set C of k points, such that $\|P_C\|_\infty$ is minimized; namely,

$$\text{opt}_\infty(P, k) = \min_{C \subseteq P, |C|=k} \|P_C\|_\infty.$$

We will denote the set of centers realizing the optimal clustering by C_{opt} . A more explicit definition (and somewhat more confusing) of the k -center clustering is to compute the set C of size k realizing $\min_{C \subseteq P} \max_{p \in P} \min_{c \in C} \mathbf{d}_M(p, c)$.

It is known that k -center clustering is **NP-HARD**, and it is in fact hard to approximate within a factor of 1.86, even for a point set in the plane (under the Euclidean distance). Surprisingly, there is a simple and elegant algorithm that achieves a 2-approximation.

Discrete vs. continuous clustering. If the input is a point set in \mathbb{R}^d , the centers of the clustering are not necessarily restricted to be a subset of the input point, as they might be placed anywhere in \mathbb{R}^d . Allowing this flexibility might further reduce the price of the clustering (by a constant factor). The variant where one is restricted to use the input points as centers is the **discrete clustering** problem. The version where centers might be placed anywhere in the given metric space is the **continuous clustering** version.

4.2.1. The greedy clustering algorithm. The algorithm **GreedyKCenter** starts by picking an arbitrary point, \bar{c}_1 , and setting $C_1 = \{\bar{c}_1\}$. Next, we compute for every point $p \in P$ its distance $d_1[p]$ from \bar{c}_1 . Now, consider the point worst served by C_1 ; this is the point realizing $r_1 = \max_{p \in P} d_1[p]$. Let \bar{c}_2 denote this point, and add it to the set C_1 , resulting in the set C_2 .

Specifically, in the i th iteration, we compute for each point $p \in P$ the quantity $d_{i-1}[p] = \min_{\bar{c} \in C_{i-1}} \mathbf{d}_M(p, \bar{c})$. We also compute the radius of the clustering

$$(4.2) \quad r_{i-1} = \|P_{C_{i-1}}\|_\infty = \max_{p \in P} d_{i-1}[p] = \max_{p \in P} \mathbf{d}(p, C_{i-1})$$

^①Although, there are recorded cases in history of building universities in the middle of nowhere.

and the bottleneck point \bar{c}_i that realizes it. Next, we add \bar{c}_i to \mathbf{C}_{i-1} to form the new set \mathbf{C}_i . We repeat this process k times.

Namely, the algorithm repeatedly picks the point furthest away from the current set of centers and adds it to this set.

To make this algorithm slightly faster, observe that

$$d_i[\mathbf{p}] = \mathbf{d}(\mathbf{p}, \mathbf{C}_i) = \min(\mathbf{d}(\mathbf{p}, \mathbf{C}_{i-1}), \mathbf{d}_{\mathcal{M}}(\mathbf{p}, \bar{c}_i)) = \min(d_{i-1}[\mathbf{p}], \mathbf{d}_{\mathcal{M}}(\mathbf{p}, \bar{c}_i)).$$

In particular, if we maintain for each point $\mathbf{p} \in \mathbf{P}$ a single variable $d[\mathbf{p}]$ with its current distance to its closest center in the current center set, then the above formula boils down to

$$d[\mathbf{p}] \leftarrow \min(d[\mathbf{p}], \mathbf{d}_{\mathcal{M}}(\mathbf{p}, \bar{c}_i)).$$

Namely, the above algorithm can be implemented using $O(n)$ space, where $n = |\mathbf{P}|$. The i th iteration of choosing the i th center takes $O(n)$ time. Thus, overall, this approximation algorithm takes $O(nk)$ time.

A **ball** of radius r around a point $\mathbf{p} \in \mathbf{P}$ is the set of points in \mathbf{P} with distance at most r from \mathbf{p} ; namely, $\mathbf{b}(\mathbf{p}, r) = \{\mathbf{q} \in \mathbf{P} \mid \mathbf{d}_{\mathcal{M}}(\mathbf{p}, \mathbf{q}) \leq r\}$. Thus, the k -center problem can be interpreted as the problem of covering the points of \mathbf{P} using k balls of minimum (maximum) radius.

Theorem 4.3. *Given a set of n points \mathbf{P} in a metric space $(\mathcal{X}, \mathbf{d})$, the algorithm **GreedyK-Center** computes a set \mathbf{K} of k centers, such that \mathbf{K} is a 2-approximation to the optimal k -center clustering of \mathbf{P} ; namely, $\|\mathbf{P}_{\mathbf{K}}\|_{\infty} \leq 2\text{opt}_{\infty}$, where $\text{opt}_{\infty} = \text{opt}_{\infty}(\mathbf{P}, k)$ is the price of the optimal clustering. The algorithm takes $O(nk)$ time.*

PROOF. The running time follows by the above description, so we concern ourselves only with the approximation quality.

By definition, we have $r_k = \|\mathbf{P}_{\mathbf{K}}\|_{\infty}$, and let \bar{c}_{k+1} be the point in \mathbf{P} realizing $r_k = \max_{\mathbf{p} \in \mathbf{P}} \mathbf{d}(\mathbf{p}, \mathbf{K})$. Let $\mathbf{C} = \mathbf{K} \cup \{\bar{c}_{k+1}\}$. Observe that by the definition of r_i (see (4.2)), we have that $r_1 \geq r_2 \geq \dots \geq r_k$. Furthermore, for $i < j \leq k + 1$ we have that

$$\mathbf{d}_{\mathcal{M}}(\bar{c}_i, \bar{c}_j) \geq \mathbf{d}_{\mathcal{M}}(\bar{c}_j, \mathbf{C}_{j-1}) = r_{j-1} \geq r_k.$$

Namely, the distance between any pair of points in \mathbf{C} is at least r_k . Now, assume for the sake of contradiction that $r_k > 2\text{opt}_{\infty}(\mathbf{P}, k)$. Consider the optimal solution that covers \mathbf{P} with k balls of radius opt_{∞} . By the triangle inequality, any two points inside such a ball are within a distance at most 2opt_{∞} from each other. Thus, none of these balls can cover two points of $\mathbf{C} \subseteq \mathbf{P}$, since the minimum distance between members of \mathbf{C} is $> 2\text{opt}_{\infty}$. As such, the optimal cover by k balls of radius opt_{∞} cannot cover \mathbf{C} (and thus \mathbf{P}), as $|\mathbf{C}| = k + 1$, a contradiction. ■

In the spirit of never trusting a claim that has only a single proof, we provide an alternative proof.^②

ALTERNATIVE PROOF. If every cluster of C_{opt} contains exactly one point of \mathbf{K} , then the claim follows. Indeed, consider any point $\mathbf{p} \in \mathbf{P}$, and let \bar{c} be the center it belongs to in C_{opt} . Also, let \bar{g} be the center of \mathbf{K} that is in cluster(C_{opt}, \bar{c}). We have that $\mathbf{d}_{\mathcal{M}}(\mathbf{p}, \bar{c}) = \mathbf{d}(\mathbf{p}, C_{\text{opt}}) \leq \text{opt}_{\infty} = \text{opt}_{\infty}(\mathbf{P}, k)$. Similarly, observe that $\mathbf{d}_{\mathcal{M}}(\bar{g}, \bar{c}) = \mathbf{d}(\bar{g}, C_{\text{opt}}) \leq \text{opt}_{\infty}$. As such, by the triangle inequality, we have that $\mathbf{d}_{\mathcal{M}}(\mathbf{p}, \bar{g}) \leq \mathbf{d}_{\mathcal{M}}(\mathbf{p}, \bar{c}) + \mathbf{d}_{\mathcal{M}}(\bar{c}, \bar{g}) \leq 2\text{opt}_{\infty}$.

^②Mark Twain is credited with saying that “I don’t give a damn for a man that can only spell a word one way.” However, there seems to be some doubt if he really said that, which brings us to the conclusion of never trusting a quote if it is credited only to a single person.

By the pigeon hole principle, the only other possibility is that there are at least two centers \bar{g} and \bar{h} of \mathbf{K} that are both in cluster $(C_{\text{opt}}, \bar{c})$, for some $\bar{c} \in C_{\text{opt}}$. Assume, without loss of generality, that \bar{h} was added later than \bar{g} to the center set \mathbf{K} by the algorithm **GreedyKCenter**, say in the i th iteration. But then, since **GreedyKCenter** always chooses the point furthest away from the current set of centers, we have that

$$\|\mathbf{P}_{\mathbf{K}}\|_{\infty} \leq \|\mathbf{P}_{\mathbf{C}_{i-1}}\|_{\infty} = \mathbf{d}(\bar{h}, \mathbf{C}_{i-1}) \leq \mathbf{d}_{\mathcal{M}}(\bar{h}, \bar{g}) \leq \mathbf{d}_{\mathcal{M}}(\bar{h}, \bar{c}) + \mathbf{d}_{\mathcal{M}}(\bar{c}, \bar{g}) \leq 2\text{opt}_{\infty}. \blacksquare$$

4.2.2. The greedy permutation. There is an interesting phenomena associated with **GreedyKCenter**. If we run it till it exhausts all the points of \mathbf{P} (i.e., $k = n$), then this algorithm generates a permutation of \mathbf{P} ; that is, $\langle \mathbf{P} \rangle = \langle \bar{c}_1, \bar{c}_2, \dots, \bar{c}_n \rangle$. We will refer to $\langle \mathbf{P} \rangle$ as the **greedy permutation** of \mathbf{P} . There is also an associated sequence of radii $\langle r_1, r_2, \dots, r_n \rangle$, where all the points of \mathbf{P} are within a distance at most r_i from the points of $\mathbf{C}_i = \langle \bar{c}_1, \dots, \bar{c}_i \rangle$.

Definition 4.4. A set $S \subseteq \mathbf{P}$ is an r -packing for \mathbf{P} if the following two properties hold.

- (i) **Covering property:** All the points of \mathbf{P} are within a distance at most r from the points of S .
 - (ii) **Separation property:** For any pair of points $p, q \in S$, we have that $\mathbf{d}_{\mathcal{M}}(p, q) \geq r$.
- (One can relax the separation property by requiring that the points of S be at a distance $\Omega(r)$ apart.)

Intuitively, an r -packing of a point set \mathbf{P} is a compact representation of \mathbf{P} in the resolution r . Surprisingly, the greedy permutation of \mathbf{P} provides us with such a representation for all resolutions.

Theorem 4.5. Let \mathbf{P} be a set of n points in a finite metric space, and let its greedy permutation be $\langle \bar{c}_1, \bar{c}_2, \dots, \bar{c}_n \rangle$ with the associated sequence of radii $\langle r_1, r_2, \dots, r_n \rangle$. For any i , we have that $\mathbf{C}_i = \langle \bar{c}_1, \dots, \bar{c}_i \rangle$ is an r_i -packing of \mathbf{P} .

PROOF. Note that by construction $r_{k-1} = \mathbf{d}(\bar{c}_k, \mathbf{C}_{k-1})$, for all $k = 2, \dots, n$. As such, for $j < k \leq i \leq n$, we have that $\mathbf{d}_{\mathcal{M}}(\bar{c}_j, \bar{c}_k) \geq \mathbf{d}_{\mathcal{M}}(\bar{c}_k, \mathbf{C}_{k-1}) = r_{k-1} \geq r_i$, since r_1, r_2, \dots, r_n is a monotonically non-increasing sequence. This implies the required separation property.

The covering property follows by definition; see (4.2)_{p49}. \blacksquare

4.3. On k -median clustering

In the **k -median clustering problem**, a set $\mathbf{P} \subseteq \mathcal{X}$ is provided together with a parameter k . We would like to find a set of k points, $\mathbf{C} \subseteq \mathbf{P}$, such that the sum of the distances of points of \mathbf{P} to their closest point in \mathbf{C} is minimized.

Formally, given a set of centers \mathbf{C} , the k -median clustering **price** of clustering \mathbf{P} by \mathbf{C} is denoted by

$$\|\mathbf{P}_{\mathbf{C}}\|_1 = \sum_{p \in \mathbf{P}} \mathbf{d}(p, \mathbf{C}).$$

Formally, the **k -median problem** is to find a set \mathbf{C} of k points, such that $\|\mathbf{P}_{\mathbf{C}}\|_1$ is minimized; namely,

$$\text{opt}_1(\mathbf{P}, k) = \min_{\mathbf{C} \subseteq \mathbf{P}, |\mathbf{C}|=k} \|\mathbf{P}_{\mathbf{C}}\|_1.$$

We will denote the set of centers realizing the optimal clustering by C_{opt} .

There is a simple and elegant constant factor approximation algorithm for k -median clustering using **local search** (its analysis however is painful).

A note on notation. Consider the set $U = \left\{ P_C \mid C \in \mathbb{P}^k \right\}$. Clearly, we have that $\text{opt}_\infty(P, k) = \min_{q \in U} \|q\|_\infty$ and $\text{opt}_1(P, k) = \min_{q \in U} \|q\|_1$.

Namely, k -center clustering under this interpretation is just finding the point minimizing the L_∞ -norm in a set U of points in n dimensions. Similarly, the k -median problem is to find the point minimizing the L_1 -norm in the set U .

Claim 4.6. *For any point set P of n points and a parameter k , we have that $\text{opt}_\infty(P, k) \leq \text{opt}_1(P, k) \leq n \cdot \text{opt}_\infty(P, k)$.*

PROOF. For any point $p \in \mathbb{R}^n$, we have that $\|p\|_\infty = \max_{i=1}^n |p_i| \leq \sum_{i=1}^n |p_i| = \|p\|_1$ and $\|p\|_1 = \sum_{i=1}^n |p_i| \leq \sum_{i=1}^n \max_{j=1}^n |p_j| \leq n \|p\|_\infty$.

Let C be the set of k points realizing $\text{opt}_1(P, k)$; that is, $\text{opt}_1(P, k) = \|P_C\|_1$. We have that $\text{opt}_\infty(P, k) \leq \|P_C\|_\infty \leq \|P_C\|_1 = \text{opt}_1(P, k)$. Similarly, if K is the set realizing $\text{opt}_\infty(P, k)$, then $\text{opt}_1(P, k) = \|P_K\|_1 \leq \|P_K\|_\infty \leq n \|P_K\|_\infty = n \cdot \text{opt}_\infty(P, k)$. ■

4.3.1. Approximation algorithm – local search. We are given a set P of n points and a parameter k . In the following, let C_{opt} denote the set of centers realizing the optimal solution, and let $\text{opt}_1 = \text{opt}_1(P, k)$.

4.3.1.1. The algorithm.

A $2n$ -approximation. The algorithm starts by computing a set of k centers L using Theorem 4.3. Claim 4.6 implies that

$$(4.3) \quad \begin{aligned} \|P_L\|_1 / 2n &\leq \|P_L\|_\infty / 2 \leq \text{opt}_\infty(P, k) \leq \text{opt}_1 \leq \|P_L\|_1 \\ &\implies \text{opt}_1 \leq \|P_L\|_1 \leq 2n \text{opt}_1. \end{aligned}$$

Namely, L is a $2n$ -approximation to the optimal solution.

Improving it. Let $0 < \tau < 1$ be a parameter to be determined shortly. The local search algorithm **algLocalSearchKMed** initially sets the current set of centers L_{curr} to be L , the set of centers computed above. Next, at each iteration it checks if the current solution L_{curr} can be improved by replacing one of the centers in it by a center from the outside. We will refer to such an operation as a *swap*. There are at most $|P| |L_{\text{curr}}| = nk$ choices to consider, as we pick a center $\bar{c} \in L_{\text{curr}}$ to throw away and a new center to replace it by $\bar{o} \in (P \setminus L_{\text{curr}})$. We consider the new candidate set of centers $K \leftarrow (L_{\text{curr}} \setminus \{\bar{c}\}) \cup \{\bar{o}\}$. If $\|P_K\|_1 \leq (1 - \tau) \|P_{L_{\text{curr}}}\|_1$, then the algorithm sets $L_{\text{curr}} \leftarrow K$. The algorithm continues iterating in this fashion over all possible swaps.

The algorithm **algLocalSearchKMed** stops when there is no swap that would improve the current solution by a factor of (at least) $(1 - \tau)$. The final content of the set L_{curr} is the required constant factor approximation.

4.3.1.2. Running time. An iteration requires checking $O(nk)$ swaps (i.e., $n - k$ candidates to be swapped in and k candidates to be swapped out). Computing the price of every such swap, done naively, requires computing the distance of every point to its nearest center, and that takes $O(nk)$ time per swap. As such, overall, each iteration takes $O((nk)^2)$ time.

Since $1/(1 - \tau) \geq 1 + \tau$, the running time of the algorithm is

$$O\left((nk)^2 \log_{1/(1-\tau)} \frac{\|P_L\|_1}{\text{opt}_1}\right) = O\left((nk)^2 \log_{1+\tau} 2n\right) = O\left((nk)^2 \frac{\log n}{\tau}\right),$$

by (4.3) and Lemma 28.10_{p348}. Thus, if τ is polynomially small, then the running time would be polynomial.

4.3.2. Proof of quality of approximation. We claim that the above algorithm provides a constant factor approximation for the optimal k -median clustering.

4.3.2.1. *Definitions and intuition.* Intuitively, since the local search got stuck in a locally optimal solution, it cannot be too far from the true optimal solution.

For the sake of simplicity of exposition, let us assume (for now) that the solution returned by the algorithm cannot be improved (at all) by any swap, and let L be this set of centers. For a center $\bar{c} \in L$ and a point $\bar{o} \in P \setminus L$, let $L - \bar{c} + \bar{o} = (L \setminus \{\bar{c}\}) \cup \{\bar{o}\}$ denote the set of centers resulting from applying the swap $\bar{c} \rightarrow \bar{o}$ to L . We are assuming that there is no beneficial swap; that is,

$$(4.4) \quad \forall \bar{c} \in L, \bar{o} \in P \setminus L \quad 0 \leq \Delta(\bar{c}, \bar{o}) = \nu_1(L - \bar{c} + \bar{o}) - \nu_1(L),$$

where $\nu_1(X) = \|P_X\|_1$.

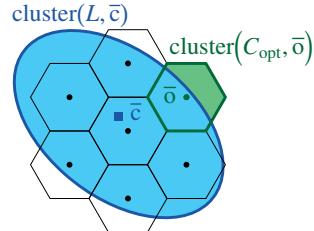
Equation (4.4) provides us with a large family of inequalities that all hold together. Each inequality is represented by a swap $\bar{c} \rightarrow \bar{o}$. We would like to combine these inequalities such that they will imply that $5\|P_{C_{\text{opt}}}\|_1 \geq \|P_L\|_1$, namely, that the local search algorithm provides a constant factor approximation to optimal clustering. This idea seems to be somewhat mysterious (or even impossible), but hopefully it will become clearer shortly.

From local clustering to local clustering complying with the optimal clustering.

The first hurdle in the analysis is that a cluster of the optimal solution cluster $(C_{\text{opt}}, \bar{o})$, for $\bar{o} \in C_{\text{opt}}$, might intersect a large number of clusters in the local clustering (i.e., clusters of the form $\text{cluster}(L, \bar{c})$ for $\bar{c} \in L$).

Fortunately, one can modify the assignment of points to clusters in the locally optimal clustering so that the resulting clustering of P complies with the optimal partition and the price of the clustering increases only moderately; that is, every cluster in the optimal clustering would be contained in a single cluster of the modified local solution. In particular, now an optimal cluster would intersect only a single cluster in the modified local solution.

Furthermore, this modified local solution Π is not much more expensive. Now, in this modified partition there are many beneficial swaps (by making it into the optimal clustering). But these swaps cannot be too profitable, since then they would have been profitable for the original local solution. This would imply that the local solution cannot be too expensive. The picture on the right depicts a local cluster and the optimal clusters in its vicinity such that their centers are contained inside it.

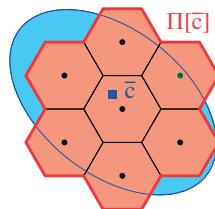


In the following, we denote by $\text{nn}(p, X)$ the nearest neighbor to p in the set X .

For a point $p \in P$, let $\bar{o}_p = \text{nn}(p, C_{\text{opt}})$ be its optimal center, and let $\alpha(p) = \text{nn}(\bar{o}_p, L)$ be the center p should use if p follows its optimal center's assignment. Let Π be the modified partition of P by the function $\alpha(\cdot)$.

That is, for $\bar{c} \in L$, its cluster in Π , denoted by $\Pi[\bar{c}]$, is the set of all points $p \in P$ such that $\alpha(p) = \bar{c}$.

Now, for any center $\bar{o} \in C_{\text{opt}}$, let $\text{nn}(\bar{o}, L)$ be its nearest neighbor in L , and observe that $\text{cluster}(C_{\text{opt}}, \bar{o}) \subseteq \Pi[\text{nn}(\bar{o}, L)]$ (see (4.1)_{p48}). The picture on the right shows the resulting modified cluster for the above example.



Let δ_p denote the price of this reassignment for the point p ; that is, $\delta_p = \mathbf{d}_M(p, \alpha(p)) - \mathbf{d}(p, L)$. Note that if p does not get reassigned, then $\delta_p = 0$ and otherwise $\delta_p \geq 0$, since $\alpha(p) \in L$ and $\mathbf{d}(p, L) = \min_{\bar{c} \in L} \mathbf{d}_M(p, \bar{c})$.

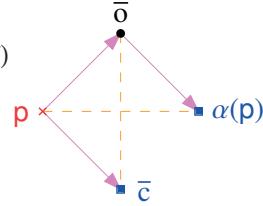
Lemma 4.7. *The increase in cost from moving from the clustering induced by L to the clustering of Π is bounded by $2 \|\mathbf{P}_{C_{\text{opt}}}\|_1$. That is, $\sum_{p \in P} \delta_p \leq 2 \|\mathbf{P}_{C_{\text{opt}}}\|_1$.*

PROOF. For a point $p \in P$, let $\bar{c} = \text{nn}(p, L)$ be its local center, let $\bar{o} = \text{nn}(p, C_{\text{opt}})$ be its optimal center, and let $\alpha(p) = \text{nn}(\bar{o}, L)$ be its new assigned center in Π . Observe that $\mathbf{d}_M(\bar{o}, \alpha(p)) = \mathbf{d}_M(\bar{o}, \text{nn}(\bar{o}, L)) \leq \mathbf{d}_M(\bar{o}, \bar{c})$.

As such, by the triangle inequality, we have that

$$\begin{aligned} \mathbf{d}_M(p, \alpha(p)) &\leq \mathbf{d}_M(p, \bar{o}) + \mathbf{d}_M(\bar{o}, \alpha(p)) \leq \mathbf{d}_M(p, \bar{o}) + \mathbf{d}_M(\bar{o}, \bar{c}) \\ &\leq \mathbf{d}_M(p, \bar{o}) + (\mathbf{d}_M(\bar{o}, p) + \mathbf{d}_M(p, \bar{c})) \\ &= 2\mathbf{d}_M(p, \bar{o}) + \mathbf{d}_M(p, \bar{c}). \end{aligned}$$

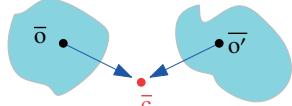
Finally, $\delta_p = \mathbf{d}_M(p, \alpha(p)) - \mathbf{d}(p, L) \leq 2\mathbf{d}_M(p, \bar{o}) + \mathbf{d}_M(p, \bar{c}) - \mathbf{d}_M(p, \bar{c}) = 2\mathbf{d}_M(p, \bar{o}) = 2\mathbf{d}(p, C_{\text{opt}})$. As such, $\sum_{p \in P} \delta_p \leq \sum_{p \in P} 2\mathbf{d}(p, C_{\text{opt}}) = 2 \|\mathbf{P}_{C_{\text{opt}}}\|_1$. ■



Drifters, anchors, and tyrants. A center of L that does not serve any center of C_{opt} (i.e., its cluster in Π is empty) is a *drifter*. Formally, we map each center of C_{opt} to its nearest neighbor in L , and for a center $\bar{c} \in L$ its *degree*, denoted by $\deg(\bar{c})$, is the number of points of C_{opt} mapped to it by this nearest neighbor mapping.

As such, a center $\bar{c} \in L$ is a *drifter* if $\deg(\bar{c}) = 0$, an *anchor* if $\deg(\bar{c}) = 1$, and a *tyrant* if $\deg(\bar{c}) > 1$. Observe that if \bar{c} is a drifter, then $\Pi[\bar{c}] = \emptyset$.

The reader should not take these names too seriously, but observe that centers that are tyrants cannot easily move around and are bad candidates for swaps. Indeed, consider the situation depicted in the figure on the right. Here the center \bar{c} serves points of P that belong to two optimal clusters \bar{o} and \bar{o}' , such that $\bar{c} = \text{nn}(\bar{o}, L) = \text{nn}(\bar{o}', L)$. If we swap $\bar{c} \rightarrow \bar{o}$, then the points in the cluster $\text{cluster}(C_{\text{opt}}, \bar{o}')$ might find themselves very far from any center in $L - \bar{c} + \bar{o}$. Similarly, the points of $\text{cluster}(C_{\text{opt}}, \bar{o})$ might be in trouble if we swap $\bar{c} \rightarrow \bar{o}'$.



Intuitively, since we shifted our thinking from the local solution to the partition Π , a drifter center is not being used by the clustering, and we can reassign it so that it decreases the price of the clustering.

That is, since moving from the local clustering of L to Π is relatively cheap, we can free a drifter \bar{c} from all its clients in the local partition. Formally, the *ransom* of a drifter center \bar{c} is $\text{ransom}(\bar{c}) = \sum_{p \in \text{cluster}(L, \bar{c})} \delta_p$. This is the price of reassigning all the points that are currently served by the drifter \bar{c} to the center in L serving their optimal center. Once this ransom is paid, \bar{c} serves nobody and can be moved with no further charge.

More generally, the *ransom* of any center $\bar{c} \in L$ is

$$\text{ransom}(\bar{c}) = \sum_{p \in \text{cluster}(L, \bar{c}) \setminus \Pi[\bar{c}]} \delta_p.$$

Note that for a drifter $\Pi[\bar{c}] = \emptyset$ and $\text{cluster}(L, \bar{c}) = \text{cluster}(L, \bar{c}) \setminus \Pi[\bar{c}]$, and in general, the points of $\text{cluster}(L, \bar{c}) \setminus \Pi[\bar{c}]$ are exactly the points of $\text{cluster}(L, \bar{c})$ being reassigned.

Hence, $\text{ransom}(\bar{c})$ is the increase in cost of reassigning the points of cluster (L, \bar{c}) when moving from the local clustering of L to the clustering of Π .

Observe that, by Lemma 4.7, we have that

$$(4.5) \quad \sum_{\bar{o} \in L} \text{ransom}(\bar{o}) \leq 2 \|\mathbf{P}_{C_{\text{opt}}}\|_1.$$

For $\bar{o} \in C_{\text{opt}}$, the **optimal price** and **local price** of cluster $(C_{\text{opt}}, \bar{o})$ are

$$\text{opt}(\bar{o}) = \sum_{p \in \text{cluster}(C_{\text{opt}}, \bar{o})} \mathbf{d}(p, C_{\text{opt}}) \quad \text{and} \quad \text{local}(\bar{o}) = \sum_{p \in \text{cluster}(C_{\text{opt}}, \bar{o})} \mathbf{d}(p, L),$$

respectively.

Lemma 4.8. *If $\bar{c} \in L$ is a drifter and \bar{o} is any center of C_{opt} , then $\text{local}(\bar{o}) \leq \text{ransom}(\bar{c}) + \text{opt}(\bar{o})$.*

PROOF. Since $\bar{c} \in L$ is a drifter, we can swap it with any center in $\bar{o} \in C_{\text{opt}}$. Since L is a locally optimal solution, we have that the change in the cost caused by the swap $\bar{c} \rightarrow \bar{o}$ is

$$(4.6) \quad \begin{aligned} 0 &\leq \Delta(\bar{c}, \bar{o}) \leq \text{ransom}(\bar{c}) - \text{local}(\bar{o}) + \text{opt}(\bar{o}) \\ &\implies \text{local}(\bar{o}) \leq \text{ransom}(\bar{c}) + \text{opt}(\bar{o}). \end{aligned}$$

Indeed, \bar{c} pays its ransom so that all the clients using it are now assigned to some other centers of L . Now, all the points of cluster $(C_{\text{opt}}, \bar{o})$ instead of paying $\text{local}(\bar{o})$ are now paying (at most) $\text{opt}(\bar{o})$. (We might pay less for a point $p \in \text{cluster}(C_{\text{opt}}, \bar{o})$ if it is closer to $L - \bar{c} + \bar{o}$ than to \bar{o}). ■

Equation (4.6) provides us with a glimmer of hope that we can bound the price of the local clustering. We next argue that if there are many tyrants, then there must also be many drifters. In particular, with these drifters we can bound the price of the local clustering cost of the optimal clusters assigned to tyrants. Also, we argue that an anchor and its associated optimal center define a natural swap which is relatively cheap. Putting all of these together will imply the desired claim.

There are many drifters. Let S_{opt} (resp. A_{opt}) be the set of all the centers of C_{opt} that are assigned to tyrants (resp. anchors) by $\text{nm}(\cdot, L)$. Observe that $S_{\text{opt}} \cup A_{\text{opt}} = C_{\text{opt}}$. Let \mathcal{D} be the set of drifters in L .

Observe that every tyrant has at least two followers in C_{opt} ; that is, $|S_{\text{opt}}| \geq 2\#\text{tyrants}$. Also, $k = |C_{\text{opt}}| = |L|$ and $\#\text{anchors} = |A_{\text{opt}}|$. As such, we have that

$$(4.7) \quad \begin{aligned} \#\text{tyrants} + \#\text{anchors} + \#\text{drifters} &= |L| = |C_{\text{opt}}| = |S_{\text{opt}}| + |A_{\text{opt}}| \\ \implies \#\text{drifters} &= |S_{\text{opt}}| + |A_{\text{opt}}| - \#\text{anchors} - \#\text{tyrants} = |S_{\text{opt}}| - \#\text{tyrants} \geq |S_{\text{opt}}| / 2. \end{aligned}$$

Namely, $2\#\text{drifters} \geq |S_{\text{opt}}|$.

Lemma 4.9. *We have that $\sum_{\bar{o} \in S_{\text{opt}}} \text{local}(\bar{o}) \leq 2 \sum_{\bar{c} \in \mathcal{D}} \text{ransom}(\bar{c}) + \sum_{\bar{o} \in S_{\text{opt}}} \text{opt}(\bar{o})$.*

PROOF. If $|S_{\text{opt}}| = 0$, then the statement holds trivially.

So assume $|S_{\text{opt}}| > 0$ and let \bar{c} be the drifter with the lowest $\text{ransom}(\bar{c})$. For any $\bar{o} \in S_{\text{opt}}$, we have that $\text{local}(\bar{o}) \leq \text{ransom}(\bar{c}) + \text{opt}(\bar{o})$, by (4.6). Summing over all such

centers, we have that

$$\sum_{\bar{o} \in S_{\text{opt}}} \text{local}(\bar{o}) \leq |S_{\text{opt}}| \text{ransom}(\bar{c}) + \sum_{\bar{o} \in S_{\text{opt}}} \text{opt}(\bar{o}),$$

which is definitely smaller than the stated bound, since $|S_{\text{opt}}| \leq 2|\mathcal{D}|$, by (4.7). \blacksquare

Lemma 4.10. *We have that $\sum_{\bar{o} \in A_{\text{opt}}} \text{local}(\bar{o}) \leq \sum_{\bar{o} \in A_{\text{opt}}} \text{ransom}(\text{nn}(\bar{o}, L)) + \sum_{\bar{o} \in A_{\text{opt}}} \text{opt}(\bar{o})$.*

PROOF. For a center $\bar{o} \in A_{\text{opt}}$, its anchor is $\bar{c} = \text{nn}(\bar{o}, L)$. Consider the swap $\bar{c} \rightarrow \bar{o}$, and the increase in clustering cost as we move from L to $L - \bar{c} + \bar{o}$.

We claim that $\text{local}(\bar{o}) \leq \text{ransom}(\bar{c}) + \text{opt}(\bar{o})$ (i.e., (4.6)) holds in this setting. The points for which their clustering is negatively affected (i.e., their clustering price might increase) by the swap are in the set $\text{cluster}(L, \bar{c}) \cup \text{cluster}(C_{\text{opt}}, \bar{o})$, and we split this set into two disjoint sets $X = \text{cluster}(L, \bar{c}) \setminus \text{cluster}(C_{\text{opt}}, \bar{o})$ and $Y = \text{cluster}(C_{\text{opt}}, \bar{o})$.

The increase in price by reassigning the points of X to some other center in L is exactly the ransom of \bar{c} . Now, the points of Y might get reassigned to \bar{o} , and the change in price of the points of Y can now be bounded by $-\text{local}(\bar{o}) + \text{opt}(\bar{o})$, as was argued in the proof of Lemma 4.8.

Note that it might be that points outside $X \cup Y$ get reassigned to \bar{o} in the clustering induced by $L - \bar{c} + \bar{o}$. However, such reassignment only further reduce the price of the swap. As such, we have that $0 \leq \Delta(\bar{c}, \bar{o}) \leq \text{ransom}(\bar{c}) - \text{local}(\bar{o}) + \text{opt}(\bar{o})$. As such, summing up the inequality $\text{local}(\bar{o}) \leq \text{ransom}(\bar{c}) + \text{opt}(\bar{o})$ over all the centers in A_{opt} implies the claim. \blacksquare

Lemma 4.11. *Let L be the set of k centers computed by the local search algorithm. We have that $\|\mathbf{P}_L\|_1 \leq 5\text{opt}_1(\mathbf{P}, k)$.*

PROOF. From the above two lemmas, we have that

$$\begin{aligned} \|\mathbf{P}_L\|_1 &= \sum_{\bar{o} \in C_{\text{opt}}} \text{local}(\bar{o}) = \sum_{\bar{o} \in S_{\text{opt}}} \text{local}(\bar{o}) + \sum_{\bar{o} \in A_{\text{opt}}} \text{local}(\bar{o}) \\ &\leq 2 \sum_{\bar{c} \in \mathcal{D}} \text{ransom}(\bar{c}) + \sum_{\bar{o} \in S_{\text{opt}}} \text{opt}(\bar{o}) + \sum_{\bar{o} \in A_{\text{opt}}} \text{ransom}(\text{nn}(\bar{o}, L)) + \sum_{\bar{o} \in A_{\text{opt}}} \text{opt}(\bar{o}) \\ &\leq 2 \sum_{\bar{c} \in L} \text{ransom}(\bar{c}) + \sum_{\bar{o} \in C_{\text{opt}}} \text{opt}(\bar{o}) \leq 4 \|\mathbf{P}_{C_{\text{opt}}}\|_1 + \|\mathbf{P}_{C_{\text{opt}}}\|_1 = 5\text{opt}_1(\mathbf{P}, k), \end{aligned}$$

by (4.5). \blacksquare

4.3.2.2. *Removing the strict improvement assumption.* In the above proof, we assumed that the current local minimum cannot be improved by a swap. Of course, this might not hold for the **algLocalSearchKMed** solution, since the algorithm allows a swap only if it makes “significant” progress. In particular, (4.4) is in fact

$$(4.8) \quad \forall \bar{c} \in L, \bar{o} \in \mathbf{P} \setminus L, \quad -\tau \|\mathbf{P}_L\|_1 \leq \|\mathbf{P}_{L - \bar{c} + \bar{o}}\|_1 - \|\mathbf{P}_L\|_1.$$

To adapt the proof to use this modified inequality, observe that the proof worked by adding up k inequalities defined by (4.4) and getting the inequality $0 \leq 5\|\mathbf{P}_{C_{\text{opt}}}\|_1 - \|\mathbf{P}_L\|_1$. Repeating the same argumentation on the modified inequalities, which is tedious but straightforward, yields

$$-\tau k \|\mathbf{P}_L\|_1 \leq 5\|\mathbf{P}_{C_{\text{opt}}}\|_1 - \|\mathbf{P}_L\|_1.$$

This implies $\|\mathbf{P}_L\|_1 \leq 5 \|\mathbf{P}_{C_{\text{opt}}}\|_1 / (1 - \tau k)$. For arbitrary $0 < \varepsilon < 1$, setting $\tau = \varepsilon/10k$, we have that $\|\mathbf{P}_L\|_1 \leq 5(1 + \varepsilon/5)\text{opt}_1$, since $1/(1 - \tau k) \leq 1 + 2\tau k = 1 + \varepsilon/5$, for $\tau \leq 1/10k$. We summarize:

Theorem 4.12. *Let \mathbf{P} be a set of n points in a metric space. For $0 < \varepsilon < 1$, one can compute a $(5 + \varepsilon)$ -approximation to the optimal k -median clustering of \mathbf{P} . The running time of the algorithm is $O(n^2 k^3 \frac{\log n}{\varepsilon})$.*

4.4. On k -means clustering

In the **k -means clustering problem**, a set $\mathbf{P} \subseteq \mathcal{X}$ is provided together with a parameter k . We would like to find a set of k points $\mathbf{C} \subseteq \mathbf{P}$, such that the sum of squared distances of all the points of \mathbf{P} to their closest point in \mathbf{C} is minimized.

Formally, given a set of centers \mathbf{C} , the k -center clustering **price** of clustering \mathbf{P} by \mathbf{C} is denoted by

$$\|\mathbf{P}_{\mathbf{C}}\|_2^2 = \sum_{\mathbf{p} \in \mathbf{P}} (\mathbf{d}_{\mathcal{M}}(\mathbf{p}, \mathbf{C}))^2,$$

and the **k -means problem** is to find a set \mathbf{C} of k points, such that $\|\mathbf{P}_{\mathbf{C}}\|_2^2$ is minimized; namely,

$$\text{opt}_2(\mathbf{P}, k) = \min_{\mathbf{C}, |\mathbf{C}|=k} \|\mathbf{P}_{\mathbf{C}}\|_2^2.$$

Local search also works for k -means and yields a constant factor approximation. We leave the proof of the following theorem to Exercise 4.4.

Theorem 4.13. *Let \mathbf{P} be a set of n points in a metric space. For $0 < \varepsilon < 1$, one can compute a $(25 + \varepsilon)$ -approximation to the optimal k -means clustering of \mathbf{P} . The running time of the algorithm is $O(n^2 k^3 \frac{\log n}{\varepsilon})$.*

4.5. Bibliographical notes

In this chapter we introduced the problem of clustering and showed some algorithms that achieve constant factor approximations. A lot more is known about these problems including faster and better clustering algorithms, but to discuss them, we need more advanced tools than what we currently have at hand.

Clustering is widely researched. Unfortunately, a large fraction of the work on this topic relies on heuristics or experimental studies. The inherent problem seems to be the lack of a universal definition of what is a good clustering. This depends on the application at hand, which is rarely clearly defined. In particular, no clustering algorithm can achieve all desired properties together; see the work by Kleinberg [Kle02] (although it is unclear if all these desired properties are indeed natural or even really desired).

k -center clustering. The algorithm **GreedyKCenter** is by Gonzalez [Gon85], but it was probably known before, as the notion of r -packing is much older. The hardness of approximating k -center clustering was shown by Feder and Greene [FG88].

k -median/means clustering. The analysis of the local search algorithm is due to Arya et al. [AGK⁺01]. Our presentation however follows the simpler proof of Gupta and Tangwongsan [GT08]. The extension to k -means is due to Kanungo et al. [KMN⁺04]. The extension is not completely trivial since the triangle inequality no longer holds. However, some approximate version of the triangle inequality does hold. Instead of performing a single swap, one can decide to do p swaps simultaneously. Thus, the running time deteriorates since there are more possibilities to check. This improves the approximation constant

for the k -median (resp., k -means) to $(3 + 2/p)$ (resp. $(3 + 2/p)^2$). Unfortunately, this is (essentially) tight in the worst case. See [AGK⁺01, KMN⁺04] for details.

The k -median and k -means clustering are more interesting in Euclidean settings where there is considerably more structure, and one can compute a $(1+\varepsilon)$ -approximation in linear time for fixed ε and k and d ; see [HM04].

Since k -median and k -means clustering can be used to solve the **dominating set** in a graph, this implies that both clustering problems are **NP-HARD** to solve exactly.

One can also compute a permutation similar to the greedy permutation (for k -center clustering) for k -median clustering. See the work by Mettu and Plaxton [MP03].

Handling outliers. The problem of handling outliers is still not well understood. See the work of Charikar et al. [CKMN01] for some relevant results. In particular, for k -center clustering they get a constant factor approximation, and Exercise 4.3 is taken from there. For k -median clustering they present a constant factor approximation using a linear programming relaxation that also approximates the number of outliers. Recently, Chen [Che08] provided a constant factor approximation algorithm by extending the work of Charikar et al. The problem of finding a simple algorithm with simple analysis for k -median clustering with outliers is still open, as Chen’s work is quite involved.

Open Problem 4.14. Get a *simple* constant factor k -median clustering algorithm that runs in polynomial time and uses exactly m outliers. Alternatively, solve this problem in the case where P is a set of n points in the plane. (The emphasize here is that the analysis of the algorithm should be simple.)

Bi-criteria approximation. All clustering algorithms tend to become considerably easier if one allows trade-off in the number of clusters. In particular, one can compute a constant factor approximation to the optimal k -median/means clustering using $O(k)$ centers in $O(nk)$ time. The algorithm succeeds with constant probability. See the work by Indyk [Ind99] and Chen [Che06] and references therein.

Facility location. All the problems mentioned here fall into the family of facility location problems. There are numerous variants. The more specific **facility location** problem is a variant of k -median clustering where the number of clusters is not specified, but instead one has to pay to open a facility in a certain location. Local search also works for this variant.

Local search. As mentioned above, **local search** also works for k -means clustering [AGK⁺01]. A collection of some basic problems for which local search works is described in the book by Kleinberg and Tardos [KT06]. Local search is a widely used heuristic for attacking **NP-HARD** problems. The idea is usually to start from a solution and try to locally improve it. Here, one defines a neighborhood of the current solution, and one tries to move to the best solution in this neighborhood. In this sense, local search can be thought of as a hill-climbing/EM (expectation maximization) algorithm. Problems for which local search was used include **vertex cover**, **traveling salesperson**, and **satisfiability**, and probably many other problems.

Provable cases where local search generates a guaranteed solution are less common and include facility location, k -median clustering [AGK⁺01], weighted max cut, k -means [KMN⁺04], the metric labeling problem with the truncated linear metric [GT00], and image segmentation [BVZ01]. See [KT06] for more references and a nice discussion of the connection of local search to the **Metropolis algorithm** and **simulated annealing**.

4.6. Exercises

Exercise 4.1 (Another algorithm for k -center clustering). Consider the algorithm that, given a point set P and a parameter k , initially picks an arbitrary set $C \subseteq P$ of k points. Next, it computes the closest pair of points $\bar{c}, \bar{f} \in C$ and the point s realizing $\|P_{C\bar{c}}\|_\infty$. If $d(s, C) > d_M(\bar{c}, \bar{f})$, then the algorithm sets $C \leftarrow C - \bar{c} + s$ and repeats this process till the condition no longer holds.

- (A) Prove that this algorithm outputs a k -center clustering of radius $\leq 2\text{opt}_\infty(P, k)$.
- (B) What is the running time of this algorithm?
- (C) If one is willing to trade off the approximation quality of this algorithm, it can be made faster. In particular, suggest a variant of this algorithm that in $O(k)$ iterations computes an $O(1)$ -approximation to the optimal k -center clustering.

Exercise 4.2 (Handling outliers). Given a point set P , we would like to perform a k -median clustering of it, where we are allowed to ignore m of the points. These m points are *outliers* which we would like to ignore since they represent irrelevant data. Unfortunately, we do not know the m outliers in advance. It is natural to conjecture that one can perform a local search for the optimal solution. Here one maintains a set of k centers and a set of m outliers. At every point in time the algorithm moves one of the centers or the outliers if it improves the solution.

Show that local search does not work for this problem; namely, the approximation factor is not a constant.

Exercise 4.3 (Handling outliers for k -center clustering). Given P , k , and m , present a polynomial time algorithm that computes a constant factor approximation to the optimal k -center clustering of P with m outliers. (Hint: Assume first that you know the radius of the optimal solution.)

Exercise 4.4 (Local search for k -means clustering). Prove Theorem 4.13.

CHAPTER 5

On Complexity, Sampling, and ε -Nets and ε -Samples

In this chapter we will try to quantify the notion of geometric complexity. It is intuitively clear that a  (i.e., disk) is a simpler shape than an  (i.e., ellipse), which is in turn simpler than a  (i.e., smiley). This becomes even more important when we consider several such shapes and how they interact with each other. As these examples might demonstrate, this notion of complexity is somewhat elusive.

To this end, we show that one can capture the structure of a distribution/point set by a small subset. The size here would depend on the complexity of the shapes/ranges we care about, but surprisingly it would be independent of the size of the point set.

5.1. VC dimension

Definition 5.1. A *range space* S is a pair (X, \mathcal{R}) , where X is a *ground set* (finite or infinite) and \mathcal{R} is a (finite or infinite) family of subsets of X . The elements of X are *points* and the elements of \mathcal{R} are *ranges*.

Our interest is in the size/weight of the ranges in the range space. For technical reasons, it will be easier to consider a finite subset x as the underlining ground set.

Definition 5.2. Let $S = (X, \mathcal{R})$ be a range space, and let x be a finite (fixed) subset of X . For a range $r \in \mathcal{R}$, its *measure* is the quantity

$$\bar{m}(r) = \frac{|r \cap x|}{|x|}.$$

While x is finite, it might be very large. As such, we are interested in getting a good estimate to $\bar{m}(r)$ by using a more compact set to represent the range space.

Definition 5.3. Let $S = (X, \mathcal{R})$ be a range space. For a subset N (which might be a multi-set) of x , its *estimate* of the measure of $\bar{m}(r)$, for $r \in \mathcal{R}$, is the quantity

$$\bar{s}(r) = \frac{|r \cap N|}{|N|}.$$

The main purpose of this chapter is to come up with methods to generate a sample N , such that $\bar{m}(r) \approx \bar{s}(r)$, for all the ranges $r \in \mathcal{R}$.

It is easy to see that in the worst case, no sample can capture the measure of all ranges. Indeed, given a sample N , consider the range $x \setminus N$ that is being completely missed by N . As such, we need to concentrate on range spaces that are “low dimensional”, where not all subsets are allowable ranges. The notion of VC dimension (named after Vapnik and Chervonenkis [VC71]) is one way to limit the complexity of a range space.

Definition 5.4. Let $S = (X, \mathcal{R})$ be a range space. For $Y \subseteq X$, let

$$(5.1) \quad \mathcal{R}_{|Y} = \left\{ \mathbf{r} \cap Y \mid \mathbf{r} \in \mathcal{R} \right\}$$

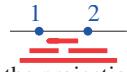
denote the **projection** of \mathcal{R} on Y . The range space S projected to Y is $S_{|Y} = (Y, \mathcal{R}_{|Y})$.

If $\mathcal{R}_{|Y}$ contains all subsets of Y (i.e., if Y is finite, we have $|\mathcal{R}_{|Y}| = 2^{|Y|}$), then Y is **shattered** by \mathcal{R} (or equivalently Y is shattered by S).

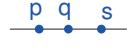
The **Vapnik-Chervonenkis dimension** (or **VC dimension**) of S , denoted by $\dim_{VC}(S)$, is the maximum cardinality of a shattered subset of X . If there are arbitrarily large shattered subsets, then $\dim_{VC}(S) = \infty$.

5.1.1. Examples.

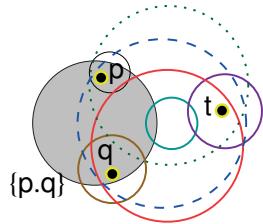
Intervals. Consider the set X to be the real line, and consider \mathcal{R} to be the set of all intervals on the real line. Consider the set $Y = \{1, 2\}$. Clearly, one can find four intervals that contain all possible subsets of Y . Formally, the projection $\mathcal{R}_{|Y} = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$. The intervals realizing each of these subsets are depicted on the right.



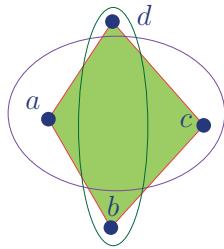
However, this is false for a set of three points $B = \{p, q, s\}$, since there is no interval that can contain the two extreme points p and s without also containing q . Namely, the subset $\{p, s\}$ is not realizable for intervals, implying that the largest shattered set by the range space (real line, intervals) is of size two. We conclude that the VC dimension of this space is two.



Disks. Let $X = \mathbb{R}^2$, and let \mathcal{R} be the set of disks in the plane. Clearly, for any three points in the plane (in general position), denoted by p, q , and s , one can find eight disks that realize all possible 2^3 different subsets. See the figure on the right.

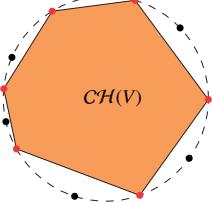


But can disks shatter a set with four points? Consider such a set P of four points. If the convex hull of P has only three points on its boundary, then the subset X having only those three vertices (i.e., it does not include the middle point) is impossible, by convexity. Namely, there is no disk that contains only the points of X without the middle point.



Alternatively, if all four points are vertices of the convex hull and they are a, b, c, d along the boundary of the convex hull, either the set $\{a, c\}$ or the set $\{b, d\}$ is not realizable. Indeed, if both options are realizable, then consider the two disks D_1 and D_2 that realize those assignments. Clearly, ∂D_1 and ∂D_2 must intersect in four points, but this is not possible, since two circles have at most two intersection points. See the figure on the left. Hence the VC dimension of this range space is 3.

Convex sets. Consider the range space $S = (\mathbb{R}^2, \mathcal{R})$, where \mathcal{R} is the set of all (closed) convex sets in the plane. We claim that $\dim_{VC}(S) = \infty$. Indeed, consider a set U of n points p_1, \dots, p_n all lying on the boundary of the unit circle in the plane. Let V be any subset of U , and consider the convex hull $CH(V)$. Clearly, $CH(V) \in \mathcal{R}$, and furthermore, $CH(V) \cap U = V$. Namely, any subset of U is realizable by S . Thus, S can shatter sets of arbitrary size, and its VC dimension is unbounded.



Complement. Consider the range space $S = (X, \mathcal{R})$ with $\delta = \dim_{VC}(S)$. Next, consider the complement space, $\bar{S} = (\bar{X}, \bar{\mathcal{R}})$, where

$$\bar{\mathcal{R}} = \left\{ X \setminus \mathbf{r} \mid \mathbf{r} \in \mathcal{R} \right\};$$

namely, the ranges of \bar{S} are the complement of the ranges in S . What is the VC dimension of \bar{S} ? Well, a set $B \subseteq X$ is shattered by \bar{S} if and only if it is shattered by S . Indeed, if S shatters B , then for any $Z \subseteq B$, we have that $(B \setminus Z) \in \mathcal{R}|_B$, which implies that $Z = B \setminus (B \setminus Z) \in \bar{\mathcal{R}}|_B$. Namely, $\bar{\mathcal{R}}|_B$ contains all the subsets of B , and \bar{S} shatters B . Thus, $\dim_{VC}(\bar{S}) = \dim_{VC}(S)$.

Lemma 5.5. *For a range space $S = (X, \mathcal{R})$ we have that $\dim_{VC}(S) = \dim_{VC}(\bar{S})$, where \bar{S} is the complement range space.*

5.1.1.1. *Halfspaces.* Let $S = (X, \mathcal{R})$, where $X = \mathbb{R}^d$ and \mathcal{R} is the set of all (closed) halfspaces in \mathbb{R}^d . We need the following technical claim.

Claim 5.6. *Let $P = \{p_1, \dots, p_{d+2}\}$ be a set of $d + 2$ points in \mathbb{R}^d . There are real numbers $\beta_1, \dots, \beta_{d+2}$, not all of them zero, such that $\sum_i \beta_i p_i = 0$ and $\sum_i \beta_i = 0$.*

PROOF. Indeed, set $q_i = (p_i, 1)$, for $i = 1, \dots, d+2$. Now, the points $q_1, \dots, q_{d+2} \in \mathbb{R}^{d+1}$ are linearly dependent, and there are coefficients $\beta_1, \dots, \beta_{d+2}$, not all of them zero, such that $\sum_{i=1}^{d+2} \beta_i q_i = 0$. Considering only the first d coordinates of these points implies that $\sum_{i=1}^{d+2} \beta_i p_i = 0$. Similarly, by considering only the $(d + 1)$ st coordinate of these points, we have that $\sum_{i=1}^{d+2} \beta_i = 0$. ■

To see what the VC dimension of halfspaces in \mathbb{R}^d is, we need the following result of Radon. (For a reminder of the formal definition of convex hulls, see Definition 28.1_{p347}.)

Theorem 5.7 (Radon's theorem). *Let $P = \{p_1, \dots, p_{d+2}\}$ be a set of $d + 2$ points in \mathbb{R}^d . Then, there exist two disjoint subsets C and D of P , such that $CH(C) \cap CH(D) \neq \emptyset$ and $C \cup D = P$.*

PROOF. By Claim 5.6 there are real numbers $\beta_1, \dots, \beta_{d+2}$, not all of them zero, such that $\sum_i \beta_i p_i = 0$ and $\sum_i \beta_i = 0$.

Assume, for the sake of simplicity of exposition, that $\beta_1, \dots, \beta_k \geq 0$ and $\beta_{k+1}, \dots, \beta_{d+2} < 0$. Furthermore, let $\mu = \sum_{i=1}^k \beta_i = -\sum_{i=k+1}^{d+2} \beta_i$. We have that

$$\sum_{i=1}^k \beta_i p_i = -\sum_{i=k+1}^{d+2} \beta_i p_i.$$

In particular, $v = \sum_{i=1}^k (\beta_i / \mu) p_i$ is a point in $CH(\{p_1, \dots, p_k\})$. Furthermore, for the same point v we have $v = \sum_{i=k+1}^{d+2} -(\beta_i / \mu) p_i \in CH(\{p_{k+1}, \dots, p_{d+2}\})$. We conclude that v is in the intersection of the two convex hulls, as required. ■

The following is a trivial observation, and yet we provide a proof to demonstrate it is true.

Lemma 5.8. *Let $P \subseteq \mathbb{R}^d$ be a finite set, let s be any point in $CH(P)$, and let h^+ be a halfspace of \mathbb{R}^d containing s . Then there exists a point of P contained inside h^+ .*

PROOF. The halfspace h^+ can be written as $h^+ = \{t \in \mathbb{R}^d \mid \langle t, v \rangle \leq c\}$. Now $s \in \mathcal{CH}(\mathcal{P}) \cap h^+$, and as such there are numbers $\alpha_1, \dots, \alpha_m \geq 0$ and points $p_1, \dots, p_m \in \mathcal{P}$, such that $\sum_i \alpha_i = 1$ and $\sum_i \alpha_i p_i = s$. By the linearity of the dot product, we have that

$$s \in h^+ \implies \langle s, v \rangle \leq c \implies \left\langle \sum_{i=1}^m \alpha_i p_i, v \right\rangle \leq c \implies \beta = \sum_{i=1}^m \alpha_i \langle p_i, v \rangle \leq c.$$

Setting $\beta_i = \langle p_i, v \rangle$, for $i = 1, \dots, m$, the above implies that β is a weighted average of β_1, \dots, β_m . In particular, there must be a β_i that is no larger than the average. That is $\beta_i \leq c$. This implies that $\langle p_i, v \rangle \leq c$. Namely, $p_i \in h^+$ as claimed. \blacksquare

Let S be the range space having \mathbb{R}^d as the ground set and all the close halfspaces as ranges. Radon's theorem implies that if a set Q of $d+2$ points is being shattered by S , then we can partition this set Q into two disjoint sets Y and Z such that $\mathcal{CH}(Y) \cap \mathcal{CH}(Z) \neq \emptyset$. In particular, let s be a point in $\mathcal{CH}(Y) \cap \mathcal{CH}(Z)$. If a halfspace h^+ contains all the points of Y , then $\mathcal{CH}(Y) \subseteq h^+$, since a halfspace is a convex set. Thus, any halfspace h^+ containing all the points of Y will contain the point $s \in \mathcal{CH}(Y)$. But $s \in \mathcal{CH}(Z) \cap h^+$, and this implies that a point of Z must lie in h^+ , by Lemma 5.8. Namely, the subset $Y \subseteq Q$ cannot be realized by a halfspace, which implies that Q cannot be shattered. Thus $\dim_{VC}(S) < d+2$. It is also easy to verify that the regular simplex with $d+1$ vertices is shattered by S . Thus, $\dim_{VC}(S) = d+1$.

5.2. Shattering dimension and the dual shattering dimension

The main property of a range space with bounded VC dimension is that the number of ranges for a set of n elements grows polynomially in n (with the power being the dimension) instead of exponentially. Formally, let the **growth function** be

$$(5.2) \quad \mathcal{G}_\delta(n) = \sum_{i=0}^{\delta} \binom{n}{i} \leq \sum_{i=0}^{\delta} \frac{n^i}{i!} \leq n^\delta,$$

for $\delta > 1$ (the cases where $\delta = 0$ or $\delta = 1$ are not interesting and we will just ignore them). Note that for all $n, \delta \geq 1$, we have $\mathcal{G}_\delta(n) = \mathcal{G}_\delta(n-1) + \mathcal{G}_{\delta-1}(n-1)$ ^①.

Lemma 5.9 (Sauer's lemma). *If (X, \mathcal{R}) is a range space of VC dimension δ with $|X| = n$, then $|\mathcal{R}| \leq \mathcal{G}_\delta(n)$.*

PROOF. The claim trivially holds for $\delta = 0$ or $n = 0$.

Let x be any element of X , and consider the sets

$$\mathcal{R}_x = \left\{ \mathbf{r} \setminus \{x\} \mid \mathbf{r} \cup \{x\} \in \mathcal{R} \text{ and } \mathbf{r} \setminus \{x\} \in \mathcal{R} \right\} \quad \text{and} \quad \mathcal{R} \setminus x = \left\{ \mathbf{r} \setminus \{x\} \mid \mathbf{r} \in \mathcal{R} \right\}.$$

Observe that $|\mathcal{R}| = |\mathcal{R}_x| + |\mathcal{R} \setminus x|$. Indeed, we charge the elements of \mathcal{R} to their corresponding element in $\mathcal{R} \setminus x$. The only bad case is when there is a range \mathbf{r} such that both $\mathbf{r} \cup \{x\} \in \mathcal{R}$ and $\mathbf{r} \setminus \{x\} \in \mathcal{R}$, because then these two distinct ranges get mapped to the same range in $\mathcal{R} \setminus x$. But such ranges contribute exactly one element to \mathcal{R}_x . Similarly, every element of \mathcal{R}_x corresponds to two such “twin” ranges in \mathcal{R} .

^①Here is a cute (and standard) counting argument: $\mathcal{G}_\delta(n)$ is just the number of different subsets of size at most δ out of n elements. Now, we either decide to not include the first element in these subsets (i.e., $\mathcal{G}_\delta(n-1)$) or, alternatively, we include the first element in these subsets, but then there are only $\delta - 1$ elements left to pick (i.e., $\mathcal{G}_{\delta-1}(n-1)$).

Observe that $(X \setminus \{x\}, \mathcal{R}_x)$ has VC dimension $\delta - 1$, as the largest set that can be shattered is of size $\delta - 1$. Indeed, any set $B \subset X \setminus \{x\}$ shattered by \mathcal{R}_x implies that $B \cup \{x\}$ is shattered in \mathcal{R} .

Thus, we have

$$|\mathcal{R}| = |\mathcal{R}_x| + |\mathcal{R} \setminus x| \leq \mathcal{G}_{\delta-1}(n-1) + \mathcal{G}_{\delta}(n-1) = \mathcal{G}_{\delta}(n),$$

by induction. \blacksquare

Interestingly, Lemma 5.9 is tight. See Exercise 5.4.

Next, we show pretty tight bounds on $\mathcal{G}_{\delta}(n)$. The proof is technical and not very interesting, and it is delegated to Section 5.6.

Lemma 5.10. *For $n \geq 2\delta$ and $\delta \geq 1$, we have $\left(\frac{n}{\delta}\right)^{\delta} \leq \mathcal{G}_{\delta}(n) \leq 2\left(\frac{ne}{\delta}\right)^{\delta}$, where $\mathcal{G}_{\delta}(n) = \sum_{i=0}^{\delta} \binom{n}{i}$.*

Definition 5.11 (Shatter function). Given a range space $S = (X, \mathcal{R})$, its **shatter function** $\pi_S(m)$ is the maximum number of sets that might be created by S when restricted to subsets of size m . Formally,

$$\pi_S(m) = \max_{\substack{B \subset X \\ |B|=m}} |\mathcal{R}_{|B}|;$$

see (5.1).

The **shattering dimension** of S is the smallest d such that $\pi_S(m) = O(m^d)$, for all m .

By applying Lemma 5.9 to a finite subset of X , we get:

Corollary 5.12. *If $S = (X, \mathcal{R})$ is a range space of VC dimension δ , then for every finite subset B of X , we have $|\mathcal{R}_{|B}| \leq \pi_S(|B|) \leq \mathcal{G}_{\delta}(|B|)$. That is, the VC dimension of a range space always bounds its shattering dimension.*

PROOF. Let $n = |B|$, and observe that $|\mathcal{R}_{|B}| \leq \mathcal{G}_{\delta}(n) \leq n^{\delta}$, by (5.2). As such, $|\mathcal{R}_{|B}| \leq n^{\delta}$, and, by definition, the shattering dimension of S is at most δ ; namely, the shattering dimension is bounded by the VC dimension. \blacksquare

Our arch-nemesis in the following is the function $x / \ln x$. The following lemma states some properties of this function, and its proof is delegated to Exercise 5.2.

Lemma 5.13. *For the function $f(x) = x / \ln x$ the following hold.*

- (A) $f(x)$ is monotonically increasing for $x \geq e$.
- (B) $f(x) \geq e$, for $x > 1$.
- (C) For $u \geq \sqrt{e}$, if $f(x) \leq u$, then $x \leq 2u \ln u$.
- (D) For $u \geq \sqrt{e}$, if $x > 2u \ln u$, then $f(x) > u$.
- (E) For $u \geq e$, if $f(x) \geq u$, then $x \geq u \ln u$.

The next lemma introduces a standard argument which is useful in bounding the VC dimension of a range space by its shattering dimension. It is easy to see that the bound is tight in the worst case.

Lemma 5.14. *If $S = (X, \mathcal{R})$ is a range space with shattering dimension d , then its VC dimension is bounded by $O(d \log d)$.*

PROOF. Let $N \subseteq X$ be the largest set shattered by S , and let δ denote its cardinality. We have that $2^\delta = |\mathcal{R}_{|N}| \leq \pi_S(|N|) \leq c\delta^d$, where c is a fixed constant. As such, we have that $\delta \leq \lg c + d \lg \delta$, which in turn implies that $\frac{\delta - \lg c}{\lg \delta} \leq d$.^② Assuming $\delta \geq \max(2, 2 \lg c)$, we have that

$$\frac{\delta}{2 \lg \delta} \leq d \implies \frac{\delta}{\ln \delta} \leq \frac{2d}{\ln 2} \leq 6d \implies \delta \leq 2(6d) \ln(6d),$$

by Lemma 5.13(C). \blacksquare

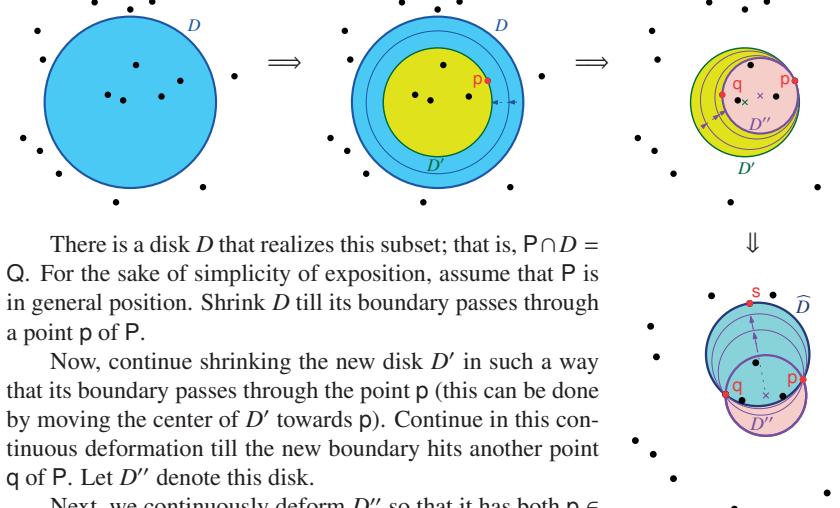
Disks revisited. To see why the shattering dimension is more convenient to work with than the VC dimension, consider the range space $S = (X, \mathcal{R})$, where $X = \mathbb{R}^2$ and \mathcal{R} is the set of disks in the plane. We know that the VC dimension of S is 3 (see Section 5.1.1).

We next use a standard continuous deformation argument to argue that the shattering dimension of this range space is also 3.

Lemma 5.15. *Consider the range space $S = (X, \mathcal{R})$, where $X = \mathbb{R}^2$ and \mathcal{R} is the set of disks in the plane. The shattering dimension of S is 3.*

PROOF. Consider any set P of n points in the plane, and consider the set $\mathcal{F} = \mathcal{R}_{|P}$. We claim that $|\mathcal{F}| \leq 4n^3$.

The set \mathcal{F} contains only n sets with a single point in them and only $\binom{n}{2}$ sets with two points in them. So, fix $Q \in \mathcal{F}$ such that $|Q| \geq 3$.



There is a disk D that realizes this subset; that is, $P \cap D = Q$. For the sake of simplicity of exposition, assume that P is in general position. Shrink D till its boundary passes through a point p of P .

Now, continue shrinking the new disk D' in such a way that its boundary passes through the point p (this can be done by moving the center of D' towards p). Continue in this continuous deformation till the new boundary hits another point q of P . Let D'' denote this disk.

Next, we continuously deform D'' so that it has both $p \in Q$ and $q \in Q$ on its boundary. This can be done by moving the center of D'' along the bisector line between p and q . Stop as soon as the boundary of the disk hits a third point $s \in P$. (We have freedom in choosing in which direction to move the center. As such, move in the direction that causes the disk boundary to hit a new point s .) Let \widehat{D} be the resulting disk. The boundary of \widehat{D} is the unique circle passing through p, q , and s . Furthermore, observe that

$$D \cap (P \setminus \{s\}) = \widehat{D} \cap (P \setminus \{s\}).$$

^②We remind the reader that $\lg = \log_2$.

That is, we can specify the point set $P \cap D$ by specifying the three points p, q, s (and thus specifying the disk \widehat{D}) and the status of the three special points; that is, we specify for each point p, q, s whether or not it is inside the generated subset.

As such, there are at most $8\binom{n}{3}$ different subsets in \mathcal{F} containing more than three points, as each such subset maps to a “canonical” disk, there are at most $\binom{n}{3}$ different such disks, and each such disk defines at most eight different subsets.

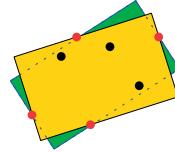
Similar argumentation implies that there are at most $4\binom{n}{2}$ subsets that are defined by a pair of points that realizes the diameter of the resulting disk. Overall, we have that

$$|\mathcal{F}| = 1 + n + 4\binom{n}{2} + 8\binom{n}{3} \leq 4n^3,$$

since there is one empty set in \mathcal{F} , n sets of size 1, and the rest of the sets are counted as described above. \blacksquare

The proof of Lemma 5.15 might not seem like a great simplification over the same bound we got by arguing about the VC dimension. However, the above argumentation gives us a very powerful tool – the shattering dimension of a range space defined by a family of shapes is always bounded by the number of points that determine a shape in the family.

Thus, the shattering dimension of, say, arbitrarily oriented rectangles in the plane is bounded by (and in this case, equal to) five, since such a rectangle is uniquely determined by five points. To see that, observe that if a rectangle has only four points on its boundary, then there is one degree of freedom left, since we can rotate the rectangle “around” these points; see the figure on the right.



5.2.1. The dual shattering dimension. Given a range space $S = (X, \mathcal{R})$, consider a point $p \in X$. There is a set of ranges of \mathcal{R} associated with p , namely, the set of all ranges of \mathcal{R} that contains p which we denote by

$$\mathcal{R}_p = \left\{ r \mid r \in \mathcal{R}, \text{ the range } r \text{ contains } p \right\}.$$

This gives rise to a natural dual range space to S .

Definition 5.16. The *dual range space* to a range space $S = (X, \mathcal{R})$ is the space $S^* = (\mathcal{R}, X^*)$, where $X^* = \{\mathcal{R}_p \mid p \in X\}$.

Naturally, the dual range space to S^* is the original S , which is thus sometimes referred to as the *primal range space*. (In other words, the dual to the dual is the primal.) The easiest way to see this, is to think about it as an abstract set system realized as an incidence matrix, where each point is a column and a set is a row in the matrix having 1 in an entry if and only if it contains the corresponding point; see Figure 5.1. Now, it is easy to verify that the dual range space is the transposed matrix.

To understand what the dual space is, consider X to be the plane and \mathcal{R} to be a set of m disks. Then, in the dual range space $S^* = (\mathcal{R}, X^*)$, every point p in the plane has a set associated with it in X^* , which is the set of disks of \mathcal{R} that contains p . In particular, if we consider the arrangement formed by the m disks of \mathcal{R} , then all the points lying inside a single face of this arrangement correspond to the same set of X^* . The number of ranges in X^* is bounded by the complexity of the arrangement of these disks, which is $O(m^2)$; see Figure 5.1.

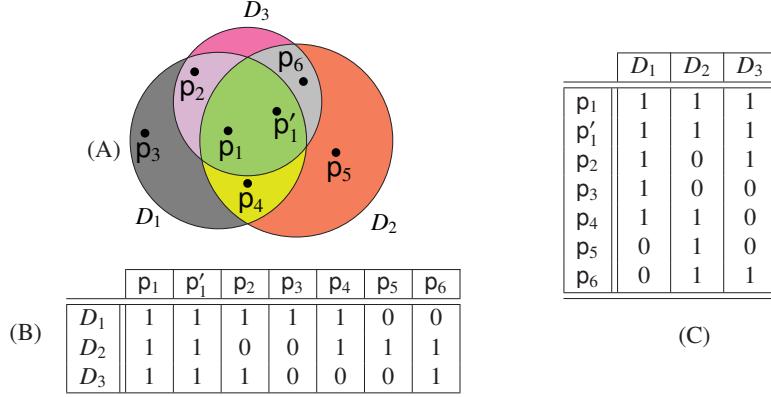


FIGURE 5.1. (A) $\mathcal{R}_{p_1} = \mathcal{R}_{p'_1}$. (B) Writing the set system as an incidence matrix where a point is a column and a set is a row. For example, D_2 contains p_4 , and as such the column of p_4 has a 1 in the row corresponding to D_2 . (C) The dual set system is represented by a matrix which is the transpose of the original incidence matrix.

Let the **dual shatter function** of the range space S be $\pi_S^*(m) = \pi_{S^*}(m)$, where S^* is the dual range space to S .

Definition 5.17. The **dual shattering dimension** of S is the shattering dimension of the dual range space S^* .

Note that the dual shattering dimension might be smaller than the shattering dimension and hence also smaller than the VC dimension of the range space. Indeed, in the case of disks in the plane, the dual shattering dimension is just 2, while the VC dimension and the shattering dimension of this range space is 3. Note, also, that in geometric settings bounding the dual shattering dimension is relatively easy, as all you have to do is bound the complexity of the arrangement of m ranges of this space.

The following lemma shows a connection between the VC dimension of a space and its dual. The interested reader^③ might find the proof amusing.

Lemma 5.18. Consider a range space $S = (X, \mathcal{R})$ with VC dimension δ . The dual range space $S^* = (\mathcal{R}, X^*)$ has VC dimension bounded by $2^{\delta+1}$.

PROOF. Assume that S^* shatters a set $\mathcal{F} = \{\mathbf{r}_1, \dots, \mathbf{r}_k\} \subseteq \mathcal{R}$ of k ranges. Then, there is a set $P \subseteq X$ of $m = 2^k$ points that shatters \mathcal{F} . Formally, for every subset $V \subseteq \mathcal{F}$, there exists a point $p \in P$, such that $\mathcal{F}_p = V$.

So, consider the matrix M (of dimensions $k \times 2^k$) having the points p_1, \dots, p_{2^k} of P as the columns, and every row is a set of \mathcal{F} , where the entry in the matrix corresponding to a point $p \in P$ and a range $r \in \mathcal{F}$ is 1 if and only if $p \in r$ and zero otherwise. Since P shatters \mathcal{F} , we know that this matrix has all possible 2^k binary vectors as columns.

^③The author is quite aware that the interest of the reader in this issue might not be the result of free choice. Nevertheless, one might draw some comfort from the realization that the existence of the interested reader is as much an illusion as the existence of free choice. Both are convenient to assume, and both are probably false. Or maybe not.

Next, let $\kappa' = 2^{\lfloor \lg k \rfloor} \leq k$, and consider the matrix \mathbf{M}' of size $\kappa' \times \lg \kappa'$, where the i th row is the binary representation of the number $i - 1$ (formally, the j th entry in the i th row is 1 if the j th bit in the binary representation of $i - 1$ is 1), where $i = 1, \dots, \kappa'$. See the figure on the right. Clearly, the $\lg \kappa'$ columns of \mathbf{M}' are all different, and we can find $\lg \kappa'$ columns of \mathbf{M} that are identical to the columns of \mathbf{M}' (in the first κ' entries starting from the top of the columns).

	p_1	p_2	\dots	p_{2^k}	
r_1	0	1		0	0 0 0
r_2	1	1		1	0 0 1
\vdots	\vdots	\vdots	\vdots	\vdots	0 1 0
r_{k-2}	1	1	\dots	0	0 1 1
r_{k-1}	0	0	\dots	1	1 0 0
r_k	1	0	\dots	1	1 0 1

	0	0	0
	0	0	1
	0	1	0
	0	1	1
	1	0	0
	1	0	1
	1	1	0
	1	1	1

Each such column corresponds to a point $p \in P$, and let $Q \subset P$ be this set of $\lg \kappa'$ points. Note that for any subset $Z \subseteq Q$, there is a row t in \mathbf{M}' that encodes this subset. Consider the corresponding row in \mathbf{M} ; that is, the range $r_t \in \mathcal{F}$. Since \mathbf{M} and \mathbf{M}' are identical (in the relevant $\lg \kappa'$ columns of \mathbf{M}) on the first κ' , we have that $r_t \cap Q = Z$. Namely, the set of ranges \mathcal{F} shatters Q . But since the original range space has VC dimension δ , it follows that $|Q| \leq \delta$. Namely, $|Q| = \lg \kappa' = \lfloor \lg k \rfloor \leq \delta$, which implies that $\lg k \leq \delta + 1$, which in turn implies that $k \leq 2^{\delta+1}$. ■

Lemma 5.19. *If a range space $S = (X, \mathcal{R})$ has dual shattering dimension δ , then its VC dimension is bounded by $\delta^{O(\delta)}$.*

PROOF. The shattering dimension of the dual range space S^* is bounded by δ , and as such, by Lemma 5.14, its VC dimension is bounded by $\delta' = O(\delta \log \delta)$. Since the dual range space to S^* is S , we have by Lemma 5.18 that the VC dimension of S is bounded by $2^{\delta'+1} = \delta^{O(\delta)}$. ■

The bound of Lemma 5.19 might not be pretty, but it is sufficient in a lot of cases to bound the VC dimension when the shapes involved are simple.

Example 5.20. Consider the range space $S = (\mathbb{R}^2, \mathcal{R})$, where \mathcal{R} is a set of shapes in the plane, so that the boundary of any pair of them intersects at most s times. Then, the VC dimension of S is $O(1)$. Indeed, the dual shattering dimension of S is $O(1)$, since the complexity of the arrangement of n such shapes is $O(sn^2)$. As such, by Lemma 5.19, the VC dimension of S is $O(1)$.

5.2.1.1. Mixing range spaces.

Lemma 5.21. *Let $S = (X, \mathcal{R})$ and $T = (X, \mathcal{R}')$ be two range spaces of VC dimension δ and δ' , respectively, where $\delta, \delta' > 1$. Let $\widehat{\mathcal{R}} = \{r \cup r' \mid r \in \mathcal{R}, r' \in \mathcal{R}'\}$. Then, for the range space $\widehat{S} = (X, \widehat{\mathcal{R}})$, we have that $\dim_{VC}(\widehat{S}) = O(\delta + \delta')$.*

PROOF. As a warm-up exercise, we prove a somewhat weaker bound here of $O((\delta + \delta') \log(\delta + \delta'))$. The stronger bound follows from Theorem 5.22 below. Let B be a set of n points in X that are shattered by \widehat{S} . There are at most $\mathcal{G}_\delta(n)$ and $\mathcal{G}_{\delta'}(n)$ different ranges of B in the range sets $\mathcal{R}|_B$ and $\mathcal{R}'|_B$, respectively, by Lemma 5.9. Every subset C of B realized by $\widehat{r} \in \widehat{\mathcal{R}}$ is a union of two subsets $B \cap r$ and $B \cap r'$, where $r \in \mathcal{R}$ and $r' \in \mathcal{R}'$, respectively. Thus, the number of different subsets of B realized by \widehat{S} is bounded by $\mathcal{G}_\delta(n)\mathcal{G}_{\delta'}(n)$. Thus, $2^n \leq n^\delta n^{\delta'}$, for $\delta, \delta' > 1$. We conclude that $n \leq (\delta + \delta') \lg n$, which implies that $n = O((\delta + \delta') \log(\delta + \delta'))$, by Lemma 5.13(C). ■

Interestingly, one can prove a considerably more general result with tighter bounds. The required computations are somewhat more painful.

Theorem 5.22. *Let $S_1 = (X, \mathcal{R}^1), \dots, S_k = (X, \mathcal{R}^k)$ be range spaces with VC dimension $\delta_1, \dots, \delta_k$, respectively. Next, let $f(\mathbf{r}_1, \dots, \mathbf{r}_k)$ be a function that maps any k -tuple of sets $\mathbf{r}_1 \in \mathcal{R}^1, \dots, \mathbf{r}_k \in \mathcal{R}^k$ into a subset of X . Consider the range set*

$$\mathcal{R}' = \left\{ f(\mathbf{r}_1, \dots, \mathbf{r}_k) \mid \mathbf{r}_1 \in \mathcal{R}_1, \dots, \mathbf{r}_k \in \mathcal{R}_k \right\}$$

and the associated range space $T = (X, \mathcal{R}')$. Then, the VC dimension of T is bounded by $O(k\delta \lg k)$, where $\delta = \max_i \delta_i$.

PROOF. Assume a set $Y \subseteq X$ of size t is being shattered by \mathcal{R}' , and observe that

$$\begin{aligned} |\mathcal{R}'_{|Y}| &\leq \left| \left\{ (\mathbf{r}_1, \dots, \mathbf{r}_k) \mid \mathbf{r}_1 \in \mathcal{R}_1^1, \dots, \mathbf{r}_k \in \mathcal{R}_Y^k \right\} \right| \leq |\mathcal{R}_1^1| \cdots |\mathcal{R}_Y^k| \leq \mathcal{G}_{\delta_1}(t) \cdot \mathcal{G}_{\delta_2}(t) \cdots \mathcal{G}_{\delta_k}(t) \\ &\leq (\mathcal{G}_{\delta}(t))^k \leq \left(2 \left(\frac{te}{\delta} \right)^{\delta} \right)^k, \end{aligned}$$

by Lemma 5.9 and Lemma 5.10. On the other hand, since Y is being shattered by \mathcal{R}' , this implies that $|\mathcal{R}'_{|Y}| = 2^t$. Thus, we have the inequality $2^t \leq \left(2 \left(\frac{te}{\delta} \right)^{\delta} \right)^k$, which implies $t \leq k(1 + \delta \lg(te/\delta))$. Assume that $t \geq e$ and $\delta \lg(te/\delta) \geq 1$ since otherwise the claim is trivial, and observe that $t \leq k(1 + \delta \lg(te/\delta)) \leq 3k\delta \lg(t/\delta)$. Setting $x = t/\delta$, we have

$$\frac{t}{\delta} \leq 3k \frac{\ln(t/\delta)}{\ln 2} \leq 6k \ln \frac{t}{\delta} \implies \frac{x}{\ln x} \leq 6k \implies x \leq 2 \cdot 6k \ln(6k) \implies x \leq 12k \ln(6k),$$

by Lemma 5.13(C). We conclude that $t \leq 12\delta k \ln(6k)$, as claimed. \blacksquare

Corollary 5.23. *Let $S = (X, \mathcal{R})$ and $T = (X, \mathcal{R}')$ be two range spaces of VC dimension δ and δ' , respectively, where $\delta, \delta' > 1$. Let $\widehat{\mathcal{R}} = \left\{ \mathbf{r} \cap \mathbf{r}' \mid \mathbf{r} \in \mathcal{R}, \mathbf{r}' \in \mathcal{R}' \right\}$. Then, for the range space $\widehat{S} = (X, \widehat{\mathcal{R}})$, we have that $\dim_{VC}(\widehat{S}) = O(\delta + \delta')$.*

Corollary 5.24. *Any finite sequence of combining range spaces with finite VC dimension (by intersecting, complementing, or taking their union) results in a range space with a finite VC dimension.*

5.3. On ε -nets and ε -sampling

5.3.1. ε -nets and ε -samples.

Definition 5.25 (ε -sample). Let $S = (X, \mathcal{R})$ be a range space, and let x be a finite subset of X . For $0 \leq \varepsilon \leq 1$, a subset $C \subseteq x$ is an **ε -sample** for x if for any range $\mathbf{r} \in \mathcal{R}$, we have

$$\left| \bar{m}(\mathbf{r}) - \bar{s}(\mathbf{r}) \right| \leq \varepsilon,$$

where $\bar{m}(\mathbf{r}) = |x \cap \mathbf{r}| / |x|$ is the measure of \mathbf{r} (see Definition 5.2) and $\bar{s}(\mathbf{r}) = |C \cap \mathbf{r}| / |C|$ is the estimate of \mathbf{r} (see Definition 5.3). (Here C might be a multi-set, and as such $|C \cap \mathbf{r}|$ is counted with multiplicity.)

As such, an ε -sample is a subset of the ground set x that “captures” the range space up to an error of ε . Specifically, to estimate the fraction of the ground set covered by a range \mathbf{r} , it is sufficient to count the points of C that fall inside \mathbf{r} .

If X is a finite set, we will abuse notation slightly and refer to C as an **ε -sample** for S .

To see the usage of such a sample, consider $x = X$ to be, say, the population of a country (i.e., an element of X is a citizen). A range in \mathcal{R} is the set of all people in the country that answer yes to a question (i.e., would you vote for party Y?, would you buy a bridge from me?, questions like that). An ε -sample of this range space enables us to estimate reliably (up to an error of ε) the answers for all these questions, by just asking the people in the sample.

The natural question of course is how to find such a subset of small (or minimal) size.

Theorem 5.26 (ε -sample theorem, [VC71]). *There is a positive constant c such that if (X, \mathcal{R}) is any range space with VC dimension at most δ , $x \subseteq X$ is a finite subset and $\varepsilon, \varphi > 0$, then a random subset $C \subseteq x$ of cardinality*

$$s = \frac{c}{\varepsilon^2} \left(\delta \log \frac{\delta}{\varepsilon} + \log \frac{1}{\varphi} \right)$$

is an ε -sample for x with probability at least $1 - \varphi$.

(In the above theorem, if $s > |x|$, then we can just take all of x to be the ε -sample.)

For a strengthened version of the above theorem with slightly better bounds, see Theorem 7.13_{p107}.

Sometimes it is sufficient to have (hopefully smaller) samples with a weaker property – if a range is “heavy”, then there is an element in our sample that is in this range.

Definition 5.27 (ε -net). A set $N \subseteq x$ is an ε -net for x if for any range $r \in \mathcal{R}$, if $\bar{m}(r) \geq \varepsilon$ (i.e., $|r \cap x| \geq \varepsilon |x|$), then r contains at least one point of N (i.e., $r \cap N \neq \emptyset$).

Theorem 5.28 (ε -net theorem, [HW87]). *Let (X, \mathcal{R}) be a range space of VC dimension δ , let x be a finite subset of X , and suppose that $0 < \varepsilon \leq 1$ and $\varphi < 1$. Let N be a set obtained by m random independent draws from x , where*

$$(5.3) \quad m \geq \max \left(\frac{4}{\varepsilon} \lg \frac{4}{\varphi}, \frac{8\delta}{\varepsilon} \lg \frac{16}{\varepsilon} \right).$$

Then N is an ε -net for x with probability at least $1 - \varphi$.

(We remind the reader that $\lg = \log_2$.)

The proofs of the above theorems are somewhat involved and we first turn our attention to some applications before presenting the proofs.

Remark 5.29. The above two theorems also hold for spaces with shattering dimension at most δ , in which case the sample size is slightly larger. Specifically, for Theorem 5.28, the sample size needed is $O\left(\frac{1}{\varepsilon} \lg \frac{1}{\varphi} + \frac{\delta}{\varepsilon} \lg \frac{\delta}{\varepsilon}\right)$.

5.3.2. Some applications. We mention two (easy) applications of these theorems, which (hopefully) demonstrate their power.

5.3.2.1. *Range searching.* So, consider a (very large) set of points P in the plane. We would like to be able to quickly decide how many points are included inside a query rectangle. Let us assume that we allow ourselves 1% error. What Theorem 5.26 tells us is that there is a subset of *constant size* (that depends only on ε) that can be used to perform this estimation, and it works for *all* query rectangles (we used here the fact that rectangles in the plane have finite VC dimension). In fact, a random sample of this size works with constant probability.

5.3.2.2. Learning a concept. Assume that we have a function f defined in the plane that returns ‘1’ inside an (unknown) disk D_{unknown} and ‘0’ outside it. There is some distribution \mathcal{D} defined over the plane, and we pick points from this distribution. Furthermore, we can compute the function for these labels (i.e., we can compute f for certain values, but it is expensive). For a mystery value $\varepsilon > 0$, to be explained shortly, Theorem 5.28 tells us to pick (roughly) $O((1/\varepsilon)\log(1/\varepsilon))$ random points in a sample R from this distribution and to compute the labels for the samples. This is demonstrated in the figure on the right, where black dots are the sample points for which $f(\cdot)$ returned 1.

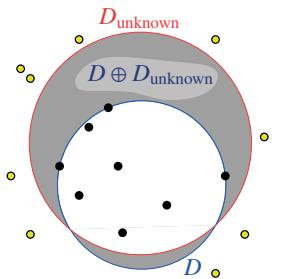
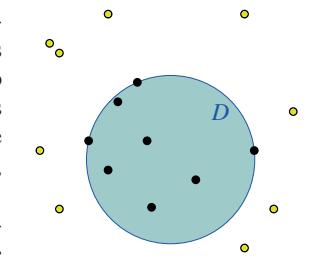
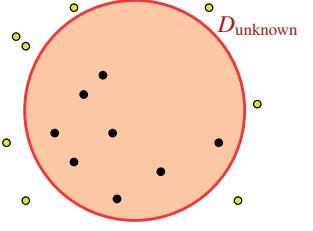
So, now we have positive examples and negative examples. We would like to find a hypothesis that agrees with all the samples we have and that hopefully is close to the true unknown disk underlying the function f . To this end, compute the smallest disk D that contains the sample labeled by ‘1’ and does not contain any of the ‘0’ points, and let $g : \mathbb{R}^2 \rightarrow \{0, 1\}$ be the function g that returns ‘1’ inside the disk and ‘0’ otherwise. We claim that g classifies correctly all but an ε -fraction of the points (i.e., the probability of misclassifying a point picked according to the given distribution is smaller than ε); that is, $\Pr_{p \in \mathcal{D}}[f(p) \neq g(p)] \leq \varepsilon$.

Geometrically, the region where g and f disagree is all the points in the symmetric difference between the two disks. That is, $\mathcal{E} = D \oplus D_{\text{unknown}}$; see the figure on the right.

Thus, consider the range space S having the plane as the ground set and the symmetric difference between any two disks as its ranges. By Corollary 5.24, this range space has finite VC dimension. Now, consider the (unknown) disk D' that induces f and the region $r = D_{\text{unknown}} \oplus D$. Clearly, the learned classifier g returns incorrect answers only for points picked inside r .

Thus, the probability of a mistake in the classification is the measure of r under the distribution \mathcal{D} . So, if $\Pr_{\mathcal{D}}[r] > \varepsilon$ (i.e., the probability that a sample point falls inside r), then by the ε -net theorem (i.e., Theorem 5.28) the set R is an ε -net for S (ignore for the time being the possibility that the random sample fails to be an ε -net) and as such, R contains a point q inside r . But, it is not possible for g (which classifies correctly all the sampled points of R) to make a mistake on q , a contradiction, because by construction, the range r is where g misclassifies points. We conclude that $\Pr_{\mathcal{D}}[r] \leq \varepsilon$, as desired.

Little lies. The careful reader might be tearing his or her hair out because of the above description. First, Theorem 5.28 might fail, and the above conclusion might not hold. This is of course true, and in real applications one might use a much larger sample to guarantee that the probability of failure is so small that it can be practically ignored. A more serious issue is that Theorem 5.28 is defined only for finite sets. Nowhere does it speak about a continuous distribution. Intuitively, one can approximate a continuous distribution to an



arbitrary precision using a huge sample and apply the theorem to this sample as our ground set. A formal proof is more tedious and requires extending the proof of Theorem 5.28 to continuous distributions. This is straightforward and we will ignore this topic altogether.

5.3.2.3. A naive proof of the ε -sample theorem. To demonstrate why the ε -sample/net theorems are interesting, let us try to prove the ε -sample theorem in the natural naive way. Thus, consider a finite range space $S = (x, \mathcal{R})$ with shattering dimension δ . Also, consider a range r that contains, say, a p fraction of the points of x , where $p \geq \varepsilon$. Consider a random sample R of r points from x , picked with replacement.

Let p_i be the i th sample point, and let X_i be an indicator variable which is one if and only if $p_i \in r$. Clearly, $(\sum_i X_i)/r$ is an estimate for $p = |r \cap x| / |x|$. We would like this estimate to be within $\pm \varepsilon$ of p and with confidence $\geq 1 - \varphi$.

As such, the sample failed if $|\sum_{i=1}^r X_i - pr| \geq \varepsilon r = (\varepsilon/p)pr$. Set $\phi = \varepsilon/p$ and $\mu = E[\sum_i X_i] = pr$. Using Chernoff's inequality (Theorem 27.17_{p340} and Theorem 27.18_{p341}), we have

$$\begin{aligned} \Pr\left[\left|\sum_{i=1}^r X_i - pr\right| \geq (\varepsilon/p)pr\right] &= \Pr\left[\left|\sum_{i=1}^r X_i - \mu\right| \geq \phi\mu\right] \leq \exp(-\mu\phi^2/2) + \exp(-\mu\phi^2/4) \\ &\leq 2 \exp(-\mu\phi^2/4) = 2 \exp\left(-\frac{\varepsilon^2}{4p}r\right) \leq \varphi, \end{aligned}$$

$$\text{for } r \geq \left\lceil \frac{4}{\varepsilon^2} \ln \frac{2}{\varphi} \right\rceil \geq \left\lceil \frac{4p}{\varepsilon^2} \ln \frac{2}{\varphi} \right\rceil.$$

Viola! We proved the ε -sample theorem. Well, not quite. We proved that the sample works correctly for a single range. Namely, we proved that for a specific range $r \in \mathcal{R}$, we have that $\Pr[|\bar{m}(r) - \bar{s}(r)| > \varepsilon] \leq \varphi$. However, we need to prove that $\forall r \in \mathcal{R}$, $\Pr[|\bar{m}(r) - \bar{s}(r)| > \varepsilon] \leq \varphi$.

Now, naively, we can overcome this by using a union bound on the bad probability. Indeed, if there are k different ranges under consideration, then we can use a sample that is large enough such that the probability of it to fail for each range is at most φ/k . In particular, let \mathcal{E}_i be the bad event that the sample fails for the i th range. We have that $\Pr[\mathcal{E}_i] \leq \varphi/k$, which implies that

$$\Pr[\text{sample fails for any range}] \leq \Pr\left[\bigcup_{i=1}^k \mathcal{E}_i\right] \leq \sum_{i=1}^k \Pr[\mathcal{E}_i] \leq k(\varphi/k) \leq \varphi,$$

by the union bound; that is, the sample works for all ranges with good probability.

However, the number of ranges that we need to prove the theorem for is $\pi_S(|x|)$ (see Definition 5.11). In particular, if we plug in confidence $\varphi/\pi_S(|x|)$ to the above analysis and use the union bound, we get that for

$$r \geq \left\lceil \frac{4}{\varepsilon^2} \ln \frac{\pi_S(|x|)}{\varphi} \right\rceil$$

the sample estimates correctly (up to $\pm \varepsilon$) the size of all ranges with confidence $\geq 1 - \varphi$. Bounding $\pi_S(|x|)$ by $O(|x|^\delta)$ (using (5.2)_{p64} for a space with VC dimension δ), we can bound the required size of r by $O(\delta\varepsilon^{-2} \log(|x|/\varphi))$. We summarize the result.

Lemma 5.30. *Let (x, \mathcal{R}) be a finite range space with VC dimension at most δ , and let $\varepsilon, \varphi > 0$ be parameters. Then a random subset $C \subseteq x$ of cardinality $O(\delta\varepsilon^{-2} \log(|x|/\varphi))$ is an ε -sample for x with probability at least $1 - \varphi$.*

Namely, the “naive” argumentation gives us a sample bound which depends on the underlying size of the ground set. However, the sample size in the ε -sample theorem (Theorem 5.26) is independent of the size of the ground set. This is the magical property of the ε -sample theorem^④.

Interestingly, using a chaining argument on Lemma 5.30, one can prove the ε -sample theorem for the finite case; see Exercise 5.3. We provide a similar proof when using discrepancy, in Section 5.4. However, the original proof uses a clever double sampling idea that is both interesting and insightful that makes the proof work for the infinite case also.

5.3.3. A quicky proof of the ε -net theorem (Theorem 5.28). Here we provide a sketchy proof of Theorem 5.28, which conveys the main ideas. The full proof in all its glory and details is provided in Section 5.5.

Let $N = (x_1, \dots, x_m)$ be the sample obtained by m independent samples from x (observe that N might contain the same element several times, and as such it is a multi-set). Let \mathcal{E}_1 be the probability that N fails to be an ε -net. Namely, for $n = |x|$, let

$$\mathcal{E}_1 = \left\{ \exists \mathbf{r} \in \mathcal{R} \mid |\mathbf{r} \cap x| \geq \varepsilon n \text{ and } \mathbf{r} \cap N = \emptyset \right\}.$$

To complete the proof, we must show that $\Pr[\mathcal{E}_1] \leq \varphi$.

Let $T = (y_1, \dots, y_m)$ be another random sample generated in a similar fashion to N . It might be that N fails for a certain range \mathbf{r} , but then since T is an independent sample, we still expect that $|\mathbf{r} \cap T| = \varepsilon m$. In particular, the probability that $\Pr[|\mathbf{r} \cap T| \geq \frac{\varepsilon m}{2}]$ is a large constant close to 1, regardless of how N performs. Indeed, if m is sufficiently large, we expect the random variable $|\mathbf{r} \cap T|$ to concentrate around εm , and one can argue this formally using Chernoff’s inequality. Namely, intuitively, for a heavy range \mathbf{r} we have that

$$\Pr[\mathbf{r} \cap N = \emptyset] \approx \Pr\left[\mathbf{r} \cap N = \emptyset \text{ and } \left(|\mathbf{r} \cap T| \geq \frac{\varepsilon m}{2}\right)\right].$$

Inspired by this, let \mathcal{E}_2 be the event that N fails for some range \mathbf{r} but T “works” for \mathbf{r} ; formally

$$\mathcal{E}_2 = \left\{ \exists \mathbf{r} \in \mathcal{R} \mid |\mathbf{r} \cap x| \geq \varepsilon n, \mathbf{r} \cap N = \emptyset \text{ and } |\mathbf{r} \cap T| \geq \frac{\varepsilon m}{2} \right\}.$$

Intuitively, since $\mathbf{E}[|\mathbf{r} \cap T|] \geq \varepsilon m$, then for the range \mathbf{r} that N fails for, we have with “good” probability that $|\mathbf{r} \cap T| \geq \varepsilon m/2$. Namely, $\Pr[\mathcal{E}_1] \approx \Pr[\mathcal{E}_2]$.

Next, let

$$\mathcal{E}'_2 = \left\{ \exists \mathbf{r} \in \mathcal{R} \mid \mathbf{r} \cap N = \emptyset \text{ and } |\mathbf{r} \cap T| \geq \frac{\varepsilon m}{2} \right\}.$$

Clearly, $\mathcal{E}_2 \subseteq \mathcal{E}'_2$ and as such $\Pr[\mathcal{E}_2] \leq \Pr[\mathcal{E}'_2]$. Now, fix $Z = N \cup T$, and observe that $|Z| = 2m$. Next, fix a range \mathbf{r} , and observe that the bad probability of \mathcal{E}'_2 is maximized if $|\mathbf{r} \cap Z| = \varepsilon m/2$. Now, the probability that all the elements of $\mathbf{r} \cap Z$ fall only into the second half of the sample is at most $2^{-\varepsilon m/2}$ as a careful calculation shows. Now, there are at most $|Z|_{\mathcal{R}} \leq \mathcal{G}_d(2m)$ different ranges that one has to consider. As such, $\Pr[\mathcal{E}_1] \approx \Pr[\mathcal{E}_2] \leq \Pr[\mathcal{E}'_2] \leq \mathcal{G}_d(2m)2^{-\varepsilon m/2}$ and this is smaller than φ , as a careful calculation shows by just plugging the value of m into the right-hand side; see (5.3)_{p71}. ■

^④The notion of magic is used here in the sense of Arthur C. Clarke’s statement that “any sufficiently advanced technology is indistinguishable from magic.”

5.4. Discrepancy

The proof of the ε -sample/net theorem is somewhat complicated. It turns out that one can get a somewhat similar result by attacking the problem from the other direction; namely, let us assume that we would like to take a truly large sample of a finite range space $S = (X, \mathcal{R})$ defined over n elements with m ranges. We would like this sample to be as representative as possible as far as S is concerned. In fact, let us decide that we would like to pick exactly half of the points of X in our sample (assume that $n = |X|$ is even).

To this end, let us color half of the points of X by -1 (i.e., black) and the other half by 1 (i.e., white). If for every range, $r \in \mathcal{R}$, the number of black points inside it is equal to the number of white points, then doubling the number of black points inside a range gives us the exact number of points inside the range. Of course, such a perfect coloring is unachievable in almost all situations. To see this, consider the complete graph K_3 – clearly, in any coloring (by two colors) of its vertices, there must be an edge with two endpoints having the same color (i.e., the edges are the ranges).

Formally, let $\chi : X \rightarrow \{-1, 1\}$ be a coloring. The *discrepancy* of χ over a range r is the amount of imbalance in the coloring inside r . Namely,

$$|\chi(r)| = \left| \sum_{p \in r} \chi(p) \right|.$$

The overall *discrepancy* of χ is $\text{disc}(\chi) = \max_{r \in \mathcal{R}} |\chi(r)|$. The *discrepancy* of a (finite) range space $S = (X, \mathcal{R})$ is the discrepancy of the best possible coloring; namely,

$$\text{disc}(S) = \min_{\chi: X \rightarrow \{-1, +1\}} \text{disc}(\chi).$$

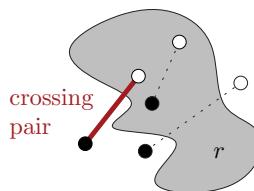
The natural question is, of course, how to compute the coloring χ of minimum discrepancy. This seems like a very challenging question, but when you do not know what to do, you might as well do something random. So, let us pick a random coloring χ of X . To this end, let Π be an arbitrary partition of X into pairs (i.e., a perfect matching). For a pair $\{p, q\} \in \Pi$, we will either color $\chi(p) = -1$ and $\chi(q) = 1$ or the other way around; namely, $\chi(p) = 1$ and $\chi(q) = -1$. We will decide how to color this pair using a single coin flip. Thus, our coloring would be induced by making such a decision for every pair of Π , and let χ be the resulting coloring. We will refer to χ as *compatible* with the partition Π if, for all $\{p, q\} \in \Pi$, we have that $\chi(\{p, q\}) = 0$; namely,

$$\begin{aligned} \forall \{p, q\} \in \Pi \quad & (\chi(p) = +1 \text{ and } \chi(q) = -1) \\ \text{or} \quad & (\chi(p) = -1 \text{ and } \chi(q) = +1). \end{aligned}$$

Consider a range r and a coloring χ compatible with Π .

If a pair $\{p, q\} \in \Pi$ falls completely inside r or completely outside r , then it does not contribute anything to the discrepancy of r . Thus, the only pairs that contribute to the discrepancy of r are the ones that *cross* it. Namely, $\{p, q\} \cap r \neq \emptyset$ and $\{p, q\} \cap (X \setminus r) \neq \emptyset$.

As such, let $\#_r$ denote the *crossing number* of r , that is, the number of pairs that cross r . Next, let $X_i \in \{-1, +1\}$ be the indicator variable which is the contribution of the i th crossing pair to the discrepancy of r . For $\Delta_r = \sqrt{2\#_r \ln(4m)}$, we have by Chernoff's



inequality (Theorem 27.13_{p338}), that

$$\begin{aligned}\Pr[|\chi(\mathbf{r})| \geq \Delta_{\mathbf{r}}] &= \Pr[\chi(\mathbf{r}) \geq \Delta_{\mathbf{r}}] + \Pr[\chi(\mathbf{r}) \leq -\Delta_{\mathbf{r}}] = 2 \Pr\left[\sum_i X_i \geq \Delta_{\mathbf{r}}\right] \\ &\leq 2 \exp\left(-\frac{\Delta_{\mathbf{r}}^2}{2\#\mathbf{r}}\right) = \frac{1}{2m}.\end{aligned}$$

Since there are m ranges in \mathcal{R} , it follows that with good probability (i.e., at least half) for all $\mathbf{r} \in \mathcal{R}$ the discrepancy of \mathbf{r} is at most $\Delta_{\mathbf{r}}$.

Theorem 5.31. *Let $S = (X, \mathcal{R})$ be a range space defined over $n = |X|$ elements with $m = |\mathcal{R}|$ ranges. Consider any partition Π of the elements of X into pairs. Then, with probability $\geq 1/2$, for any range $\mathbf{r} \in \mathcal{R}$, a random coloring $\chi : X \rightarrow \{-1, +1\}$ that is compatible with the partition Π has discrepancy at most*

$$|\chi(\mathbf{r})| < \Delta_{\mathbf{r}} = \sqrt{2\#\mathbf{r} \ln(4m)},$$

where $\#\mathbf{r}$ denotes the number of pairs of Π that cross \mathbf{r} . In particular, since $\#\mathbf{r} \leq |\mathbf{r}|$, we have $|\chi(\mathbf{r})| \leq \sqrt{2|\mathbf{r}| \ln(4m)}$.

Observe that for every range \mathbf{r} we have that $\#\mathbf{r} \leq n/2$, since $2\#\mathbf{r} \leq |X|$. As such, we have:

Corollary 5.32. *Let $S = (X, \mathcal{R})$ be a range space defined over n elements with m ranges. Let Π be an arbitrary partition of X into pairs. Then a random coloring which is compatible with Π has $\text{disc}(\chi) < \sqrt{n \ln(4m)}$, with probability $\geq 1/2$.*

One can easily amplify the probability of success of the coloring by increasing the threshold. In particular, for any constant $c \geq 1$, one has that

$$\forall \mathbf{r} \in \mathcal{R} \quad |\chi(\mathbf{r})| \leq \sqrt{2c\#\mathbf{r} \ln(4m)},$$

with probability $\geq 1 - \frac{2}{(4m)^c}$.

5.4.1. Building ε -sample via discrepancy. Let $S = (X, \mathcal{R})$ be a range space with shattering dimension δ . Let $P \subseteq X$ be a set of n points, and consider the induced range space $S_{|P} = (P, \mathcal{R}_{|P})$; see Definition 5.4_{p62}. Here, by the definition of shattering dimension, we have that $m = |\mathcal{R}_{|P}| = O(n^\delta)$. Without loss of generality, we assume that n is a power of 2. Consider a coloring χ of P with discrepancy bounded by Corollary 5.32. In particular, let Q be the points of P colored by, say, -1 . We know that $|Q| = n/2$, and for any range $\mathbf{r} \in \mathcal{R}$, we have that

$$\chi(\mathbf{r}) = \left| |(P \setminus Q) \cap \mathbf{r}| - |Q \cap \mathbf{r}| \right| < \sqrt{n \ln(4m)} = \sqrt{n \ln O(n^\delta)} \leq c \sqrt{n \ln(n^\delta)},$$

for some absolute constant c . Observe that $|(P \setminus Q) \cap \mathbf{r}| = |P \cap \mathbf{r}| - |Q \cap \mathbf{r}|$. In particular, we have that for any range \mathbf{r} ,

$$(5.4) \quad \left| |P \cap \mathbf{r}| - 2|Q \cap \mathbf{r}| \right| \leq c \sqrt{n \ln(n^\delta)}.$$

Dividing both sides by $n = |P| = 2|Q|$, we have that

$$(5.5) \quad \left| \frac{|P \cap \mathbf{r}|}{|P|} - \frac{|Q \cap \mathbf{r}|}{|Q|} \right| \leq \tau(n) \quad \text{for } \tau(n) = c \sqrt{\frac{\delta \ln n}{n}}.$$

Namely, a coloring with discrepancy bounded by Corollary 5.32 yields a $\tau(n)$ -sample. Intuitively, if n is very large, then Q provides a good approximation to P . However, we

want an ε -sample for a prespecified $\varepsilon > 0$. Conceptually, ε is a fixed constant while $\tau(n)$ is considerably smaller. Namely, Q is a sample which is too tight for our purposes (and thus too big). As such, we will coarsen (and shrink) Q till we get the desired ε -sample by repeated application of Corollary 5.32. Specifically, we can “chain” together several approximations generated by Corollary 5.32. This is sometimes referred to as the *sketch* property of samples. Informally, as testified by the following lemma, a sketch of a sketch is a sketch^⑤.

Lemma 5.33. *Let $Q \subseteq P$ be a ρ -sample for P (in some underlying range space S), and let $R \subseteq Q$ be a ρ' -sample for Q . Then R is a $(\rho + \rho')$ -sample for P .*

PROOF. By definition, we have that, for every $r \in R$,

$$\left| \frac{|r \cap P|}{|P|} - \frac{|r \cap Q|}{|Q|} \right| \leq \rho \quad \text{and} \quad \left| \frac{|r \cap Q|}{|Q|} - \frac{|r \cap R|}{|R|} \right| \leq \rho'.$$

By adding the two inequalities together, we get

$$\left| \frac{|r \cap P|}{|P|} - \frac{|r \cap R|}{|R|} \right| = \left| \frac{|r \cap P|}{|P|} - \frac{|r \cap Q|}{|Q|} + \frac{|r \cap Q|}{|Q|} - \frac{|r \cap R|}{|R|} \right| \leq \rho + \rho'. \quad \blacksquare$$

Thus, let $P_0 = P$ and $P_1 = Q$. Now, in the i th iteration, we will compute a coloring χ_{i-1} of P_{i-1} with low discrepancy, as guaranteed by Corollary 5.32, and let P_i be the points of P_{i-1} colored white by χ_{i-1} . Let $\delta_i = \tau(n_{i-1})$, where $n_{i-1} = |P_{i-1}| = n/2^{i-1}$. By Lemma 5.33, we have that P_k is a $(\sum_{i=1}^k \delta_i)$ -sample for P . Since we would like the smallest set in the sequence P_1, P_2, \dots that is still an ε -sample, we would like to find the maximal k , such that $(\sum_{i=1}^k \delta_i) \leq \varepsilon$. Plugging in the value of δ_i and $\tau(\cdot)$, see (5.5), it is sufficient for our purposes that

$$\sum_{i=1}^k \delta_i = \sum_{i=1}^k \tau(n_{i-1}) = \sum_{i=1}^k c \sqrt{\frac{\delta \ln(n/2^{i-1})}{n/2^{i-1}}} \leq c_1 \sqrt{\frac{\delta \ln(n/2^{k-1})}{n/2^{k-1}}} = c_1 \sqrt{\frac{\delta \ln n_{k-1}}{n_{k-1}}} \leq \varepsilon,$$

since the above series behaves like a geometric series, and as such its total sum is proportional to its largest element^⑥, where c_1 is a sufficiently large constant. This holds for

$$c_1 \sqrt{\frac{\delta \ln n_{k-1}}{n_{k-1}}} \leq \varepsilon \iff c_1^2 \frac{\delta \ln n_{k-1}}{n_{k-1}} \leq \varepsilon^2 \iff \frac{c_1^2 \delta}{\varepsilon^2} \leq \frac{n_{k-1}}{\ln n_{k-1}}.$$

The last inequality holds for $n_{k-1} \geq 2 \frac{c_1^2 \delta}{\varepsilon^2} \ln \frac{c_1^2 \delta}{\varepsilon^2}$, by Lemma 5.13(D). In particular, taking the largest k for which this holds results in a set P_k of size $O((\delta/\varepsilon^2) \ln(\delta/\varepsilon))$ which is an ε -sample for P .

Theorem 5.34 (ε -sample via discrepancy). *For a range space (X, \mathcal{R}) with shattering dimension at most δ and $B \subseteq X$ a finite subset and $\varepsilon > 0$, there exists a subset $C \subseteq B$, of cardinality $O((\delta/\varepsilon^2) \ln(\delta/\varepsilon))$, such that C is an ε -sample for B .*

Note that it is not obvious how to turn Theorem 5.34 into an efficient construction algorithm of such an ε -sample. Nevertheless, this theorem can be turned into a relatively efficient deterministic algorithm using conditional probabilities. In particular, there is a

^⑤Try saying this quickly 100 times.

^⑥Formally, one needs to show that the ratio between two consecutive elements in the series is larger than some constant, say 1.1. This is easy but tedious, but the well-motivated reader (of little faith) might want to do this calculation.

deterministic $O(n^{\delta+1})$ time algorithm for computing an ε -sample for a range space of VC dimension δ and with n points in its ground set using the above approach (see the bibliographical notes in Section 5.7 for details). Inherently, however, it is a far cry from the simplicity of Theorem 5.26 that just requires us to take a random sample. Interestingly, there are cases where using discrepancy leads to smaller ε -samples; again see bibliographical notes for details.

5.4.1.1. Faster deterministic construction of ε -samples. One can speed up the deterministic construction mentioned above by using a sketch-and-merge approach. To this end, we need the following *merge* property of ε -samples. (The proof of the following lemma is quite easy. Nevertheless, we provide the proof in excruciating detail for the sake of completeness.)

Lemma 5.35. *Consider the sets $R \subseteq P$ and $R' \subseteq P'$. Assume that P and P' are disjoint, $|P| = |P'|$, and $|R| = |R'|$. Then, if R is an ε -sample of P and R' is an ε -sample of P' , then $R \cup R'$ is an ε -sample of $P \cup P'$.*

PROOF. We have for any range r that

$$\begin{aligned} \left| \frac{|r \cap (P \cup P')|}{|P \cup P'|} - \frac{|r \cap (R \cup R')|}{|R \cup R'|} \right| &= \left| \frac{|r \cap P|}{|P \cup P'|} + \frac{|r \cap P'|}{|P \cup P'|} - \frac{|r \cap R|}{|R \cup R'|} - \frac{|r \cap R'|}{|R \cup R'|} \right| \\ &= \left| \frac{|r \cap P|}{2|P|} + \frac{|r \cap P'|}{2|P'|} - \frac{|r \cap R|}{2|R|} - \frac{|r \cap R'|}{2|R'|} \right| \\ &= \frac{1}{2} \left| \left(\frac{|r \cap P|}{|P|} - \frac{|r \cap R|}{|R|} \right) + \left(\frac{|r \cap P'|}{|P'|} - \frac{|r \cap R'|}{|R'|} \right) \right| \\ &\leq \frac{1}{2} \left| \frac{|r \cap P|}{|P|} - \frac{|r \cap R|}{|R|} \right| + \frac{1}{2} \left| \frac{|r \cap P'|}{|P'|} - \frac{|r \cap R'|}{|R'|} \right| \\ &\leq \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon. \end{aligned}$$

■

Interestingly, by breaking the given ground sets into sets of equal size and building a balanced binary tree over these sets, one can speed up the deterministic algorithm for building ε -samples. The idea is to compute the sample bottom-up, where at every node we merge the samples provided by the children (i.e., using Lemma 5.35), and then we sketch the resulting set using Lemma 5.33. By carefully fine-tuning this construction, one can get an algorithm for computing ε -samples in time which is near linear in n (assuming ε and δ are small constants). We delegate the details of this construction to Exercise 5.6.

This algorithmic idea is quite useful and we will refer to it as *sketch-and-merge*.

5.4.2. Building ε -net via discrepancy. We are given range space (X, \mathcal{R}) with shattering dimension d and $\varepsilon > 0$ and the target is to compute an ε -net for this range space.

We need to be slightly more careful if we want to use discrepancy to build ε -nets, and we will use Theorem 5.31 instead of Corollary 5.32 in the analysis.

The construction is as before – we set $P_0 = P$, and P_i is all the points colored +1 in the coloring of P_{i-1} by Theorem 5.31. We repeat this till we get a set that is the required net.

To analyze this construction (and decide when it should stop), let r be a range in a given range space (X, \mathcal{R}) with shattering dimension d , and let

$$\nu_i = |P_i \cap r|$$

denote the size of the range \mathbf{r} in the i th set P_i and let $n_i = |\mathsf{P}_i|$, for $i \geq 0$. Observe that the number of points in \mathbf{r} colored by $+1$ and -1 when coloring P_{i-1} is

$$\alpha_i = |\mathsf{P}_i \cap \mathbf{r}| = \nu_i \quad \text{and} \quad \beta_i = |\mathsf{P}_{i-1} \cap \mathbf{r}| - |\mathsf{P}_i \cap \mathbf{r}| = \nu_{i-1} - \nu_i,$$

respectively. As such, setting $m_i = |\mathcal{R}_{\mathsf{P}_i}| = O(n_i^d)$, we have, by Theorem 5.31, that the discrepancy of \mathbf{r} in this coloring of P_{i-1} is

$$|\alpha_i - \beta_i| = |\nu_i - 2\nu_{i-1}| \leq \sqrt{2\nu_{i-1} \ln 4m_{i-1}} \leq c \sqrt{d\nu_{i-1} \ln n_{i-1}}$$

for some constant c , since the crossing number $\#_{\mathbf{r}}$ of a range $\mathbf{r} \cap \mathsf{P}_{i-1}$ is always bounded by its size. This is equivalent to

$$(5.6) \quad |2^{i-1}\nu_{i-1} - 2^i\nu_i| \leq c2^{i-1} \sqrt{d\nu_{i-1} \ln n_{i-1}}.$$

We need the following technical claim that states that the size of ν_k behaves as we expect; as long as the set P_k is large enough, the size of ν_k is roughly $\nu_0/2^k$.

Claim 5.36. *There is a constant c_4 (independent of d), such that for all k with $\nu_0/2^k \geq c_4 d \ln n_k$, $(\nu_0/2^k)/2 \leq \nu_k \leq 2(\nu_0/2^k)$.*

PROOF. The proof is by induction. For $k = 0$ the claim trivially holds. Assume that it holds for $i < k$. Adding up the inequalities of (5.6), for $i = 1, \dots, k$, we have that

$$|\nu_0 - 2^k\nu_k| \leq \sum_{i=1}^k c2^{i-1} \sqrt{d\nu_{i-1} \ln n_{i-1}} \leq \sum_{i=1}^k c2^{i-1} \sqrt{2d \frac{\nu_0}{2^{i-1}} \ln n_{i-1}} \leq c_3 2^k \sqrt{d \frac{\nu_0}{2^k} \ln n_k},$$

for some constant c_3 since this summation behaves like an increasing geometric series and the last term dominates the summation. Thus,

$$\frac{\nu_0}{2^k} - c_3 \sqrt{d \frac{\nu_0}{2^k} \ln n_k} \leq \nu_k \leq \frac{\nu_0}{2^k} + c_3 \sqrt{d \frac{\nu_0}{2^k} \ln n_k}.$$

By assumption, we have that $\sqrt{\frac{\nu_0}{c_4 2^k}} \geq \sqrt{d \ln n_k}$. This implies that

$$\nu_k \leq \frac{\nu_0}{2^k} + c_3 \sqrt{\frac{\nu_0}{2^k} \cdot \frac{\nu_0}{c_4 2^k}} = \frac{\nu_0}{2^k} \left(1 + \frac{c_3}{\sqrt{c_4}}\right) \leq 2 \frac{\nu_0}{2^k},$$

by selecting $c_4 \geq 4c_3^2$. Similarly, we have

$$\nu_k \geq \frac{\nu_0}{2^k} \left(1 - \frac{c_3 \sqrt{d \ln n_k}}{\sqrt{\nu_0/2^k}}\right) \geq \frac{\nu_0}{2^k} \left(1 - \frac{c_3 \sqrt{\nu_0/c_4 2^k}}{\sqrt{\nu_0/2^k}}\right) = \frac{\nu_0}{2^k} \left(1 - \frac{c_3}{\sqrt{c_4}}\right) \geq \frac{\nu_0}{2^k}/2. \quad \blacksquare$$

So consider a “heavy” range \mathbf{r} that contains at least $\nu_0 \geq \varepsilon n$ points of P . To show that P_k is an ε -net, we need to show that $\mathsf{P}_k \cap \mathbf{r} \neq \emptyset$. To apply Claim 5.36, we need a k such that $\varepsilon n/2^k \geq c_4 d \ln n_{k-1}$, or equivalently, such that

$$\frac{2n_k}{\ln(2n_k)} \geq \frac{2c_4 d}{\varepsilon},$$

which holds for $n_k = \Omega\left(\frac{d}{\varepsilon} \ln \frac{d}{\varepsilon}\right)$, by Lemma 5.13(D). But then, by Claim 5.36, we have that

$$\nu_k = |\mathsf{P}_k \cap \mathbf{r}| \geq \frac{|\mathsf{P} \cap \mathbf{r}|}{2 \cdot 2^k} \geq \frac{1}{2} \cdot \frac{\varepsilon n}{2^k} = \frac{\varepsilon}{2} n_k = \Omega\left(d \ln \frac{d}{\varepsilon}\right) > 0.$$

We conclude that the set P_k , which is of size $\Omega\left(\frac{d}{\varepsilon} \ln \frac{d}{\varepsilon}\right)$, is an ε -net for P .

Theorem 5.37 (ε -net via discrepancy). *For any range space (X, \mathcal{R}) with shattering dimension at most d , a finite subset $B \subseteq X$, and $\varepsilon > 0$, there exists a subset $C \subseteq B$, of cardinality $O((d/\varepsilon) \ln(d/\varepsilon))$, such that C is an ε -net for B .*

5.5. Proof of the ε -net theorem

In this section, we finally prove Theorem 5.28.

Let (X, \mathcal{R}) be a range space of VC dimension δ , and let x be a subset of X of cardinality n . Suppose that m satisfies (5.3)_{p71}. Let $N = (x_1, \dots, x_m)$ be the sample obtained by m independent samples from x (the elements of N are not necessarily distinct, and we treat N as an ordered set). Let \mathcal{E}_1 be the probability that N fails to be an ε -net. Namely,

$$\mathcal{E}_1 = \left\{ \exists \mathbf{r} \in \mathcal{R} \mid |\mathbf{r} \cap x| \geq \varepsilon n \text{ and } \mathbf{r} \cap N = \emptyset \right\}.$$

(Namely, there exists a “heavy” range \mathbf{r} that does not contain any point of N .) To complete the proof, we must show that $\Pr[\mathcal{E}_1] \leq \varphi$. Let $T = (y_1, \dots, y_m)$ be another random sample generated in a similar fashion to N . Let \mathcal{E}_2 be the event that N fails but T “works”; formally

$$\mathcal{E}_2 = \left\{ \exists \mathbf{r} \in \mathcal{R} \mid |\mathbf{r} \cap x| \geq \varepsilon n, \mathbf{r} \cap N = \emptyset, \text{ and } |\mathbf{r} \cap T| \geq \frac{\varepsilon m}{2} \right\}.$$

Intuitively, since $\mathbf{E}[|\mathbf{r} \cap T|] \geq \varepsilon m$, we have that for the range \mathbf{r} that N fails for, it follows with “good” probability that $|\mathbf{r} \cap T| \geq \varepsilon m/2$. Namely, \mathcal{E}_1 and \mathcal{E}_2 have more or less the same probability.

Claim 5.38. $\Pr[\mathcal{E}_2] \leq \Pr[\mathcal{E}_1] \leq 2 \Pr[\mathcal{E}_2]$.

PROOF. Clearly, $\mathcal{E}_2 \subseteq \mathcal{E}_1$, and thus $\Pr[\mathcal{E}_2] \leq \Pr[\mathcal{E}_1]$. As for the other part, note that by the definition of conditional probability, we have

$$\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] = \Pr[\mathcal{E}_2 \cap \mathcal{E}_1] / \Pr[\mathcal{E}_1] = \Pr[\mathcal{E}_2] / \Pr[\mathcal{E}_1].$$

It is thus enough to show that $\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1/2$.

Assume that \mathcal{E}_1 occurs. There is $\mathbf{r} \in R$, such that $|\mathbf{r} \cap x| > \varepsilon n$ and $\mathbf{r} \cap N = \emptyset$. The required probability is at least the probability that for this specific \mathbf{r} , we have $|\mathbf{r} \cap T| \geq \frac{\varepsilon n}{2}$. However, $X = |\mathbf{r} \cap T|$ is a binomial variable with expectation $\mathbf{E}[X] = pm$, and variance $\mathbf{V}[X] = p(1-p)m \leq pm$, where $p = |\mathbf{r} \cap x|/n \geq \varepsilon$. Thus, by Chebychev’s inequality (Theorem 27.3)_{p335},

$$\begin{aligned} \Pr[X < \frac{\varepsilon n}{2}] &\leq \Pr[X < \frac{pm}{2}] \leq \Pr\left[|X - pm| > \frac{pm}{2}\right] \\ &= \Pr\left[|X - pm| > \frac{\sqrt{pm}}{2} \sqrt{pm}\right] \leq \Pr\left[|X - \mathbf{E}[X]| > \frac{\sqrt{pm}}{2} \sqrt{\mathbf{V}[X]}\right] \\ &\leq \left(\frac{2}{\sqrt{pm}}\right)^2 \leq \frac{1}{2}, \end{aligned}$$

since $m \geq 8/\varepsilon \geq 8/p$; see (5.3)_{p71}. Thus, for $\mathbf{r} \in \mathcal{E}_1$, we have

$$\frac{\Pr[\mathcal{E}_2]}{\Pr[\mathcal{E}_1]} \geq \Pr\left[|\mathbf{r} \cap T| \geq \frac{\varepsilon n}{2}\right] = 1 - \Pr\left[|\mathbf{r} \cap T| < \frac{\varepsilon n}{2}\right] \geq \frac{1}{2}. \quad \blacksquare$$

Claim 5.38 implies that to bound the probability of \mathcal{E}_1 , it is enough to bound the probability of \mathcal{E}_2 . Let

$$\mathcal{E}'_2 = \left\{ \exists \mathbf{r} \in R \mid \mathbf{r} \cap N = \emptyset, |\mathbf{r} \cap T| \geq \frac{\varepsilon m}{2} \right\}.$$

Clearly, $\mathcal{E}_2 \subseteq \mathcal{E}'_2$. Thus, bounding the probability of \mathcal{E}'_2 is enough to prove Theorem 5.28. Note, however, that a shocking thing happened! We no longer have x participating in our event. Namely, we turned bounding an event that depends on a global quantity (i.e., the ground set x) into bounding a quantity that depends only on a local quantity/experiment (involving only N and T). This is the crucial idea in this proof.

Claim 5.39. $\Pr[\mathcal{E}_2] \leq \Pr[\mathcal{E}'_2] \leq \mathcal{G}_d(2m)2^{-\epsilon m/2}$.

PROOF. We imagine that we sample the elements of $N \cup T$ together, by picking $Z = (z_1, \dots, z_{2m})$ independently from x . Next, we randomly decide the m elements of Z that go into N , and the remaining elements go into T . Clearly,

$$\begin{aligned} \Pr[\mathcal{E}'_2] &= \sum_{z \in X^{2m}} \Pr[\mathcal{E}'_2 \cap (Z = z)] = \sum_{z \in X^{2m}} \frac{\Pr[\mathcal{E}'_2 \cap (Z = z)]}{\Pr[Z = z]} \cdot \Pr[Z = z] \\ &= \sum_z \Pr[\mathcal{E}'_2 | Z = z] \Pr[Z = z] = \mathbf{E}[\Pr[\mathcal{E}'_2 | Z = z]]. \end{aligned}$$

Thus, from this point on, we fix the set Z , and we bound $\Pr[\mathcal{E}'_2 | Z]$. Note that $\Pr[\mathcal{E}'_2]$ is a weighted average of $\Pr[\mathcal{E}'_2 | Z = z]$, and as such a bound on this quantity would imply the same bound on $\Pr[\mathcal{E}'_2]$.

It is now enough to consider the ranges in the projection space $(Z, \mathcal{R}|_Z)$ (which has VC dimension δ). By Lemma 5.9, we have $|\mathcal{R}|_Z| \leq \mathcal{G}_\delta(2m)$.

Let us fix any $\mathbf{r} \in \mathcal{R}|_Z$, and consider the event

$$\mathcal{E}_{\mathbf{r}} = \left\{ \mathbf{r} \cap N = \emptyset \text{ and } |\mathbf{r} \cap T| > \frac{\epsilon m}{2} \right\}.$$

We claim that $\Pr[\mathcal{E}_{\mathbf{r}}] \leq 2^{-\epsilon m/2}$. Observe that if $k = |\mathbf{r} \cap (N \cup T)| \leq \epsilon m/2$, then the event is empty, and this claim trivially holds. Otherwise, $\Pr[\mathcal{E}_{\mathbf{r}}] = \Pr[\mathbf{r} \cap N = \emptyset]$. To bound this probability, observe that we have the $2m$ elements of Z , and we can choose any m of them to be N , as long as none of them is one of the k “forbidden” elements of $\mathbf{r} \cap (N \cup T)$. The probability of that is $\binom{2m-k}{m}/\binom{2m}{m}$. We thus have

$$\begin{aligned} \Pr[\mathcal{E}_{\mathbf{r}}] &\leq \Pr[\mathbf{r} \cap N = \emptyset] = \frac{\binom{2m-k}{m}}{\binom{2m}{m}} = \frac{(2m-k)(2m-k-1)\cdots(m-k+1)}{2m(2m-1)\cdots(m+1)} \\ &= \frac{m(m-1)\cdots(m-k+1)}{2m(2m-1)\cdots(2m-k+1)} \leq 2^{-k} \leq 2^{-\epsilon m/2}. \end{aligned}$$

Thus,

$$\Pr[\mathcal{E}'_2 | Z] = \Pr\left[\bigcup_{\mathbf{r} \in \mathcal{R}|_Z} \mathcal{E}_{\mathbf{r}}\right] \leq \sum_{\mathbf{r} \in \mathcal{R}|_Z} \Pr[\mathcal{E}_{\mathbf{r}}] \leq |\mathcal{R}|_Z 2^{-\epsilon m/2} \leq \mathcal{G}_\delta(2m)2^{-\epsilon m/2},$$

implying that $\Pr[\mathcal{E}'_2] \leq \mathcal{G}_\delta(2m)2^{-\epsilon m/2}$. ■

PROOF OF THEOREM 5.28. By Claim 5.38 and Claim 5.39, we have that $\Pr[\mathcal{E}_1] \leq 2\mathcal{G}_\delta(2m)2^{-\epsilon m/2}$. It thus remains to verify that if m satisfies (5.3), then $2\mathcal{G}_\delta(2m)2^{-\epsilon m/2} \leq \varphi$.

Indeed, we know that $2m \geq 8\delta$ (by (5.3)_{p71}) and by Lemma 5.10, $\mathcal{G}_\delta(2m) \leq 2(2em/\delta)^\delta$, for $\delta \geq 1$. Thus, it is sufficient to show that the inequality $4(2em/\delta)^\delta 2^{-\epsilon m/2} \leq \varphi$ holds. By

rearranging and taking \lg of both sides, we have that this is equivalent to

$$2^{\varepsilon m/2} \geq \frac{4}{\varphi} \left(\frac{2em}{\delta} \right)^\delta \implies \frac{\varepsilon m}{2} \geq \delta \lg \frac{2em}{\delta} + \lg \frac{4}{\varphi}.$$

By our choice of m (see (5.3)), we have that $\varepsilon m/4 \geq \lg(4/\varphi)$. Thus, we need to show that

$$\frac{\varepsilon m}{4} \geq \delta \lg \frac{2em}{\delta}.$$

We verify this inequality for $m = \frac{8\delta}{\varepsilon} \lg \frac{16}{\varepsilon}$ (this would also hold for bigger values, as can be easily verified). Indeed

$$2\delta \lg \frac{16}{\varepsilon} \geq \delta \lg \left(\frac{16e}{\varepsilon} \lg \frac{16}{\varepsilon} \right).$$

This is equivalent to $\left(\frac{16}{\varepsilon} \right)^2 \geq \frac{16e}{\varepsilon} \lg \frac{16}{\varepsilon}$, which is equivalent to $\frac{16}{e\varepsilon} \geq \lg \frac{16}{\varepsilon}$, which is certainly true for $0 < \varepsilon \leq 1$.

This completes the proof of the theorem. \blacksquare

5.6. A better bound on the growth function

In this section, we prove Lemma 5.10_{p65}. Since the proof is straightforward but tedious, the reader can safely skip reading this section.

Lemma 5.40. *For any positive integer n , the following hold.*

- (i) $(1 + 1/n)^n \leq e$.
- (ii) $(1 - 1/n)^{n-1} \geq e^{-1}$.
- (iii) $n! \geq (n/e)^n$.
- (iv) For any $k \leq n$, we have $\binom{n}{k} \leq \binom{ne}{k}$.

PROOF. (i) Indeed, $1 + 1/n \leq \exp(1/n)$, since $1 + x \leq e^x$, for $x \geq 0$. As such $(1 + 1/n)^n \leq \exp(n(1/n)) = e$.

(ii) Rewriting the inequality, we have that we need to prove $\left(\frac{n-1}{n} \right)^{n-1} \geq \frac{1}{e}$. This is equivalent to proving $e \geq \left(\frac{n}{n-1} \right)^{n-1} = \left(1 + \frac{1}{n-1} \right)^{n-1}$, which is our friend from (i).

(iii) Indeed,

$$\frac{n^n}{n!} \leq \sum_{i=0}^{\infty} \frac{n^i}{i!} = e^n,$$

by the Taylor expansion of $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$. This implies that $(n/e)^n \leq n!$, as required.

(iv) Indeed, for any $k \leq n$, we have $\frac{n}{k} \leq \frac{n-1}{k-1}$, as can be easily verified. As such, $\frac{n}{k} \leq \frac{n-i}{k-i}$, for $1 \leq i \leq k-1$. As such,

$$\binom{n}{k} \leq \frac{n}{k} \cdot \frac{n-1}{k-1} \cdots \frac{n-k+1}{1} = \binom{ne}{k}.$$

As for the other direction, by (iii), we have $\binom{n}{k} \leq \frac{n^k}{k!} \leq \frac{n^k}{\left(\frac{k}{e}\right)^k} = \left(\frac{ne}{k}\right)^k$. \blacksquare

Lemma 5.10 restated. *For $n \geq 2\delta$ and $\delta \geq 1$, we have $\left(\frac{n}{\delta}\right)^\delta \leq \mathcal{G}_\delta(n) \leq 2\left(\frac{ne}{\delta}\right)^\delta$, where*

$$\mathcal{G}_\delta(n) = \sum_{i=0}^{\delta} \binom{n}{i}.$$

PROOF. Note that by Lemma 5.40(iv), we have $\mathcal{G}_\delta(n) = \sum_{i=0}^{\delta} \binom{n}{i} \leq 1 + \sum_{i=1}^{\delta} \left(\frac{ne}{i}\right)^i$. This series behaves like a geometric series with constant larger than 2, since

$$\left(\frac{ne}{i}\right)^i / \left(\frac{ne}{i-1}\right)^{i-1} = \frac{ne}{i} \left(\frac{i-1}{i}\right)^{i-1} = \frac{ne}{i} \left(1 - \frac{1}{i}\right)^{i-1} \geq \frac{ne}{i} \frac{1}{e} = \frac{n}{i} \geq \frac{n}{\delta} \geq 2,$$

by Lemma 5.40. As such, this series is bounded by twice the largest element in the series, implying the claim. \blacksquare

5.7. Bibliographical notes

The exposition of the ε -net and ε -sample theorems is roughly based on Alon and Spencer [AS00] and Komlós et al. [KPW92]. In fact, Komlós et al. proved a somewhat stronger bound; that is, a random sample of size $(\delta/\varepsilon) \ln(1/\varepsilon)$ is an ε -net with constant probability. For a proof that shows that in general ε -nets cannot be much smaller in the worst case, see [PA95]. The original proof of the ε -net theorem is due to Haussler and Welzl [HW87]. The proof of the ε -sample theorem is due to Vapnik and Chervonenkis [VC71]. The bound in Theorem 5.26 can be improved to $O\left(\frac{\delta}{\varepsilon^2} + \frac{1}{\varepsilon^2} \log \frac{1}{\varphi}\right)$ [AB99].

An alternative proof of the ε -net theorem proceeds by first computing an $(\varepsilon/4)$ -sample of sufficient size, using the ε -sample theorem (Theorem 5.26_{p71}), and then computing and $\varepsilon/4$ -net for this sample using a direct sample of the right size. It is easy to verify the resulting set is an ε -net. Furthermore, using the “naive” argument (see Section 5.3.2.3) then implies that this holds with the right probability, thus implying the ε -net theorem (the resulting constants might be slightly worse). Exercise 5.3 deploys similar ideas.

The beautiful alternative proof of both theorems via the usage of discrepancy is due to Chazelle and Matoušek [CM96]. The discrepancy method is a beautiful topic which is quite deep mathematically, and we have just skimmed the thin layer of melted water on top of the tip of the iceberg⁷. Two nice books on the topic are the books by Chazelle [Cha01] and Matoušek [Mat99]. The book by Chazelle [Cha01] is currently available online for free from Chazelle’s webpage.

We will revisit discrepancy since in some geometric cases it yields better results than the ε -sample theorem. In particular, the random coloring of Theorem 5.31 can be derandomized using conditional probabilities. One can then use it to get an ε -sample/net by applying it repeatedly. A faster algorithm results from a careful implementation of the sketch-and-merge approach. The disappointing feature of all the deterministic constructions of ε -samples/nets is that their running time is exponential in the dimension δ , since the number of ranges is usually exponential in δ .

A similar result to the one derived by Haussler and Welzl [HW87], using a more geometric approach, was done independently by Clarkson at the same time [Cla87], exposing the fact that VC dimension is not necessary if we are interested only in geometric applications. This was later refined by Clarkson [Cla88], leading to a general technique that, in geometric settings, yields stronger results than the ε -net theorem. This technique has numerous applications in discrete and computational geometry and leads to several “proofs from the book” in discrete geometry.

Exercise 5.5 is from Anthony and Bartlett [AB99].

⁷The iceberg is melting because of global warming; so sorry, climate change.

5.7.1. Variants and extensions. A natural application of the ε -sample theorem is to use it to estimate the weights of ranges. In particular, given a finite range space (X, \mathcal{R}) , we would like to build a data-structure such that we can decide quickly, given a query range r , what the number of points of X inside r is. We could always use a sample of size (roughly) $O(\varepsilon^{-2})$ to get an estimate of the weight of a range, using the ε -sample theorem. The error of the estimate of the size $|r \cap X|$ is $\leq \varepsilon n$, where $n = |X|$; namely, the error is additive. The natural question is whether one can get a multiplicative estimate ρ , such that $|r \cap X| \leq \rho \leq (1 + \varepsilon)|r \cap X|$, where $|r \cap X|$.

In particular, a subset $A \subset X$ is a (relative) **(ε, p) -sample** if for each $r \in \mathcal{R}$ of weight $\geq pn$,

$$\left| \frac{|r \cap A|}{|A|} - \frac{|r \cap X|}{|X|} \right| \leq \varepsilon \frac{|r \cap X|}{|X|}.$$

Of course, one can simply generate an εp -sample of size (roughly) $O(1/(\varepsilon p)^2)$ by the ε -sample theorem. This is not very interesting when $p = 1/\sqrt{n}$. Interestingly, the dependency on p can be improved.

Theorem 5.41 ([LLS01]). *Let (X, \mathcal{R}) be a range space with shattering dimension d , where $|X| = n$, and let $0 < \varepsilon < 1$ and $0 < p < 1$ be given parameters. Then, consider a random sample $A \subseteq X$ of size $\frac{c}{\varepsilon^2 p} \left(d \log \frac{1}{p} + \log \frac{1}{\varphi} \right)$, where c is a constant. Then, it holds that for each range $r \in \mathcal{R}$ of at least pn points, we have*

$$\left| \frac{|r \cap A|}{|A|} - \frac{|r \cap X|}{|X|} \right| \leq \varepsilon \frac{|r \cap X|}{|X|}.$$

In other words, A is a (p, ε) -sample for (X, \mathcal{R}) . The probability of success is $\geq 1 - \varphi$.

A similar result is achievable by using discrepancy; see Exercise 5.7.

5.8. Exercises

Exercise 5.1 (Compute clustering radius). Let C and P be two given sets of points in the plane, such that $k = |C|$ and $n = |P|$. Let $r = \max_{p \in P} \min_{c \in C} \|c - p\|$ be the **covering radius** of P by C (i.e., if we place a disk of radius r centered at each point of C , all those disks cover the points of P).

- (A) Give an $O(n + k \log n)$ expected time algorithm that outputs a number α , such that $r \leq \alpha \leq 10r$.
- (B) For $\varepsilon > 0$ a prescribed parameter, give an $O(n + k\varepsilon^{-2} \log n)$ expected time algorithm that outputs a number α , such that $r \leq \alpha \leq (1 + \varepsilon)r$.

Exercise 5.2 (Some calculus required). Prove Lemma 5.13.

Exercise 5.3 (A direct proof of the ε -sample theorem). For the case that the given range space is finite, one can prove the ε -sample theorem (Theorem 5.26_{p71}) directly. So, we are given a range space $S = (x, \mathcal{R})$ with VC dimension δ , where x is a finite set.

- (A) Show that there exists an ε -sample of S of size $O\left(\delta\varepsilon^{-2} \log \frac{\log|x|}{\varepsilon}\right)$ by extracting an $\varepsilon/3$ -sample from an $\varepsilon/9$ -sample of the original space (i.e., apply Lemma 5.30 twice and use Lemma 5.33).
- (B) Show that for any k , there exists an ε -sample of S of size $O\left(\delta\varepsilon^{-2} \log \frac{\log^{(k)}|x|}{\varepsilon}\right)$.
- (C) Show that there exists an ε -sample of S of size $O\left(\delta\varepsilon^{-2} \log \frac{1}{\varepsilon}\right)$.

Exercise 5.4 (Sauer's lemma is tight). Show that Sauer's lemma (Lemma 5.9) is tight. Specifically, provide a finite range space that has the number of ranges as claimed by Lemma 5.9.

Exercise 5.5 (Flip and flop). (A) Let b_1, \dots, b_{2m} be m binary bits. Let Ψ be the set of all permutations of $1, \dots, 2m$, such that for any $\sigma \in \Psi$, we have $\sigma(i) = i$ or $\sigma(i) = m + i$, for $1 \leq i \leq m$, and similarly, $\sigma(m+i) = i$ or $\sigma(m+i) = m+i$. Namely, $\sigma \in \Psi$ either leaves the pair $i, i+m$ in their positions or it exchanges them, for $1 \leq i \leq m$. As such $|\Psi| = 2^m$.

Prove that for a random $\sigma \in \Psi$, we have

$$\Pr\left[\left|\frac{\sum_{i=1}^m b_{\sigma(i)}}{m} - \frac{\sum_{i=1}^m b_{\sigma(i+m)}}{m}\right| \geq \varepsilon\right] \leq 2e^{-\varepsilon^2 m/2}.$$

(B) Let Ψ' be the set of all permutations of $1, \dots, 2m$. Prove that for a random $\sigma \in \Psi'$, we have

$$\Pr\left[\left|\frac{\sum_{i=1}^m b_{\sigma(i)}}{m} - \frac{\sum_{i=1}^m b_{\sigma(i+m)}}{m}\right| \geq \varepsilon\right] \leq 2e^{-C\varepsilon^2 m/2},$$

where C is an appropriate constant. [Hint: Use (A), but be careful.]

(C) Prove Theorem 5.26 using (B).

Exercise 5.6 (Sketch and merge). Assume that you are given a deterministic algorithm that can compute the discrepancy of Theorem 5.31 in $O(nm)$ time, where n is the size of the ground set and m is the number of induced ranges. We are assuming that the VC dimension δ of the given range space is small and that the algorithm input is only the ground set X (i.e., the algorithm can figure out on its own what the relevant ranges are).

- (A) For a prespecified $\varepsilon > 0$, using the ideas described in Section 5.4.1.1, show how to compute a small ε -sample of X quickly. The running time of your algorithm should be (roughly) $O(n/\varepsilon^{O(\delta)} \text{polylog})$. What is the exact bound on the running time of your algorithm?
- (B) One can slightly improve the running of the above algorithm by more aggressively sketching the sets used. That is, one can add additional sketch layers in the tree. Show how by using such an approach one can improve the running time of the above algorithm by a logarithmic factor.

Exercise 5.7 (Building relative approximations). Prove the following theorem using discrepancy.

Theorem 5.42. Let (X, \mathcal{R}) be a range space with shattering dimension δ , where $|X| = n$, and let $0 < \varepsilon < 1$ and $0 < p < 1$ be given parameters.

Then one can construct a set $N \subseteq X$ of size $O\left(\frac{\delta}{\varepsilon^2 p} \ln \frac{\delta}{\varepsilon p}\right)$, such that, for each range $\mathbf{r} \in \mathcal{R}$ of at least pn points, we have

$$\left| \frac{|\mathbf{r} \cap N|}{|N|} - \frac{|\mathbf{r} \cap X|}{|X|} \right| \leq \varepsilon \frac{|\mathbf{r} \cap X|}{|X|}.$$

In other words, N is a relative (p, ε) -approximation for (X, \mathcal{R}) .

2

Necessary Background

In this section, we briefly overview the background needed for reading the rest of the survey. In Section 2.1, we discuss linear programming and duality. In Section 2.2, we discuss several classical methods for deriving (offline) approximation algorithms for intractable optimization problems. We demonstrate these ideas on the set-cover problem which we later consider in the online setting. In Section 2.3, we provide basic concepts and definitions related to online computation. This section is not meant to give a comprehensive introduction, but rather only provide the basic notation and definitions used later on in the text. For a more comprehensive discussion of these subjects, we refer the reader to the many excellent textbooks on these subjects. For more information on linear programming and duality, we refer the reader to [42]. For further information on approximation techniques we refer the reader to [90]. Finally, for more details on online computation and competitive analysis we refer the reader to [28].

2.1 Linear Programming and Duality

Linear programming is the problem of minimizing or maximizing a linear objective function over a feasible set defined by a set of linear

98 Necessary Background

inequalities. There are several equivalent ways of formulating a linear program. For our purposes, the most convenient one is the following:

$$(P) : \min \sum_{i=1}^n c_i x_i \text{ s.t.}$$

For any $1 \leq j \leq m$:

$$\sum_{i=1}^n a_{ij} x_i \geq b_j, \quad \forall 1 \leq i \leq n, \quad x_i \geq 0.$$

It is well known that any linear program can be formulated in this way. We refer to such a minimization problem as the primal problem (P) . Any vector $x = (x_1, x_2, \dots, x_n)$ that satisfies all the constraints of (P) is referred to as a feasible solution to the linear program (P) . Every primal linear program has a corresponding dual program. The dual linear program of (P) is a maximization linear program: it has m dual variables that correspond to the primal constraints and it has n constraints that correspond to the primal variables. The dual program (D) that corresponds to the linear program formulation (P) is the following:

$$(D) : \max \sum_{j=1}^m b_j y_j \text{ s.t.}$$

For any $1 \leq i \leq n$:

$$\sum_{j=1}^m a_{ij} y_j \leq c_i, \quad \forall 1 \leq j \leq m, \quad y_j \geq 0.$$

The useful properties of the dual program are summarized in the following theorems.

Theorem 2.1 (Weak duality). Let $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_m)$ be feasible solutions to the primal and the dual linear programs, respectively, then:

$$\sum_{i=1}^n c_i x_i \geq \sum_{j=1}^m b_j y_j.$$

Weak duality states that the value of any feasible dual solution is at most the value of any feasible primal solution. Thus, the dual program can actually be used as a lower bound for any feasible primal solution. The proof of this theorem is quite simple.

Proof.

$$\sum_{i=1}^n c_i x_i \geq \sum_{i=1}^n \left(\sum_{j=1}^m a_{ij} y_j \right) x_i \quad (2.1)$$

$$= \sum_{j=1}^m \left(\sum_{i=1}^n a_{ij} x_i \right) y_j \quad (2.2)$$

$$\geq \sum_{j=1}^m b_j y_j, \quad (2.3)$$

where inequality (2.1) follows since $y = (y_1, y_2, \dots, y_m)$ is feasible and each x_i is non-negative. Equality (2.2) follows by changing the order of summation. Inequality (2.3) follows since $x = (x_1, x_2, \dots, x_n)$ is feasible and each y_j is non-negative. \square

The next theorem is sometimes referred to as the strong duality theorem. It states that if the primal and dual programs are bounded, then the optima of the two programs is equal. The proof of the strong duality theorem is harder and we only state the theorem here without a proof.

Theorem 2.2 (Strong duality). The primal linear program is feasible if and only if the dual linear program has a finite optimal solution. In this case, the value of the optimal solutions of the primal and dual programs is equal.

Finally, we prove an important theorem on approximate complementary slackness which is used extensively in the context of approximation algorithms.

Theorem 2.3 (Approximate complementary slackness). Let $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_m)$ be feasible solutions to the

primal and dual linear programs, respectively, satisfying the following conditions:

- *Primal complementary slackness:* For $\alpha \geq 1$, for each i , $1 \leq i \leq n$, if $x_i > 0$ then $c_i/\alpha \leq \sum_{j=1}^m a_{ij}y_j \leq c_i$.
- *Dual complementary slackness:* For $\beta \geq 1$, for each j , $1 \leq j \leq m$, if $y_j > 0$ then $b_j \leq \sum_{i=1}^n a_{ij}x_i \leq b_j \cdot \beta$.

Then

$$\sum_{i=1}^n c_i x_i \leq \alpha \cdot \beta \sum_{j=1}^m b_j y_j.$$

In particular, if the complementary slackness conditions hold with $\alpha = \beta = 1$, then we get that x and y are both optimal solutions to the primal and dual linear programs, respectively. The proof of the theorem is again very short.

Proof.

$$\sum_{i=1}^n c_i x_i \leq \alpha \sum_{i=1}^n \left(\sum_{j=1}^m a_{ij} y_j \right) x_i \quad (2.4)$$

$$= \alpha \sum_{j=1}^m \left(\sum_{i=1}^n a_{ij} x_i \right) y_j \quad (2.5)$$

$$\leq \alpha \cdot \beta \sum_{j=1}^m b_j y_j, \quad (2.6)$$

where (2.4) follows from the primal complementary slackness condition. Equality (2.5) follows by changing the order of summation, and Inequality (2.6) follows from the dual complementary slackness condition. \square

Theorem 2.3 gives an efficient tool for proving that a solution is approximately optimal. Consider, for example, a minimization problem. Suppose you can find primal and dual solutions that satisfy the (approximate) complementary slackness conditions. Then, the solution for the

minimization problem is at most $\alpha \cdot \beta$ times a feasible dual solution. Since, by weak duality (Theorem 2.1), the value of any dual solution is a lower bound on the value of any primal solution, the solution obtained is also at most $\alpha \cdot \beta$ times the optimal primal solution.

Covering/packing linear formulations: A special class of linear programs are those in which the coefficients a_{ij}, b_j and c_i are all non-negative. In this case, the primal program forms a *covering problem* and the dual program forms a *packing problem*. The meaning of these names will become clear in Section 2.2 where we discuss the set cover problem. In the sequel, we sometimes use the notion of *covering-packing* primal-dual pair.

2.2 Approximation Algorithms

In this section, we give a very short background on some basic methods for developing approximation algorithms for intractable problems. Later on we show that similar ideas can be extended and used in the context of online algorithms. We start by formally defining the notions of optimization problems and approximation factors. In an (offline) minimization optimization problem we are given set of instances \mathbb{I} . For each instance $I \in \mathbb{I}$ there is a set of feasible solutions. Each feasible solution is associated with a *cost*. Let $\text{OPT}(I)$ be the cost of the minimal feasible solution for instance I . A polynomial time algorithm A is called a c -approximation for a minimization optimization problem if for every instance I it outputs a solution with cost at most $c \cdot \text{OPT}(I)$. The definitions for maximization optimization problems are analogous. In this case, each instance is associated with a *profit*. A c -approximation algorithm is guaranteed to return a solution with cost at least $\text{OPT}(I)/c$, where $\text{OPT}(I)$ is the maximum profit solution.

The set-cover problem: We demonstrate several classic ideas used for developing approximation algorithms via the set cover problem. In the set-cover problem, we are given a set of n elements $X = e_1, e_2, \dots, e_n$, and a family $\mathbb{S} = s_1, s_2, \dots, s_m$ of subsets of X , $|\mathbb{S}| = m$. Each set s_j is associated with a non-negative cost c_s . A *cover* is a collection of sets such that their union is X . The objective is to find a cover of

X of minimum cost. This problem is known to be NP-hard. Linear programming relaxations constitute a very useful way for obtaining lower bounds on the optimum value of a given combinatorial problem. To this end, we introduce a non-negative variable x_s for each set $s \in \mathbb{S}$. Initially, we formulate the problem as an integer program, allowing x_s to be either 0 or 1. The following is an integer formulation of the set-cover problem:

$$\min \sum_{s \in \mathbb{S}} c_s x_s \text{ s.t.}$$

For each element e_i ($1 \leq i \leq n$):

$$\sum_{s | e_i \in S} x_s \geq 1, \quad \forall s \in \mathbb{S}, \quad x_s \in \{0, 1\}.$$

It is not hard to verify that the set of feasible solution to this program corresponds to the set of feasible covers. Thus, a minimum cover corresponds to an optimal solution to the integer program. Next, we relax the constraint $x_s \in \{0, 1\}$ and get the following linear formulation of the problem:

$$(P) : \min \sum_{s \in \mathbb{S}} c_s x_s \text{ s.t.}$$

For each element e_i ($1 \leq i \leq n$):

$$\sum_{s | e_i \in S} x_s \geq 1, \quad \forall s \in \mathbb{S}, \quad x_s \geq 0.$$

Since the feasible space of the linear formulation contains all the integral solutions of the integer formulation, we get that the optimal solution to the linear formulation is a lower bound on the value of any integral set cover solution. We also remark that there is no need to demand that $x_s \leq 1$, since a minimum cost solution will never increase the value of any x_s above 1. A solution to the linear formulation is called a fractional set cover solution. In such a relaxed set cover solution, one is allowed to take a fraction x_s of each set and pay only $c_s x_s$ for this fraction. The restriction is that the sum of fractions of sets that contain each element e_i should be at least 1. In the corresponding dual

linear program of (P) there is a variable for each element e_i . The dual program (D) is the following:

$$(D) : \max \sum_{e \in X} y_{e_i} \text{ s.t.}$$

For each set $s \in \mathbb{S}$:

$$\sum_{e_i \in S} y_{e_i} \leq c_s, \quad \forall 1 \leq i \leq n, \quad y_{e_i} \geq 0.$$

We next demonstrate several classic approximation techniques using the set cover problem as our running example.

2.2.1 Dual Fitting

We present the dual fitting method by analyzing the greedy algorithm for the set cover problem. The greedy algorithm is depicted below.

Greedy algorithm: Initially, $\mathbb{C} = \emptyset$. Let U be the set of yet uncovered elements.

As long as $U \neq \emptyset$, let $s \in \mathbb{S}$ be the set that minimizes the ratio $c_s/|U \cap s|$:

- (1) Add s to \mathbb{C} and set $x_s \leftarrow 1$.
- (2) For each $e_i \in (U \cap s)$, $y_{e_i} \leftarrow c_s/|U \cap s|$.

Theorem 2.4. The greedy algorithm is an $O(\log n)$ -approximation algorithm for the set-cover problem.

Proof. We are going to show that the algorithm produces throughout its execution both primal and dual solutions. Let P and D be the values of the objective functions of the primal and dual solutions produced, respectively. Initially, $P = D = 0$. We focus on a single iteration of the algorithm and denote by ΔP and ΔD the change in the primal and dual cost, respectively. We prove three simple claims:

- (1) The algorithm produces a primal (covering) feasible solution.
- (2) In each iteration: $\Delta P \leq \Delta D$.

- (3) Each packing constraint in the dual program is violated by a multiplicative factor of at most $O(\log n)$.

The proof follows immediately from the three claims together with weak duality. First, by claim (1) our solution is feasible. By the fact that initially $P = D = 0$ and by claim (2) we get that we produce primal and dual solutions such that $P \leq D$. Finally, by claim (3) we get that dividing the dual solution by $c \log n$, for a constant c , yields a dual feasible solution with value $D' = D/(c \log n)$. Therefore, we get that the primal cost is at most $c \log n$ times a feasible dual solution. Since by weak duality a feasible dual solution is at most the cost of any primal solution, we get that the primal solution is at most $c \log n$ times the optimal (minimal) primal solution.

Proof of (1): Clearly, if there exists a feasible primal solution, then the algorithm will also produce a feasible solution.

Proof of (2): Consider an iteration of the algorithm in which U is the set of uncovered elements and set s is chosen. The change in the primal cost (due to the addition of set s) is c_s . In the dual program, we set the variables corresponding to the $|U \cap s|$ elements covered in the iteration to $c_s/|U \cap s|$. Thus, the total change in the dual profit is also c_s . Note that we only set the dual variable corresponding to each element only once. This happens in the iteration in which the element is covered.

Proof of (3): Consider the dual constraint corresponding to set s . Let e_1, e_2, \dots, e_k be the elements belonging to set s , sorted with respect to the order they were covered by the greedy algorithm. Consider element e_i ; we claim that $y_{e_i} \leq c_s/(k - i + 1)$. This is true since the algorithm chooses the set that minimizes the ratio of cost to number of uncovered elements, which is in fact the value assigned to y_{e_i} . At the time e_i is covered by the greedy algorithm, set s contained at least $k - i + 1$ elements that were still uncovered, thus we get that $y_{e_i} \leq c_s/(k - i + 1)$. Therefore, we get for set s :

$$\sum_{e_i \in s} y_{e_i} \leq \sum_{i=1}^k \frac{c_s}{k - i + 1} = H_k \cdot c_s = c_s \cdot O(\log n),$$

where H_k is the k th harmonic number. \square

2.2.2 Randomized Rounding

In this section, we design a different $O(\log n)$ -approximation algorithm for the set cover problem using a useful technique called *randomized rounding*. The first step is to compute an optimal fractional solution to linear program (P) . Then, the fractional solution is rounded by treating the fractions as probabilities. We describe the rounding algorithm slightly different from standard textbooks; however, this description will be useful in the sequel. The rounding algorithm is the following:

Randomized rounding algorithm:

- (1) For each set $s \in \mathbb{S}$, choose $2 \ln n$ independently random variables $X(s, i)$ uniformly at random in the interval $[0, 1]$.
- (2) For each set s , let $\Theta(s) = \min_{i=1}^{2 \ln n} X(s, i)$.
- (3) Solve linear program (P) .
- (4) Take set s to the cover if $\Theta(s) \leq x_s$.

Theorem 2.5. The algorithm produces a solution with the following properties:

- (1) The expected cost of the solution is $O(\log n)$ times the cost of the fractional solution.
 - (2) The solution is feasible with probability $1 - 1/n > 1/2$.
-

Since the fractional solution provides a lower bound on any integral solution, we get that the algorithm is an $O(\log n)$ -approximation.

Proof. The proof strongly uses the fact that in a feasible solution, for each element, the sum of fractions of the sets containing it is at least 1.

To prove (1) note that for each i , $1 \leq i \leq 2 \ln n$, the probability that $X(s, i) \leq x_s$ is exactly x_s . The probability that s is chosen to the solution is the probability that there exists an i , $1 \leq i \leq 2 \ln n$, such that $X(s, i) \leq x_s$. Let A_i be the event that $X(s, i) \leq x_s$. Thus, the probability that s is chosen to the solution is the probability of $\bigcup_{i=1}^{2 \ln n} A_i$. By the union bound this probability is at most the sum of

the probabilities of the different events, which is $2x_s \ln n$. Therefore, using linearity of expectation the expected cost of the solution is at most $2 \ln n$ times the cost of the fractional solution.

To prove (2) pick an element e . Fix any i , $1 \leq i \leq 2 \ln n$. The probability that e is not covered due to i is the probability that none of the variables $X(s, i)$ (for all sets s , $e \in s$) result in choosing a set s covering e . This probability is

$$\prod_{s \in S | e \in s} (1 - x_s) \leq \exp\left(-\sum_{s \in S | e \in s} x_s\right) \leq \exp(-1),$$

where the first inequality follows since $1 - x \leq \exp(-x)$. The second inequality follows since each element is fractionally covered. Since we choose $2 \ln n$ random variables independently, the probability that e is not covered is at most $\exp(-2 \ln n) = 1/n^2$. Using the union bounds we get that the probability that there exists an element e which is not covered is at most $n \cdot 1/n^2 = 1/n$. \square

2.2.3 The Primal–Dual Schema

In this section, we give a third approximation algorithm for the set cover problem which is based on the primal–dual schema. The algorithm is the following:

Primal–dual algorithm:

While there exists an uncovered element e_i :

- (1) Increase the dual variable y_{e_i} continuously.
- (2) If there exists a set s such that $\sum_{e \in s} y_e = c_s$: take s to the cover and set $x_s \leftarrow 1$.

Let f be the maximum frequency of an element (i.e., the maximum number of sets an element belongs to).

Theorem 2.6. The algorithm is an f -approximation for the set-cover problem.

Proof. Clearly, the primal solution produced by the algorithm is feasible, since we pick sets to the cover as long as the solution is infeasible. The dual solution produced by the algorithm is also feasible, since set s is taken to the solution only when the dual constraint corresponding to it becomes tight. From then on, we never increase a dual variable corresponding to an element belonging to s . Finally, the primal complementary slackness condition holds with $\alpha = 1$, since, if $x_s > 0$, then $\sum_{e \in s} y_{e_i} = c_s$. The dual complementary slackness condition holds with $\beta = f$, since, if $y_{e_i} > 0$, then $1 \leq \sum_{s|e \in s} x_s \leq f$. Thus, by Theorem 2.3, we get that the algorithm is an f -approximation. \square

Remark 2.7. For the special case of the vertex cover problem this algorithm is a 2-approximation.

2.3 Online Computation

The traditional design and analysis of algorithms assumes that complete knowledge of the entire input is available to an algorithm. However, this is not the case in an online problem, where the input is revealed in parts, and an online algorithm is required to respond to each new input upon arrival. Previous decisions of the online algorithm cannot be revoked. Thus, the main issue in online computation is obtaining good performance in the face of uncertainty, since the “future” is unknown to the algorithm. A standard measure by now for evaluating the performance of an online algorithm is the *competitive ratio*, which compares the performance of an online algorithm to that of an offline algorithm which is given the whole input sequence beforehand.

The precise definition of the competitive factor is the following. Suppose we are given a minimization optimization problem I . For each instance of I there is a set of feasible solutions. Each feasible solution is associated with a *cost*. Let $\text{OPT}(I)$ be the optimal cost of a feasible solution for instance I . In the online case, the instance is given to the algorithm in parts. An online algorithm A is said to be c -competitive for I if for every instance of I it outputs a solution of

cost at most $c \cdot \text{OPT}(I) + \alpha$, where the additive term α is independent of the request sequence. if $\alpha = 0$ then the algorithm is called *strictly c-competitive*. We are not going to distinguish between the two notions. The definition of competitiveness of maximization optimization problems is analogous. When considering a maximization problem each instance is associated with a *profit*. A c -competitive algorithm is guaranteed to return a solution with cost at least $\text{OPT}(I)/c - \alpha$, where $\text{OPT}(I)$ is the maximum profit solution, and α is an additive term independent of the request sequence.

A common concept in competitive analysis is that of an *adversary*. The online solution can be viewed as a game between an online algorithm and a malicious adversary. While the online algorithm's strategy is to minimize its cost, the adversary's strategy is to construct the worst possible input for the algorithm. Using this view, the adversary produces a sequence $\sigma = \sigma_1, \sigma_2, \dots$ of requests that define the instance I . A c -competitive online algorithm should then be able to produce a solution of cost no more than c times $\text{OPT}(\sigma)$ for every request sequence σ .

There are several known natural models of adversaries in the context of randomized online algorithms. In this work, we only consider a model in which the adversary knows the online algorithm, as well as the probability distribution used by the online algorithm to make its random decisions. However, the adversary is unaware of the actual random choices made by the algorithm throughout its execution. This kind of adversary is called an *oblivious adversary*. A randomized online algorithm is c -competitive against an oblivious adversary, if for every request sequence σ , the expected cost of the algorithm on σ is at most $c \cdot \text{OPT}(\sigma) + \alpha$. The expectation is taken over all random choices made by the algorithm. Since the oblivious adversary has no information about the actual random choices made by the algorithm, the sequence σ can be constructed ahead of time and $\text{OPT}(\sigma)$ is not a random variable. In the sequel, whenever we have a randomized online algorithm, we simply say that it is c -competitive, and mean that it is c -competitive against an oblivious adversary. Again, the definitions for maximization problems are analogous.

2.4 Notes

The dual-fitting analysis in Section 2.2.1 of the greedy heuristic is due to [78, 41]. The algorithm in Section 2.2.3 is due to Bar-Yehuda and Even [17]. The set-cover problem is a classic NP-hard problem that was studied extensively in the literature. The best approximation factor achievable for it in polynomial time (assuming $P \neq NP$) is $\Theta(\log n)$ [41, 47, 68, 78].

The introduction here is only meant to establish basic notation and terminology for the rest of our discussion. The area of linear programming, duality, approximation algorithms, and online computation have been studied extensively. For more information on linear programming and duality we refer the reader to [42]. For further information on approximation techniques we refer the reader to [90]. Finally, for more details on online computation and competitive analysis we refer the reader to [28].

3

A First Glimpse: The Ski Rental Problem

Let us start with a classic online problem, *ski rental*, through which we will demonstrate the basic ideas underlying the online primal–dual approach. At a ski resort renting skis costs \$1 per day, while buying skis costs $\$B$. A skier arrives at the ski resort for a ski vacation and has to decide whether to rent or buy skis. However, an unknown factor is the number of remaining skiing days that are left before the snow melts. (We should note that this is the skier’s last vacation.) In spite of its apparent simplicity, the ski rental problem captures the essence of online rent or buy dilemmas. The ski rental problem is well understood. There exists a simple deterministic 2-competitive algorithm for the problem and a randomized $e/(e - 1)$ -competitive algorithm. Both results are tight. We show here how to obtain these results using a primal–dual approach.

The first step towards obtaining an online primal–dual algorithm is formulating the problem in hand as a linear program. Since the offline ski rental problem is so simple, casting it as a linear program may seem a bit unnatural. However, this formulation turns out to be very useful. We define an indicator variable x which is set to 1 if the skier buys the skis. For each day j , $1 \leq j \leq k$, we define an indicator variable z_j which

is set to 1 if the skier decides to rent skis on day j . The constraints guarantee that on each day we either rent skis or buy them. This gives us the following integer formulation for the ski rental problem:

$$\min B \cdot x + \sum_{j=1}^k z_j$$

For each day j :

$$x + z_j \geq 1, \quad x \in \{0, 1\}, \quad \forall j, z_j \in \{0, 1\}.$$

We next relax the problem and allow x and each z_j to be in $[0, 1]$. The linear program is given in Figure 3.1 (the primal program). Note that there is always an optimal integral solution, and thus the relaxation has no integrality gap. The dual program is also extremely simple and has a variable y_j corresponding to each day j . We then have a constraint $y_j \leq 1$ that corresponds to each primal variable z_j , and a constraint $\sum_{j=1}^k y_j \leq B$ that corresponds to the primal variable x . Note that the linear programming formulation forms a covering–packing primal–dual pair.

Next, consider the online scenario in which k (the number of ski days) is unknown in advance. This scenario can be captured in the linear formulation in a very natural way. Whenever we have a new ski day, the primal linear program is updated by adding a new covering constraint. The dual program is updated by adding a new dual variable which is added to the packing constraints. The online requirement is that previous decisions cannot be undone. That is, if we already rented skis yesterday, we cannot change this decision today. This requirement is captured in the primal linear program by the restriction that the primal variables are monotonically non-decreasing over time.

Dual (Packing)	Primal (Covering)
Maximize: $\sum_{j=1}^k y_j$	Minimize : $B \cdot x + \sum_{j=1}^k z_j$
Subject to:	Subject to:
$\sum_{j=1}^k y_j \leq B$	For each day j : $x + z_j \geq 1$
For each day j : $0 \leq y_j \leq 1$	$x \geq 0, \forall j, z_j \geq 0$

Fig. 3.1 The fractional ski problem (the primal) and the corresponding dual problem.

Obtaining an optimal deterministic online algorithm for the ski rental problem is very simple. First, rent the skis for the first $B - 1$ days, and then buy them on the B th day. If $k < B$, the algorithm is optimal. If $k \geq B$, then the algorithm spends $\$2B$ dollars, while the optimal solution buys skis on the first day and spends only $\$B$ dollars. Thus, the algorithm is 2-competitive. This simple algorithm and analysis can be obtained in a somewhat less natural way using a primal–dual analysis. On the j th day, a new primal constraint $x + z_j \geq 1$ and a new dual variable y_j arrive. If the primal constraint is already satisfied, then do nothing. Otherwise, increase y_j continuously until some dual constraint becomes tight. Set the corresponding primal variable to be 1. The above algorithm is a simple application of the primal–dual schema, and its analysis is very simple using the approximate complementary slackness conditions: If $y_j > 0$ then $1 \leq x + z_j \leq 2$. Moreover, if $x > 0$ then $\sum_{j=1}^k y_j = B$, and if $z_j > 0$ then $y_j = 1$. Thus, by Theorem 2.3, the algorithm is 2-competitive. It is not hard to see that both of the above algorithms are actually equivalent.

Developing an optimal randomized algorithm is not as straightforward as the deterministic one, yet the primal–dual approach turns out to be very useful towards this end. The first step is designing a deterministic fractional competitive algorithm. Recall that in the fractional case the primal variables can be in the interval $[0, 1]$, and the variables are required to be monotonically non-decreasing over time, during the execution of the algorithm. The online algorithm is the following:

- (1) Initially, $x \leftarrow 0$.
- (2) Each new day (j th new constraint), if $x < 1$:
 - (a) $z_j \leftarrow 1 - x$.
 - (b) $x \leftarrow x(1 + 1/B) + 1/(c \cdot B)$. (The value of c will be determined later.)
 - (c) $y_j \leftarrow 1$.

The analysis is simple. We show that:

- (1) The primal and dual solutions are feasible.
- (2) In each iteration (day), the ratio between the change in the primal and dual objective functions is bounded by $(1 + 1/c)$.

Using the weak duality theorem (Theorem 2.1) we immediately conclude that the algorithm is $(1 + 1/c)$ -competitive.

The proof is very simple. Denote the number of ski days by k . First, since we set $z_j = 1 - x$ (whenever $x < 1$), the primal solution produced is feasible. To show feasibility of the dual solution, we need to show that

$$\sum_{j=1}^k y_j \leq B.$$

We prove this by showing that $x \geq 1$ after at most B days of ski. Denote the increments of x (in each day) by x_1, x_2, \dots, x_k , where $x = \sum_{j=1}^k x_j$. It can be easily seen that x_1, x_2, \dots, x_k is a geometric sequence, defined by $x_1 = 1/(cB)$ and $q = 1 + 1/B$. Thus, after B days,

$$x = \frac{(1 + \frac{1}{B})^B - 1}{c}.$$

Setting $c = (1 + 1/B)^B - 1$ guarantees that $x = 1$ after B days.

Second, if $x < 1$, in each iteration the dual objective function increases by 1, and the increase in the primal objective function is $B\Delta x + z_j = x + 1/c + 1 - x = 1 + 1/c$, and thus the ratio is $(1 + 1/c)$. Concluding, the competitive ratio is $1 + 1/c \approx e/(e - 1)$ for $B \gg 1$.

Converting the fractional solution into a randomized competitive algorithm with the same competitive ratio is easy. We arrange the increments of x on the $[0, 1]$ interval and choose ahead of time (before executing the algorithm) $\alpha \in [0, 1]$ uniformly in random. We are going to buy skis on the day corresponding to the increment of x to which α belongs.

We now analyze the expected cost of the randomized algorithm. Since the probability of buying skis on the j th day is equal to x_j , the expected cost of buying skis is precisely $B \cdot \sum_{j=1}^k x_j = Bx$, which is exactly the first term in the primal objective function. The probability of renting skis on the j th day is equal to the probability of *not* buying skis on or before the j th day, which is $1 - \sum_{i=1}^j x_i$. Since

$$z_j = 1 - \sum_{i=1}^{j-1} x_i \geq 1 - \sum_{i=1}^j x_i,$$

we get that the probability of renting on the j th day is at most z_j , corresponding to the second term in the primal objective function. Thus, by linearity of expectation, for any number of ski days, the expected cost of the randomized algorithm is at most the cost of the fractional solution.

3.1 Notes

The results in this chapter are based on the work of Buchbinder et al. [30] who showed how to obtain a randomized $e/(e - 1)$ -competitive algorithm via the primal–dual approach. The randomized $e/(e - 1)$ -competitive factor for the ski rental problem was originally obtained by Karlin et al. [71]. The role of randomization in the ski problem was later restudied, along with other problems, in [70]. The deterministic 2-competitive algorithm is due to [72].

4

The Basic Approach

We are now ready to extend the ideas used in the previous chapter for the ski rental problem and develop a general recipe for online problems which can be formulated as packing–covering linear programs. In Section 4.1, we formally define a general online framework for packing–covering problems. In Section 4.2, we develop several competitive online algorithms for this framework. In Section 4.3, we prove lower bounds and show that these algorithms are optimal. Finally, in Section 4.4, we give two simple examples that utilize the new ideas developed here.

4.1 The Online Packing–Covering Framework

We are going to define here a general online framework for packing–covering problems. Let us first consider an “offline” covering linear problem. The objective is to minimize the total cost given by a linear cost function $\sum_{i=1}^n c_i x_i$. The feasible solution space is defined by a set of m linear constraints of the form $\sum_{i=1}^n a(i,j)x_i \geq b(j)$, where the entries $a(i,j), b(j)$ and c_i are *non-negative*. For simplicity, we consider in this chapter a simpler setting in which $b(j) = 1$ and $a(i,j) \in \{0, 1\}$. In Section 14, we show how to extend the ideas we present here to handle

general (non-negative) values of $a(i,j)$ and $b(j)$. In the simpler setting, each covering constraint j can be associated with a set $S(j)$ such that $i \in S(j)$ if $a(i,j) = 1$. The j th covering constraint then reduces to simply $\sum_{i \in S(j)} x_i \geq 1$. Any primal covering instance has a corresponding dual packing problem that provides a lower bound on any feasible solution to the instance. A general form of a (simpler) primal covering problem along with its dual packing problem is given in Figure 4.1.

The *online covering problem* is an online version of the covering problem. In the online setting, the cost function is known in advance, but the linear constraints that define the feasible solution space are given to the algorithm one-by-one. In order to maintain a feasible solution to the current set of given constraints, the algorithm is allowed to increase the variables. It may not, however, decrease any previously increased variable. The objective of the algorithm is to minimize the objective function. The reader may already have noticed that this online setting captures the ski rental problem (Chapter 3) as a special case.

The *online packing problem* is an online version of the packing problem. In the online setting, the values c_i ($1 \leq i \leq n$) are known in advance. However, the profit function and the exact packing constraints are not known in advance. In the j th round, a new variable y_j is introduced, along with the set of packing constraints it appears in. The algorithm may increase the value of a variable y_j only in the round in which it is given, and may not decrease or increase the values of any previously given variables. Note that variables that have not yet been introduced may also later appear in the same packing constraints. This actually means that each packing constraint is revealed to the algorithm gradually. The objective of the algorithm is to maximize the objective function while maintaining the feasibility of all packing constraints. Although this online setting seems at first glance a bit

(P): Primal (Covering)	(D): Dual (Packing)
Minimize:	$\sum_{i=1}^n c_i x_i$
subject to:	
$\forall 1 \leq j \leq m: \quad \sum_{i \in S(j)} x_i \geq 1$	$\forall 1 \leq i \leq n: \quad \sum_{j i \in S(j)} y_j \leq c_i$
$\forall 1 \leq i \leq n: \quad x_i \geq 0$	$\forall 1 \leq j \leq m: \quad y_j \geq 0$

Fig. 4.1 Primal (covering) and dual (packing) problems.

unnatural, we later show that many natural online problems reduce to this online setting.

Clearly, at any point of time the linear constraints that have appeared so far define a *sub-instance* of the final covering instance. The dual packing problem of this sub-instance is a *sub-instance* of the final dual packing problem. In the dual packing sub-instance, only part of the dual variables are known, along with their corresponding coefficients. Thus, the two sub-instances derived from the above online settings form a *primal-dual pair*.

The algorithms we propose in the next section maintain at each step solutions for both the primal and dual sub-instances. When a new constraint appears in the online covering problem, our algorithms also consider the new *corresponding* dual variable and its coefficients. Conversely, when a new variable along with its coefficients appear in the online fractional packing problem, our algorithms also consider the *corresponding* new constraint in the primal sub-instance.

4.2 Three Simple Algorithms

In this section, we present three algorithms with the same (optimal) performance guarantees for the online covering/packing problem. Although the performance of all three algorithms in the worst case is the same, their properties do vary, and thus certain algorithms are better suited for particular applications. Also, the ideas underlying each of the algorithms can be extended later to more complex settings. The first algorithm is referred to as the *basic discrete algorithm* and is a direct extension of the algorithm for the ski rental in Section 3. The algorithm is the following:

Algorithm 1:

Upon arrival of a new primal constraint $\sum_{i \in S(j)} x_i \geq 1$ and the corresponding dual variable y_j :

- (1) While $\sum_{i \in S(j)} x_i < 1$:
 - (a) For each $i \in S(j)$: $x_i \leftarrow x_i (1 + 1/c_i) + 1/(|S(j)|c_i)$.
 - (b) $y_j \leftarrow y_j + 1$.

We assume that each $c_i \geq 1$. This assumption is not restrictive and we discuss the reason for that later on. Let $d = \max_j |S(j)| \leq m$ be the maximum “size” of a covering constraint. We prove the following theorem:

Theorem 4.1. The algorithm produces:

- A (fractional) covering solution which is $O(\log d)$ -competitive.
 - An “integral” packing solution which is 2-competitive and violates each packing constraint by at most a factor of $O(\log d)$.
-

We remark that it is possible to obtain a feasible packing solution by scaling the update of each y_j by a factor of $O(\log d)$. This, however, will yield a non-integral packing solution. It is also beneficial for the reader to note the similarity (“in spirit”) between the proof of Theorem 4.1 and the dual fitting based proof of the greedy heuristic for the set cover problem in Section 2.2.1.

Proof. Let P and D be the values of the objective function of the primal and the dual solutions the algorithm produces, respectively. Initially, $P = D = 0$. The dual variables start from zero and increase in increments of one unit, the therefore the dual solution is integral.

Let ΔP and ΔD be the changes in the primal and dual cost, respectively, in a particular iteration of the algorithm in which we execute the inner loop. We prove three simple claims:

- (1) The algorithm produces a primal (covering) feasible solution.
- (2) In each iteration: $\Delta P \leq 2\Delta D$.
- (3) Each packing constraint in the dual program is violated by at most $O(\log d)$.

The proof then follows immediately from the three claims together with weak duality.

Proof of (1): Consider a primal constraint $\sum_{i \in S(j)} x_i \geq 1$. During the j th iteration the algorithm increases the values of the variables x_i until

the constraint is satisfied. Subsequent increases of the variables cannot result in infeasibility.

Proof of (2): Whenever the algorithm updates the primal and dual solutions, the change in the dual profit is 1. The change in the primal cost is

$$\sum_{i \in S(j)} c_i \Delta x_i = \sum_{i \in S(j)} c_i \left(\frac{x_i}{c_i} + \frac{1}{|S(j)|c_i} \right) = \sum_{i \in S(j)} \left(x_i + \frac{1}{|S(j)|} \right) \leq 2,$$

where the final inequality follows since the covering constraint is infeasible at the time of the update.

Proof of (3): Consider any dual constraint $\sum_{j|i \in S(j)} y_j \leq c_i$. Whenever we increase a variable y_j , $i \in S(j)$, by one unit we also increase the variable x_i in line (1a). We prove by simple induction that the variable x_i is bounded from below by the sum of a geometric sequence with $a_1 = 1/(dc_i)$ and $q = (1 + 1/c_i)$. That is,

$$x_i \geq \frac{1}{d} \left(\left(1 + \frac{1}{c_i} \right)^{\sum_{j|i \in S(j)} y_j} - 1 \right). \quad (4.1)$$

Initially, $x_i = 0$, so the induction hypothesis holds. Next, consider an iteration in which variable y_k increases by 1. Let $x_i(\text{start})$ and $x_i(\text{end})$ be the values of variable x_i before and after the increment, respectively. Then,

$$\begin{aligned} x_i(\text{end}) &= x_i(\text{start}) \left(1 + \frac{1}{c_i} \right) + \frac{1}{|S(j)|c_i} \\ &\geq x_i(\text{start}) \left(1 + \frac{1}{c_i} \right) + \frac{1}{d \cdot c_i} \\ &\geq \frac{1}{d} \left(\left(1 + \frac{1}{c_i} \right)^{\sum_{j|i \in S(j) \setminus \{k\}} y_j} - 1 \right) \left(1 + \frac{1}{c_i} \right)^{y_k} + \frac{1}{d \cdot c_i} \\ &= \frac{1}{d} \left(\left(1 + \frac{1}{c_i} \right)^{\sum_{j|i \in S(j)} y_j} - 1 \right). \end{aligned}$$

Note that the inductive hypothesis is applied to $x_i(\text{start})$.

Next, observe that the algorithm never updates any variable $x_i \geq 1$ (since it cannot belong to any unsatisfied constraint). Since each $c_i \geq 1$

and $d \geq 1$, we have that $x_i < 1(1 + 1) + 1 = 3$. Together with inequality (4.1) we get that:

$$3 \geq x_i \geq \frac{1}{d} \left(\left(1 + \frac{1}{c_i} \right)^{\sum_{j|i \in S(j)} y_j} - 1 \right).$$

Using again the fact that $c_i \geq 1$ and simplifying we get the desired result:

$$\sum_{j|i \in S(j)} y_j \leq c_i \log_2(3d + 1) = c_i \cdot O(\log d).$$

□

The basic discrete algorithm is extremely simple and we show in the sequel its many applications. We are now going to derive a slightly different algorithm, which has a continuous flavor and is more in the spirit of the primal–dual schema. This algorithm will also be of guidance for gaining intuition about the right relationship between primal and dual variables. The algorithm is the following:

Algorithm 2:

Upon arrival of a new primal constraint $\sum_{i \in S(j)} x_i \geq 1$ and the corresponding dual variable y_j :

- (1) While $\sum_{i \in S(j)} x_i < 1$:
 - (a) Increase the variable y_j continuously.
 - (b) For each variable x_i that appears in the (yet unsatisfied) primal constraint increase x_i according to the following function:

$$x_i \leftarrow \frac{1}{d} \left[\exp \left(\frac{\ln(1 + d)}{c_i} \sum_{j|i \in S(j)} y_j \right) - 1 \right].$$

Note that the exponential function for variable x_i contains dual variables that correspond to future constraints. However, these variables are all initialized to 0, so they do not contribute to the value of the

function. Although the algorithm is described in a continuous fashion, it is not hard to implement it in a discrete fashion in any desired accuracy. We discuss the intuition of the exponential function we use after proving the following theorem:

Theorem 4.2. The algorithm produces:

- A (fractional) covering solution which is feasible and $O(\log d)$ -competitive.
 - A (fractional) packing solution which is feasible and $O(\log d)$ -competitive.
-

Proof. Let P and D be the values of the objective function of the primal and the dual solutions produced by the algorithm, respectively. Initially, $P = D = 0$. We prove three simple claims:

- (1) The algorithm produces a primal (covering) feasible solution.
- (2) In each iteration j : $\partial P / \partial y_j \leq 2\ln(1 + d) \cdot \partial D / \partial y_j$.
- (3) Each packing constraint in the dual program is feasible.

The theorem then follows immediately from the three claims together with weak duality.

Proof of (1): Consider a primal constraint $\sum_{i \in S(j)} x_i \geq 1$. During the iteration in which the j th primal constraint and dual variable y_j appear, the algorithm increases the values of the variables x_i until the constraint is satisfied. Subsequent increases of variables cannot result in infeasibility.

Proof of (2): Whenever the algorithm updates the primal and dual solutions, $\partial D / \partial y_j = 1$. The derivative of the primal cost is

$$\begin{aligned} \frac{\partial P}{\partial y_j} &= \sum_{i \in S(j)} c_i \frac{\partial x_i}{\partial y_j} \\ &= \sum_{i \in S(j)} c_i \left(\frac{\ln(1 + d)}{c_i} \frac{1}{d} \left[\exp \left(\frac{\ln(1 + d)}{c_i} \sum_{j|i \in S(j)} y_j \right) \right] \right) \end{aligned}$$

$$\begin{aligned}
&= \ln(1+d) \sum_{i \in S(j)} \left(\frac{1}{d} \left[\exp \left(\frac{\ln(1+d)}{c_i} \sum_{j|i \in S(j)} y_j \right) - 1 \right] + \frac{1}{d} \right) \\
&= \ln(1+d) \sum_{i \in S(j)} \left(x_i + \frac{1}{d} \right) \leq 2 \ln(1+d).
\end{aligned} \tag{4.2}$$

The last inequality follows since the covering constraint is infeasible.

Proof of (3): Consider any dual constraint $\sum_{j|i \in S(j)} y_j \leq c_i$. The corresponding variable x_i is always at most 1, since otherwise it cannot belong to any unsatisfied constraint. Thus, we get that:

$$x_i = \frac{1}{d} \left[\exp \left(\frac{\ln(1+d)}{c_i} \sum_{j|i \in S(j)} y_j \right) - 1 \right] \leq 1.$$

Simplifying we get that:

$$\sum_{j|i \in S(j)} y_j \leq c_i.$$

□

Discussion: As can be seen from the proof, the basic discrete algorithm and the continuous algorithm are essentially the same, since $(1 + 1/c_i)$ is approximately $\exp(1/c_i)$. The function in the continuous algorithm is then approximated by Inequality (4.1) in Theorem 4.1. The approximate equality is true as long as c_i is not too small, which is why the assumption that $c_i \geq 1$ is needed in the discrete algorithm. In addition, the discrete algorithm allows the primal variables to exceed the value of 1, which is unnecessary (and can easily be avoided). For these reasons, the proof of the continuous algorithm is a bit simpler. However, in contrast, the description of the discrete algorithm is simpler and more intuitive. Also, in the discrete algorithm, it is not necessary to know the value of d in advance (as long as it is not needed to scale down the dual variables to make the dual solution feasible).

The reader may wonder at this point how did we choose the function used in the algorithm for updating the primal and dual variables. We will try to give here a systematic way of deriving this function. Consider

the point in time in which the j th primal constraint is given and assume that it is not satisfied. Our goal is to bound the derivative of the primal cost (denoted by P) as a function of the dual profit (denoted by D). That is, show that

$$\frac{\partial P}{\partial y_j} = \sum_{i \in S(j)} c_i \frac{\partial x_i}{\partial y_j} \leq \alpha \frac{\partial D}{\partial y_j},$$

where α is going to be the competitive factor. Suppose that the derivative of the primal cost satisfies:

$$\sum_{i \in S(j)} c_i \frac{\partial x_i}{\partial y_j} = A \sum_{i \in S(j)} \left(x_i + \frac{1}{d} \right). \quad (4.3)$$

Then, since $\sum_{i \in S(j)} x_i \leq 1$, $\sum_{i \in S(j)} 1/d \leq 1$, and $\partial D / \partial y_j = 1$, we get that

$$A \sum_{i \in S(j)} \left(x_i + \frac{1}{d} \right) \leq 2A \frac{\partial D}{\partial y_j}.$$

Thus, $\alpha = 2A$. Now, satisfying equality (4.3) requires solving the following differential equation for each $i \in S(j)$:

$$\frac{\partial x_i}{\partial y_j} = \frac{A}{c_i} \left(x_i + \frac{1}{d} \right).$$

It is easy to verify that the solution is a family of functions of the following form:

$$x_i = B \cdot \exp \left(\frac{A}{c_i} \sum_{\ell | i \in S(j)} y_\ell \right) - \frac{1}{d},$$

where B can take on any value. Next, we have the following two boundary conditions on the solution:

- Initially, $x_i = 0$, and this happens when $1/c_i \sum_{j|i \in S(j)} y_j = 0$.
- If $1/c_i \sum_{j|i \in S(j)} y_j = 1$, (i.e., the dual constraint is tight), then $x_i = 1$. (Then, the primal constraint is also satisfied.)

The first boundary condition gives $B = 1/d$. The second boundary condition gives us $A = \ln(d+1)$. Putting everything together we get the exact function used in the algorithm.

We next describe a third algorithm. This algorithm is also continuous, yet different from the previous one. The idea is to utilize the complementary slackness conditions in the online algorithm. This idea turns out to be useful in some applications we discuss in later chapters. Again, let $d = \max_j |S(j)| \leq m$ be the maximum size of a constraint. The description of the algorithm is the following:

Algorithm 3:

Upon arrival of a new primal constraint $\sum_{i \in S(j)} x_i \geq 1$ and the corresponding dual variable y_j :

- (1) While $\sum_{i \in S(j)} x_i < 1$:
 - (a) Increase variable y_j continuously.
 - (b) If $x_i = 0$ and $\sum_{j|i \in S(j)} y_j = c_i$ then set $x_i \leftarrow 1/d$.
 - (c) For each variable x_i , $1/d \leq x_i < 1$, that appears in the (yet unsatisfied) primal constraint, increase x_i according to the following function:

$$x_i \leftarrow \frac{1}{d} \exp \left(\frac{\sum_{j|i \in S(j)} y_j}{c_i} - 1 \right).$$

First, note that the exponential function equals $1/d$ when $\sum_{j|i \in S(j)} y_j = c_i$ and so the algorithm is well defined. We next prove the following theorem:

Theorem 4.3. The algorithm produces:

- A (fractional) $O(\log d)$ -competitive covering solution.
 - A (fractional) 2-competitive packing solution and violates each packing constraint by a factor of at most $O(\log d)$.
-

We remark that similarly to the basic discrete algorithm, it is possible to make the packing solution feasible (and $O(\log d)$ -competitive) by scaling down each y_j by a factor of $O(\log d)$.

Proof. Let P and D be the values of the objective function of the primal and dual solutions, respectively. Initially, $P = D = 0$. We prove three simple claims:

- (1) The algorithm produces a primal (covering) feasible solution.
- (2) Each packing constraint in the dual program is violated by a factor of at most $O(\log d)$.
- (3) $P \leq 2D$.

The theorem then follows immediately from the three claims together with weak duality.

Proof of (1): Consider a primal constraint $\sum_{i \in S(j)} x_i \geq 1$. During the j th iteration the algorithm increases the values of the variables x_i until the constraint is satisfied. Subsequent increases of the variables cannot result in infeasibility.

Proof of (2): Consider any dual constraint $\sum_{j|i \in S(j)} y_j \leq c_i$. The corresponding variable x_i cannot exceed 1, since otherwise it would not belong to any unsatisfied constraint. Thus, we get that:

$$x_i = \frac{1}{d} \exp\left(\frac{\sum_{j|i \in S(j)} y_j}{c_i} - 1\right) \leq 1.$$

Simplifying, we get that

$$\sum_{j|i \in S(j)} y_j \leq c_i(1 + \ln d).$$

Proof of (3): We partition the contribution to the primal objective function into two parts. Let C_1 be the contribution to the primal cost from Step (1b), due to the increase of primal variables from 0 to $1/d$. Let C_2 be the contribution to the primal cost from Step (1c) of the algorithm. It is also beneficial for the reader to observe the similarity between the arguments used for bounding C_1 and those used for the proof of the approximation factor of the primal–dual algorithm in Section 2.2.3.

Bounding C_1 : Let $\tilde{x}_i = \min(x_i, 1/d)$. Our goal is to bound $\sum_{i=1}^n c_i \tilde{x}_i$. To do this we observe the following. First, the algorithm guarantees

that if $x_i > 0$, and therefore $\tilde{x}_i > 0$, then:

$$\sum_{j|i \in S(j)} y_j \geq c_i \quad (\text{primal complementary slackness}) \quad (4.4)$$

Next, if $y_j > 0$, then:

$$\sum_{i \in S(j)} \tilde{x}_i \leq 1 \quad (\text{dual complementary slackness}) \quad (4.5)$$

Inequality (4.5) follows since $\tilde{x}_i \leq 1/d$. Thus, even if for all i , $\tilde{x}_i = 1/d \leq 1/|S(j)|$, then $\sum_{i \in S(j)} \tilde{x}_i \leq 1$. Note that the inequality holds for any y_j , regardless if $y_j > 0$. By the primal and dual complementary slackness conditions we get that:

$$\sum_{i=1}^n c_i \tilde{x}_i \leq \sum_{i=1}^n \left(\sum_{j|i \in S(j)} y_j \right) \tilde{x}_i \quad (4.6)$$

$$= \sum_{j=1}^m \left(\sum_{i \in S(j)} \tilde{x}_i \right) y_j \quad (4.7)$$

$$\leq \sum_{j=1}^m y_j, \quad (4.8)$$

where Inequality (4.6) follows from Inequality (4.4). Equality (4.7) follows by changing the order of summation. Inequality (4.8) follows from Inequality (4.5). Thus, we get that C_1 is at most the dual cost.

Bounding C_2 : Whenever the algorithm updates the primal and dual solutions, $\partial D/\partial y_j = 1$. It is easy to verify that the derivative of the primal cost is:

$$\frac{\partial P}{\partial y_j} = \sum_{i \in S(j)} c_i \frac{\partial x_i}{y_j} = \sum_{i \in S(j)} c_i \frac{x_i}{c_i} \leq 1. \quad (4.9)$$

The last inequality follows since the covering constraint is infeasible at the update time. Thus, C_2 is also bounded from above by the dual cost.

We conclude: $P = C_1 + C_2 \leq 2D$. \square

Remark 4.4. It is possible to even further optimize the performance of Algorithm 3. Initialize each x_i to $\mu = 1/d \ln d$ (instead of $1/d$), and change the continuous update rule to be $x_i \leftarrow \mu \exp(\sum_{j|i \in S(j)} y_j / c_i - 1)$. This will guarantee that $C_1 + C_2 = (1 + 1/\ln d) \cdot D$, while the dual solution is violated by a factor of $1 + \ln d + \ln \ln d$, yielding that the competitive ratio is $\ln d + \ln \ln d + O(1)$. It is also possible to prove a corresponding lower bound of H_d on the competitive ratio of any online algorithm, meaning that the competitive ratio of the algorithm is tight up to additive terms.

4.3 Lower Bounds

In this section, we show that the competitive ratios obtained in Section 4.2 are optimal up to constants. We prove a lower bound for the online packing problem and another lower bound for the online covering problem.

Lemma 4.5. There is an instance of the online (fractional) packing problem with n constraints, such that for any B -competitive online algorithm, there exists a constraint for which

$$\sum_{j|i \in S(j)} y_j \geq c_i \frac{1}{B} \left(1 + \frac{\log_2 n}{2} \right) = c_i \Omega\left(\frac{\log n}{B}\right).$$

Proof. Consider the following instance with $n = 2^k$ packing constraints. The right-hand side of each packing constraint is 1. In the first iteration, a new variable $y(1,1)$ belonging to all constraints arrives. In the second iteration, two variables $y(2,1)$ and $y(2,2)$ arrive. Variable $y(2,1)$ belongs to the first 2^{k-1} constraints and $y(2,2)$ belongs to the last 2^{k-1} constraints. In the third iteration, four dual variables $y(3,1), y(3,2), y(3,3)$, and $y(3,4)$ arrive belonging each to 2^{k-2} packing constraints. The process ends in the $(k+1)$ th iteration in which 2^k variables arrive, each belonging to a single packing constraint. The optimal solution sets the new 2^{i-1} variables in the i th iteration

to 1. Since the algorithm is B -competitive we get the following set of constraints:

For each $1 \leq i \leq k + 1$:

$$\sum_{j=1}^i \sum_{\ell=1}^{2^{j-1}} y(j, \ell) \geq \frac{2^{i-1}}{B}.$$

Multiplying the $(k + 1)$ th inequality by 1 and each inequality i , $1 \leq i \leq k$, by 2^{k-i} and summing up, we get that:

$$\sum_{j=1}^{k+1} 2^{k-j+1} \left(\sum_{\ell=1}^{2^{j-1}} y(j, \ell) \right) \geq \frac{1}{B} (k2^{k-1} + 2^k).$$

This follows since for each j , $\left(\sum_{\ell=1}^{2^{j-1}} y(j, \ell) \right)$ is multiplied by $1 + 1 + 2 + 4 + \dots + 2^{k-j} = 2^{k-j+1}$. However, the left hand side is also the sum over all packing constraints. Thus, by an averaging argument, since there are 2^k constraints, we get that there exists a constraint whose right hand side is at least

$$\frac{1}{2^k \cdot B} (k2^{k-1} + 2^k) = \frac{1}{B} \left(1 + \frac{k}{2} \right).$$

Since $n = 2^k$ we get the desired bound. \square

Lemma 4.6. There is an instance of the online fractional covering problem with n variables such that any online algorithm is $\Omega(\log n)$ -competitive on this instance.

Proof. Consider the following instance with $n = 2^k$ variables x_1, x_2, \dots, x_n . The first constraint that arrives is $\sum_{i=1}^n x_i \geq 1$. If $\sum_{i=1}^{n/2} x_i \geq \sum_{i=n/2+1}^n x_i$, then the next constraint that is given is $\sum_{i=n/2+1}^n x_i \geq 1$, otherwise the constraint $\sum_{i=1}^{n/2} x_i \geq 1$ is given. The process of halving and then continuing with the smaller sum goes on until we remain with a single variable. The optimal offline solution satisfying all the constraints sets the last variable to one. However, for any

online algorithm, the variables in each iteration that do not appear in subsequent iterations add up to at least $1/2$. There are $k + 1$ iterations, and thus the cost of any online algorithm is at least $1 + k/2$, concluding the proof. \square

4.4 Two Warm-Up Problems

In this section, we demonstrate the use of the online primal-dual framework on two simple examples, a covering problem and a packing problem.

4.4.1 The Online Set-Cover Problem

Consider an online version of the offline set-cover problem discussed in Section 2.2. In the online version of the problem, a subset of the elements X arrive one-by-one in an online fashion. The algorithm has to cover each element upon arrival. The restriction is that sets already chosen to the cover by the online algorithm cannot be “unchosen.”

This online setting exactly fits the online covering setting, since whenever a new element arrives a new constraint is added to the set-cover linear formulation. Hence, we can use any of the algorithms presented earlier in Section 4.2 to derive a monotonically increasing fractional solution to the set-cover problem.

Getting a randomized integral solution is extremely simple. We can simply adapt the randomized rounding procedure appearing in Section 2.2.2 to the online setting. Note that the algorithm picks *a priori* uniformly in random a threshold $\Theta(s) \in [0,1]$ for each set s . The algorithm then chooses the set s to the cover if $x_s \geq \Theta(s)$. Since x_s is monotonically increasing, the online algorithm simply chooses the set s to the cover once x_s reaches $\Theta(s)$. Note that the number of elements is not known in advance; however, we do choose $\Theta(s)$ by taking the minimum between $O(\log n)$ random variables. Thus, we can increase the number of random variables as the number of elements increases. The threshold $\Theta(s)$ can only decrease by doing that. The analysis is then straightforward proving that the algorithm produces a solution covering all requested elements with high probability, and its expected cost is $O(\log n)$ times the fractional solution. Since the fractional solution is

$O(\log m)$ -competitive with respect to the optimal solution, we get that the integral algorithm is $O(\log n \log m)$ -competitive.

In Section 5, we show how to obtain a deterministic online algorithm for the set cover problem with the same competitive ratio.

4.4.2 Online Routing

In this section, we give a simple example of an online packing problem. We study the problem of maximizing the throughput of scheduled virtual circuits. In the simplest version of the problem, we are given a graph $G = (V, E)$ with capacities $u(e)$ on the edges. A set of requests $r_i = (s_i, t_i)$ ($1 \leq i \leq n$) arrives in an online fashion. To serve a request, the algorithm chooses a path between s_i and t_i and allocates a bandwidth of one unit on this path. The decisions of the algorithm are irrevocable, and all requests are permanent, meaning that once accepted they stay forever. If the total capacity routed on edge e is $\ell \cdot u(e)$, we say that the *load* on edge e is ℓ . Ideally, the total bandwidth allocated on any edge should not exceed its capacity (load $\ell \leq 1$). The total profit of the algorithm is the number of requests served and as usual the competitive factor is defined with respect to the maximum number of requests that could have been served offline.

In the fractional version of the problem, the allocation is not restricted to an integral bandwidth equal to either 0 or 1; instead, we can allocate to each request a fractional bandwidth in the range $[0, 1]$. In addition, the bandwidth allocated to a request can be divided between several paths. This problem is an online version of the maximum multi-commodity flow problem. We describe the problem as a packing problem in Figure 4.2. For $r_i = (s_i, t_i)$, let $\mathbb{P}(r_i)$ be the set of

Primal	Dual
Minimize: $\sum_{e \in E} u(e)x(e) + \sum_{r_i} z(r_i)$ subject to: $\forall r_i \in \mathbb{R}, P \in \mathbb{P}(r_i): \sum_{e \in P} x(e) + z(r_i) \geq 1$ $\forall r_i, z(r_i) \geq 0, \forall e, x(e) \geq 0$	Maximize: $\sum_{r_i} \sum_{P \in \mathbb{P}(r_i)} f(r_i, P)$ subject to: $\forall r_i \in \mathbb{R}: \sum_{P \in \mathbb{P}(r_i)} f(r_i, P) \leq 1$ $\forall e \in E: \sum_{r_i \in \mathbb{R}, P \in \mathbb{P}(r_i) e \in P} f(r_i, P) \leq u(e)$ $\forall r_i, P: f(r_i, P) \geq 0$

Fig. 4.2 The splittable routing problem (maximization) and its corresponding primal problem.

simple paths between s_i and t_i . For each $P \in \mathbb{P}(r_i)$, the variable $f(r_i, P)$ corresponds to the amount of flow (service) given to request r_i on the path P . The first set of constraints guarantees that each request gets at most a fractional flow (bandwidth) of 1. The second set of constraints follows from the capacity constraints on the edges. In the primal problem, we assign a variable $z(r_i)$ to each request r_i and a variable $x(e)$ to each edge in the graph.

This online setting exactly fits our online packing setting, as new dual variables arrive whenever a new request arrives. However, in each iteration it may happen that an exponential number of variables arrives. We show in the following that we can still overcome this problem and get an efficient algorithm. We present two algorithms for the problem, each having different properties. Let $d \leq n$ be the length of the longest simple path between any two vertices in the graph. The first algorithm is the following:

Routing algorithm 1:

When a new request $r_i = (s_i, t_i)$ arrives:

- (1) if there exists a path $P \in \mathbb{P}(r_i)$ such that $\sum_{e \in P} x(e) < 1$:
 - (a) Route the request on P and set $f(r_i, P) \leftarrow 1$.
 - (b) Set $z(r_i) \leftarrow 1$.
 - (c) For each $e \in P$: $x(e) \leftarrow x(e)(1 + 1/u(e)) + 1/(|P| \cdot u(e))$, where $|P|$ is the length of the path P .

Theorem 4.7. The algorithm is 3-competitive and it violates the capacity of each edge by at most a factor of $O(\log d)$ (i.e., the load on each edge is at most $O(\log d)$).

The proof of Theorem 4.7 is almost identical to the proof of Theorem 4.1, and we leave it as a simple exercise to the reader. The main observation is that the exponential number of dual variables is not obstacle, since the algorithm only needs to check the validity of the condition in line (1). If, for example, $\mathbb{P}(r_i)$ is the set of all simple paths

between s_i and t_i , then this condition can be easily checked by computing a shortest path between s_i and t_i .

The above algorithm may exceed the capacity of the edges. We next show a different algorithm that fully respects the capacities of the edges.

Routing algorithm 2:

Initially: $x(e) \leftarrow 0$.

When a new request $r_i = (s_i, t_i, \mathbb{P}(r_i))$ arrives:

- (1) If there exists a path $P(r_i) \in \mathbb{P}(r_i)$ of length < 1 with respect to $x(e)$:
 - (a) Route the request on “any” path $P \in \mathbb{P}(r_i)$ with length < 1 .
 - (b) $z(r_i) \leftarrow 1$.
 - (c) For each edge e in $P(r_i)$:

$$x(e) \leftarrow x(e) \exp\left(\frac{\ln(1+n)}{u(e)}\right) + \frac{1}{n} \left[\exp\left(\frac{\ln(1+n)}{u(e)}\right) - 1 \right].$$

Theorem 4.8. The algorithm is $O(u(\min)[\exp(\ln(1+n)/u(\min)) - 1])$ -competitive and does not violate the capacity constraints. If $u(\min) \geq \log n$ then the algorithm is $O(\log n)$ -competitive.

Proof. Note first that the function $(u(e)[\exp(\ln(1+n)/u(e)) - 1])$ is monotonically decreasing with respect to $u(e)$. Thus, when a request r_i is routed, the increase of the primal cost is at most:

$$1 + \sum_{e \in P} u(e) \left(x(e) \left[\exp\left(\frac{\ln(1+n)}{u(e)}\right) - 1 \right] + \frac{1}{n} \left[\exp\left(\frac{\ln(1+n)}{u(e)}\right) - 1 \right] \right).$$

This expression is at most

$$2 \left(u(\min) \left[\exp\left(\frac{\ln(1+n)}{u(\min)}\right) - 1 \right] \right) + 1. \quad (4.10)$$

This follows since $z(r_i)$ is set to 1, and edges on the path P satisfy $\sum_{e \in P} x(e) \leq 1$. Each time a request is routed, the dual profit is 1.

Thus, the ratio between the primal and dual solutions is at most Expression (4.10).

Second, observe that the algorithm maintains a feasible primal solution at all times. This follows since $z(r_i)$ is set to 1 for any request for which the distance between s_i and t_i (with respect to the $x(e)$ -variables) is strictly less than 1.

Finally, it remains to prove that the algorithm routes at most $u(e)$ requests on each edge e , and so the dual solution it maintains is feasible. To this end, observe that for each edge e , the value $x(e)$ is the sum of a geometric sequence with initial value $1/n[\exp(\ln(1+n)/u(e)) - 1]$ and a multiplier $q = \exp(\ln(1+n)/u(e))$. Thus, after $u(e)$ requests are routed through edge e , the value of $x(e)$ is

$$\begin{aligned} x(e) &= \frac{1}{n} \left(\exp\left(\frac{\ln(1+n)}{u(e)}\right) - 1 \right) \frac{\exp\left(\frac{u(e)\ln(1+n)}{u(e)}\right) - 1}{\exp\left(\frac{\ln(1+n)}{u(e)}\right) - 1} \\ &= \frac{1}{n} (1 + n - 1) = 1. \end{aligned}$$

Since the algorithm never routes requests on edges for which $x(e) \geq 1$, we are done.

Finally, it is not hard to verify that when $u(\min) \geq \log n$, then

$$2 \left(u(\min) \left[\exp\left(\frac{\ln(1+n)}{u(\min)}\right) - 1 \right] \right) + 1 = O(\log n).$$

□

4.5 Notes

The definitions of the online covering/packing framework along with the basic algorithms in Section 4.2 and the lower bounds in Section 4.3 are based on the work of Buchbinder and Naor [32]. These algorithms draw on ideas from previous algorithms by Alon et al. [3, 4]. The third basic algorithm that incorporates the complementary slackness conditions into the online algorithm is based on the later work of Bansal et al. [14]. The online set-cover problem was considered in [3]. There, they gave a deterministic algorithm for the problem that is discussed later on in Section 5. The second routing algorithm in Section 4.4.2

appeared in [34]. It is basically an alternative description and analysis of a previous algorithm by Awerbuch et al. [11].

There is a long line of work on generating a near-optimal fractional solution for *offline* covering and packing problems, e.g. [52, 53, 54, 55, 75, 79, 84, 93]. All these methods take advantage of the offline nature of the problems. As several of these methods use primal–dual analysis, our approach can be viewed in a sense as an adaptation of these methods to the context of online computation.

5

The Online Set-Cover Problem

In Section 4, we saw how to derive a simple randomized $O(\log m \log n)$ -competitive algorithm for the online set-cover problem. An intriguing question is whether we can obtain a deterministic algorithm for the problem with no degradation in the competitive ratio. A common approach for obtaining deterministic algorithms is *derandomization*, which means converting randomized algorithms into deterministic algorithms. For more details on derandomization methods we refer the reader to [6]. We note that one of the fundamental approaches to (offline) derandomization is the so-called method of conditional expectations or pessimistic estimators [6], which performs a deterministic rounding process. Coming up with a function that guides this process (the pessimistic estimator) is the key ingredient to a successful application of the method of conditional expectations.

Fractional solutions can often be converted into randomized algorithms, but it is usually much harder to perform this conversion online. Indeed, this conversion is possible for the online set-cover problem because of the way the fractional solution evolves in time. Furthermore, the randomized algorithm can be converted into a deterministic one by

an *implicit* use of the method of conditional expectations, which leads to the development of a potential function guiding the online algorithm. The competitive factor of the deterministic algorithm obtained remains $O(\log m \log n)$.

5.1 Obtaining a Deterministic Algorithm

If the set system is not known in advance to the algorithm, then it is quite easy to verify that any deterministic algorithm is $\Omega(n)$ -competitive. Thus, we assume that the universe of elements X is known to the algorithm along with the family of sets \mathbb{S} . It is unknown, however, which subset $X' \subseteq X$ of the elements the algorithm would eventually have to cover (as well as their order of appearance). Let $c(\mathbb{C}_{\text{OPT}})$ denote the cost of the optimal solution. We design an online algorithm that is given a value $\alpha \geq c(\mathbb{C}_{\text{OPT}})$ as input, and produces a feasible solution with cost $O(\alpha \log m \log n)$. However, in case the algorithm is given an infeasible value of α (i.e., $\alpha < c(\mathbb{C}_{\text{OPT}})$), it may fail.

Our algorithm guesses the value of α by doubling. We start by guessing $\alpha = \min_{s \in \mathbb{S}} c_s$, and then run the algorithm with this value. If the algorithm fails we “forget” about all sets chosen so far to \mathbb{C} , update the value of α by doubling it, and continue on. We note that the cost of the sets that we have “forgotten” about can increase the cost of our solution by at most a factor of 2, since the value of α doubles in each step. In the final iteration, the value of α can be at most $2c(\mathbb{C}_{\text{OPT}})$, losing altogether a factor of 4 as a result of the doubling procedure. Also, for each choice of α , our algorithm ignores all sets of cost exceeding α , and chooses all sets of cost at most α/m to \mathbb{C} .

The algorithm we design uses as a subroutine an online algorithm that generates an $O(\log m)$ -competitive fractional solution, e.g., the online fractional algorithm presented in Section 4. This algorithm maintains a monotonically increasing fraction w_s for each set s . Let $w_e = \sum_{s|e \in s} w_s$. Note that the fractional algorithm guarantees that $w_e \geq 1$ for each element e which is requested. Initially, our algorithm starts with the empty cover $\mathbb{C} = \emptyset$. Define C to be the union of all the elements covered by members of \mathbb{C} . The following potential function is

used throughout the algorithm:

$$\Phi = \sum_{e \notin C} n^{2w_e} + n \cdot \exp \left(\frac{1}{2\alpha} \sum_{s \in S} (c_s \chi_C(s) - 3w_s c_s \log n) \right).$$

The function χ_C above is the characteristic function of C , that is, $\chi_C(s) = 1$ if $s \in C$, and $\chi_C(s) = 0$ otherwise.

The deterministic online algorithm is as follows:

Run the algorithm presented in Section 4.2 to produce a monotonically increasing online fractional solution. When the weight of set s is increased:

- (1) If $s \in C$ do nothing; Otherwise:
- (2) Φ_{start} — value of Φ before increasing the weight of s .
- (3) Φ_{end} — value of Φ after increasing the weight of s .
- (4) Φ'_{end} — value of Φ after increasing the weight of s and choosing s to the cover.
- (5) Choose set s to C if $\Phi'_{\text{end}} \leq \Phi_{\text{start}}$.
- (6) If $\Phi_{\text{start}} > \max\{\Phi'_{\text{end}}, \Phi_{\text{end}}\}$, return “FAIL.”

In the following, we analyze the performance of the algorithm in a single iteration in which the condition $\alpha \geq c(C_{\text{OPT}})$ is satisfied. We prove that the algorithm never fails in this iteration.

Lemma 5.1. Consider a step in which the weight of a set s is augmented by the algorithm. Let Φ_{start} and Φ_{end} be the values of the potential function Φ before and after the step, respectively. Then $\Phi_{\text{end}} \leq \Phi_{\text{start}}$. In particular, when $\alpha \geq c(C_{\text{OPT}})$ the algorithm never fails.

Proof. Consider first the case in which $s \in C$. In this case, the first term of the potential function remains unchanged. The second term of the potential function decreases and therefore the lemma holds.

The second case is when $s \notin C$. The proof for this case is probabilistic. We prove that either including s in C , or not including it, does not increase the potential function. Let w_s and $w_s + \delta_s$ denote the weight

of s before and after the increase, respectively. Add set s to \mathbb{C} with probability $1 - n^{-2\delta_s}$. (This is roughly equivalent to choosing s with probability $\delta_s/2$ and repeating it $4 \log n$ times.)

We first bound the expected value of the first term of the potential function. This is similar to the unweighted case. Consider an element $e \in X$ such that $e \notin C$. If $e \notin s$ then the term that corresponds to this element remains unchanged. Otherwise, let the weight of e before and after the step be w_e and $w_e + \delta_s$, respectively. Before the step, element e contributes to the first term of the potential function the value n^{2w_e} . The probability that we do not choose set s containing element e is $n^{-2\delta_s}$. Therefore, the expected contribution of element e to the potential function after the step is at most $n^{-2\delta_s} n^{2(w_e + \delta_s)} = n^{2w_e}$. By linearity of expectation it follows that the expected value of $\sum_{e \notin C} n^{2w_e}$ after the step is precisely its value before the step.

It remains to bound the expected value of the second term of the potential function. Let

$$T = n \cdot \exp \left(\frac{1}{2\alpha} \sum_{s \in \mathbb{S}} (c_s \chi_{\mathbb{C}}(s) - 3w_s c_s \log n) \right)$$

denote the value of the second term of the potential function before the step, and let T' denote the same term after the weight increase and the probabilistic choice of set s to the cover. Recall that $s \notin \mathbb{C}$. Then,

$$\mathbf{Exp}[T'] = T \cdot \exp \left(-\frac{1}{2\alpha} 3\delta_s c_s \log n \right) \cdot \mathbf{Exp} \left[\exp \left(\frac{1}{2\alpha} c_s \chi_{\mathbb{C}'}(s) \right) \right] \quad (5.1)$$

where $\chi_{\mathbb{C}'}(s) = 1$ is the indicator of the event that set s is chosen to the cover, which happens with probability $1 - n^{-2\delta_s}$. We bound the right-hand side of (5.1) as follows:

$$\mathbf{Exp} \left[\exp \left(\frac{1}{2\alpha} c_s \chi_{\mathbb{C}'}(s) \right) \right] = n^{-2\delta_s} + (1 - n^{-2\delta_s}) \cdot \exp \left(\frac{c_s}{2\alpha} \right) \quad (5.2)$$

$$\leq 1 - 2\delta_s \log n + 2\delta_s \log n \exp \left(\frac{c_s}{2\alpha} \right) \quad (5.3)$$

$$= 1 + 2\delta_s \log n \left(\exp \left(\frac{c_s}{2\alpha} \right) - 1 \right) \quad (5.4)$$

$$\leq 1 + 2\delta_s \log n \frac{3c_s}{4\alpha} \quad (5.5)$$

$$\leq \exp \left(\frac{3\delta_s c_s \log n}{2\alpha} \right). \quad (5.6)$$

Here, (5.3) follows since for all $x \geq 0$ and $z \geq 1$, $e^{-x} + (1 - e^{-x}) \cdot z \leq 1 - x + x \cdot z$, (5.5) follows¹ since $e^y - 1 \leq 3y/2$ for all $0 \leq y \leq 1/2$, and (5.6) follows since $1 + x \leq e^x$ for all $x \geq 0$. Plugging in (5.1), we conclude that the expected value of the second term after the increase step and the probabilistic choices is at most

$$\mathbf{Exp}[T'] = T \cdot \exp \left(-\frac{1}{2\alpha} 3\delta_s c_s \log n \right) \cdot \exp \left(\frac{1}{2\alpha} 3\delta_s c_s \log n \right) \leq T.$$

By linearity of expectation it now follows that $\mathbf{Exp}[\Phi_{\text{end}}] \leq \Phi_{\text{start}}$. Therefore, either the event of choosing s to the cover, or the event of not choosing s to the cover, does not increase the potential function. We conclude that after each step $\Phi_{\text{end}} \leq \Phi_{\text{start}}$. \square

Theorem 5.2. Throughout the algorithm, the following holds:

- (i) Every $e \in X$ of weight $w_e \geq 1$ is covered, that is, $e \in C$.
- (ii) $\sum_{s \in \mathbb{C}} c_s = \alpha \cdot O(\log m \log n)$.

Proof. Initially, the value of the potential function Φ is at most $n \cdot n^0 + n < n^2$, and hence it remains smaller than n^2 throughout the whole algorithm. Therefore, if for element e , $w_e \geq 1$, at some point of time, then $e \in C$, since otherwise the contribution of the term n^{2w_e} itself would be at least n^2 . This proves part (i). To prove part (ii), note that by the same argument, throughout the algorithm

$$n \cdot \exp \left(\frac{1}{2\alpha} \sum_{s \in \mathbb{S}} c_s \chi_{\mathbb{C}}(s) - 3w_s c_s \log n \right) < n^2.$$

¹Note that here we use the fact that $\alpha \geq c(\mathbb{C}_{\text{OPT}})$, since we ignored all sets with cost greater than α .

Therefore,

$$\sum_{s \in \mathbb{S}} c_s \chi_{\mathbb{C}}(s) \leq \sum_{s \in \mathbb{S}} 3w_s c_s \log n + 2\alpha \log n,$$

and the desired result follows from the fact that the fractional solution is $O(\log m)$ -competitive. \square

5.2 Notes

The results in this section are based on the work of Alon et al. [3]. We note that the algorithms of [3] were not originally stated as primal–dual algorithms, yet interpreting them as primal–dual algorithms was the starting point of extending the primal–dual method to the realm of online computation. Alon et al. [3] also proved that any deterministic algorithm for the online set-cover problem is $\Omega(\log n \log m / (\log \log m + \log \log n))$ -competitive for many values of m and n . In the unweighted version of the set-cover problem all sets are of unit cost and so the goal is to minimize the number of sets needed for covering the elements. Buchbinder and Naor [32] used the improved offline rounding technique of [89] to obtain an $O(\log d \log(n/\text{OPT}))$ -competitive algorithm, where d is the maximum frequency of an element (i.e., the maximum number of sets an element belongs to), n is the number of elements and OPT is the optimal size of the set cover.

Derandomization has proved to be useful for other online problems as well. Buchbinder and Naor [32], using derandomization methods, obtained an alternative routing algorithm that achieves the same competitiveness as the second routing algorithm in Section 4.4.2. The algorithm is based on the basic algorithms of Section 4.2 along with an online derandomization of the rounding method in [85, 86]. Another example of a derandomization of an online algorithm is by Buchbinder et al. [30] for the ad-auctions problem (see also Section 10). However, we note that we do not yet fully understand when derandomization methods can be applied in online settings. For example, for the online group Steiner problem on trees discussed in Section 11, there is currently only a randomized online algorithm, even though the offline randomized algorithm can be derandomized.

Another online variation of the set cover problem was considered in [10]. There, we are also given m sets and n elements that arrive one at a time. However, the goal of the online algorithm is to pick k sets so as to maximize the number of elements that are covered. The algorithm only gets credit for elements that are contained in a set that it selected *before* or *during* the step in which the element arrived. The authors of [10] showed a *randomized* $\Theta(\log m \log(n/k))$ competitive algorithm for the problem, where the bound is optimal for many values of n , m , and k . A different extension of the online set cover problem is studied in [5]. They considered an admission control problem where the goal is to minimize the number of rejections. The problem is solved by reducing it to an instance of online set cover with repetitions, i.e., each element may need to be covered several times.

sets of *equal* size so that the weight of the cut between the two sets is as large as possible (i.e., the total weight of edges with one end in each set is as large as possible). Note that the restriction that the graph is a tree is crucial here, but the assumption that the tree is binary is not. The problem is NP-hard in general graphs.

Notes and Further Reading

The first topic in this chapter, on how to avoid a running time of $O(kn^{k+1})$ for Vertex Cover, is an example of the general theme of *parameterized complexity*: for problems with two such “size parameters” n and k , one generally prefers running times of the form $O(f(k) \cdot p(n))$, where $p(\cdot)$ is a polynomial, rather than running times of the form $O(n^k)$. A body of work has grown up around this issue, including a methodology for identifying NP-complete problems that are unlikely to allow for such improved running times. This area is covered in the book by Downey and Fellows (1999).

The problem of coloring a collection of circular arcs was shown to be NP-complete by Garey, Johnson, Miller, and Papadimitriou (1980). They also described how the algorithm presented in this chapter follows directly from a construction due to Tucker (1975). Both Interval Coloring and Circular-Arc Coloring belong to the following class of problems: Take a collection of geometric objects (such as intervals or arcs), define a graph by joining pairs of objects that intersect, and study the problem of coloring this graph. The book on graph coloring by Jensen and Toft (1995) includes descriptions of a number of other problems in this style.

The importance of tree decompositions and tree-width was brought into prominence largely through the work of Robertson and Seymour (1990). The algorithm for constructing a tree decomposition described in Section 10.5 is due to Diestel et al. (1999). Further discussion of tree-width and its role in both algorithms and graph theory can be found in the survey by Reed (1997) and the book by Diestel (2000). Tree-width has also come to play an important role in inference algorithms for probabilistic models in machine learning (Jordan 1998).

Notes on the Exercises Exercise 2 is based on a result of Uwe Schöning; and Exercise 8 is based on a problem we learned from Amit Kumar.

Chapter 11

Approximation Algorithms

Following our encounter with NP-completeness and the idea of computational intractability in general, we've been dealing with a fundamental question: How should we design algorithms for problems where polynomial time is probably an unattainable goal?

In this chapter, we focus on a new theme related to this question: *approximation algorithms*, which run in polynomial time and find solutions that are guaranteed to be close to optimal. There are two key words to notice in this definition: *close* and *guaranteed*. We will not be seeking the optimal solution, and as a result, it becomes feasible to aim for a polynomial running time. At the same time, we will be interested in proving that our algorithms find solutions that are guaranteed to be close to the optimum. There is something inherently tricky in trying to do this: In order to prove an approximation guarantee, we need to compare our solution with—and hence reason about—an optimal solution that is computationally very hard to find. This difficulty will be a recurring issue in the analysis of the algorithms in this chapter.

We will consider four general techniques for designing approximation algorithms. We start with *greedy algorithms*, analogous to the kind of algorithms we developed in Chapter 4. These algorithms will be simple and fast, as in Chapter 4, with the challenge being to find a greedy rule that leads to solutions provably close to optimal. The second general approach we pursue is the *pricing method*. This approach is motivated by an economic perspective; we will consider a price one has to pay to enforce each constraint of the problem. For example, in a graph problem, we can think of the nodes or edges of the graph sharing the cost of the solution in some equitable way. The pricing method is often referred to as the *primal-dual technique*, a term inherited from

the study of linear programming, which can also be used to motivate this approach. Our presentation of the pricing method here will not assume familiarity with linear programming. We will introduce linear programming through our third technique in this chapter, *linear programming and rounding*, in which one exploits the relationship between the computational feasibility of linear programming and the expressive power of its more difficult cousin, *integer programming*. Finally, we will describe a technique that can lead to extremely good approximations: using dynamic programming on a rounded version of the input.

11.1 Greedy Algorithms and Bounds on the Optimum: A Load Balancing Problem

As our first topic in this chapter, we consider a fundamental *Load Balancing Problem* that arises when multiple servers need to process a set of jobs or requests. We focus on a basic version of the problem in which all servers are identical, and each can be used to serve any of the requests. This simple problem is useful for illustrating some of the basic issues that one needs to deal with in designing and analyzing approximation algorithms, particularly the task of comparing an approximate solution with an optimum solution that we cannot compute efficiently. Moreover, we'll see that the general issue of load balancing is a problem with many facets, and we'll explore some of these in later sections.

The Problem

We formulate the Load Balancing Problem as follows. We are given a set of m machines M_1, \dots, M_m and a set of n jobs; each job j has a processing time t_j . We seek to assign each job to one of the machines so that the loads placed on all machines are as “balanced” as possible.

More concretely, in any assignment of jobs to machines, we can let $A(i)$ denote the set of jobs assigned to machine M_i ; under this assignment, machine M_i needs to work for a total time of

$$T_i = \sum_{j \in A(i)} t_j,$$

and we declare this to be the *load* on machine M_i . We seek to minimize a quantity known as the *makespan*; it is simply the maximum load on any machine, $T = \max_i T_i$. Although we will not prove this, the scheduling problem of finding an assignment of minimum makespan is NP-hard.

Designing the Algorithm

We first consider a very simple greedy algorithm for the problem. The algorithm makes one pass through the jobs in any order; when it comes to job j , it assigns j to the machine whose load is smallest so far.

```

Greedy-Balance:
Start with no jobs assigned
Set  $T_i = 0$  and  $A(i) = \emptyset$  for all machines  $M_i$ 
For  $j = 1, \dots, n$ 
  Let  $M_i$  be a machine that achieves the minimum  $\min_k T_k$ 
  Assign job  $j$  to machine  $M_i$ 
  Set  $A(i) \leftarrow A(i) \cup \{j\}$ 
  Set  $T_i \leftarrow T_i + t_j$ 
EndFor

```

For example, Figure 11.1 shows the result of running this greedy algorithm on a sequence of six jobs with sizes 2, 3, 4, 6, 2, 2; the resulting makespan is 8, the “height” of the jobs on the first machine. Note that this is not the optimal solution; had the jobs arrived in a different order, so that the algorithm saw the sequence of sizes 6, 4, 3, 2, 2, 2, then it would have produced an allocation with a makespan of 7.

Analyzing the Algorithm

Let T denote the makespan of the resulting assignment; we want to show that T is not much larger than the minimum possible makespan T^* . Of course, in trying to do this, we immediately encounter the basic problem mentioned above: We need to compare our solution to the optimal value T^* , even though we don’t know what this value is and have no hope of computing it. For the

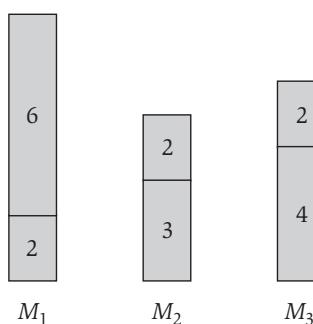


Figure 11.1 The result of running the greedy load balancing algorithm on three machines with job sizes 2, 3, 4, 6, 2, 2.

analysis, therefore, we will need a *lower bound* on the optimum—a quantity with the guarantee that no matter how good the optimum is, it cannot be less than this bound.

There are many possible lower bounds on the optimum. One idea for a lower bound is based on considering the total processing time $\sum_j t_j$. One of the m machines must do at least a $1/m$ fraction of the total work, and so we have the following.

(11.1) *The optimal makespan is at least*

$$T^* \geq \frac{1}{m} \sum_j t_j.$$

There is a particular kind of case in which this lower bound is much too weak to be useful. Suppose we have one job that is extremely long relative to the sum of all processing times. In a sufficiently extreme version of this, the optimal solution will place this job on a machine by itself, and it will be the last one to finish. In such a case, our greedy algorithm would actually produce the optimal solution; but the lower bound in (11.1) isn't strong enough to establish this.

This suggests the following additional lower bound on T^* .

(11.2) *The optimal makespan is at least $T^* \geq \max_j t_j$.*

Now we are ready to evaluate the assignment obtained by our greedy algorithm.

(11.3) *Algorithm Greedy-Balance produces an assignment of jobs to machines with makespan $T \leq 2T^*$.*

Proof. Here is the overall plan for the proof. In analyzing an approximation algorithm, one compares the solution obtained to what one knows about the optimum—in this case, our lower bounds (11.1) and (11.2). We consider a machine M_i that attains the maximum load T in our assignment, and we ask: What was the last job j to be placed on M_i ? If t_j is not too large relative to most of the other jobs, then we are not too far above the lower bound (11.1). And, if t_j is a very large job, then we can use (11.2). Figure 11.2 shows the structure of this argument.

Here is how we can make this precise. When we assigned job j to M_i , the machine M_i had the smallest load of any machine; this is the key property of our greedy algorithm. Its load just before this assignment was $T_i - t_j$, and since this was the smallest load at that moment, it follows that every machine

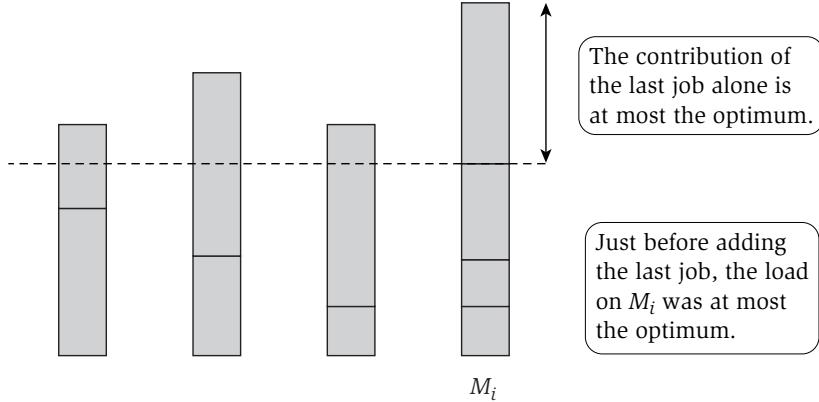


Figure 11.2 Accounting for the load on machine M_i in two parts: the last job to be added, and all the others.

had load at least $T_i - t_j$. Thus, adding up the loads of all machines, we have $\sum_k T_k \geq m(T_i - t_j)$, or equivalently,

$$T_i - t_j \leq \frac{1}{m} \sum_k T_k.$$

But the value $\sum_k T_k$ is just the total load of all jobs $\sum_j t_j$ (since every job is assigned to exactly one machine), and so the quantity on the right-hand side of this inequality is exactly our lower bound on the optimal value, from (11.1). Thus

$$T_i - t_j \leq T^*.$$

Now we account for the remaining part of the load on M_i , which is just the final job j . Here we simply use the other lower bound we have, (11.2), which says that $t_j \leq T^*$. Adding up these two inequalities, we see that

$$T_i = (T_i - t_j) + t_j \leq 2T^*.$$

Since our makespan T is equal to T_i , this is the result we want. ■

It is not hard to give an example in which the solution is indeed close to a factor of 2 away from optimal. Suppose we have m machines and $n = m(m - 1) + 1$ jobs. The first $m(m - 1) = n - 1$ jobs each require time $t_j = 1$. The last job is much larger; it requires time $t_n = m$. What does our greedy algorithm do with this sequence of jobs? It evenly balances the first $n - 1$ jobs, and then has to add the giant job n to one of them; the resulting makespan is $T = 2m - 1$.

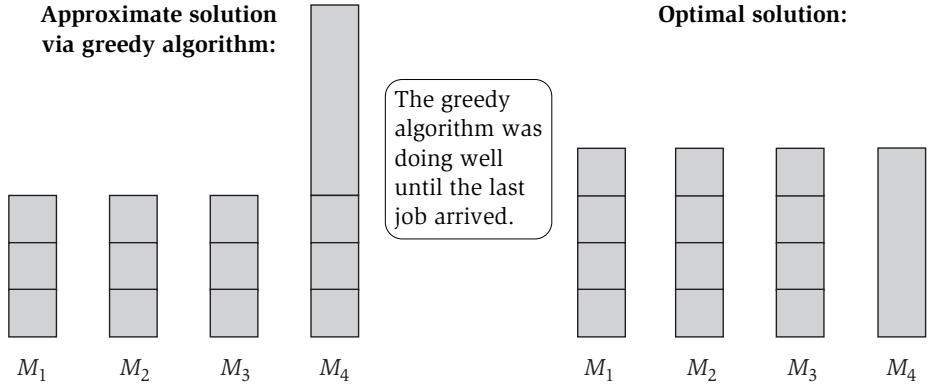


Figure 11.3 A bad example for the greedy balancing algorithm with $m = 4$.

What does the optimal solution look like in this example? It assigns the large job to one of the machines, say, M_1 , and evenly spreads the remaining jobs over the other $m - 1$ machines. This results in a makespan of m . Thus the ratio between the greedy algorithm's solution and the optimal solution is $(2m - 1)/m = 2 - 1/m$, which is close to a factor of 2 when m is large.

See Figure 11.3 for a picture of this with $m = 4$; one has to admire the perversity of the construction, which misleads the greedy algorithm into perfectly balancing everything, only to mess everything up with the final giant item.

In fact, with a little care, one can improve the analysis in (11.3) to show that the greedy algorithm with m machines is within exactly this factor of $2 - 1/m$ on every instance; the example above is really as bad as possible.

Extensions: An Improved Approximation Algorithm

Now let's think about how we might develop a better approximation algorithm—in other words, one for which we are always guaranteed to be within a factor strictly smaller than 2 away from the optimum. To do this, it helps to think about the worst cases for our current approximation algorithm. Our earlier bad example had the following flavor: We spread everything out very evenly across the machines, and then one last, giant, unfortunate job arrived. Intuitively, it looks like it would help to get the largest jobs arranged nicely first, with the idea that later, small jobs can only do so much damage. And in fact, this idea does lead to a measurable improvement.

Thus we now analyze the variant of the greedy algorithm that first sorts the jobs in decreasing order of processing time and then proceeds as before.

We will prove that the resulting assignment has a makespan that is at most 1.5 times the optimum.

```

Sorted-Balance:
Start with no jobs assigned
Set  $T_i = 0$  and  $A(i) = \emptyset$  for all machines  $M_i$ 
Sort jobs in decreasing order of processing times  $t_j$ 
Assume that  $t_1 \geq t_2 \geq \dots \geq t_n$ 
For  $j = 1, \dots, n$ 
  Let  $M_i$  be the machine that achieves the minimum  $\min_k T_k$ 
  Assign job  $j$  to machine  $M_i$ 
  Set  $A(i) \leftarrow A(i) \cup \{j\}$ 
  Set  $T_i \leftarrow T_i + t_j$ 
EndFor

```

The improvement comes from the following observation. If we have fewer than m jobs, then the greedy solution will clearly be optimal, since it puts each job on its own machine. And if we have more than m jobs, then we can use the following further lower bound on the optimum.

(11.4) *If there are more than m jobs, then $T^* \geq 2t_{m+1}$.*

Proof. Consider only the first $m + 1$ jobs in the sorted order. They each take at least t_{m+1} time. There are $m + 1$ jobs and only m machines, so there must be a machine that gets assigned two of these jobs. This machine will have processing time at least $2t_{m+1}$. ■

(11.5) *Algorithm Sorted-Balance produces an assignment of jobs to machines with makespan $T \leq \frac{3}{2}T^*$.*

Proof. The proof will be very similar to the analysis of the previous algorithm. As before, we will consider a machine M_i that has the maximum load. If M_i only holds a single job, then the schedule is optimal.

So let's assume that machine M_i has at least two jobs, and let t_j be the last job assigned to the machine. Note that $j \geq m + 1$, since the algorithm will assign the first m jobs to m distinct machines. Thus $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$, where the second inequality is (11.4).

We now proceed as in the proof of (11.3), with the following single change. At the end of that proof, we had inequalities $T_i - t_j \leq T^*$ and $t_j \leq T^*$, and we added them up to get the factor of 2. But in our case here, the second of these

inequalities is, in fact, $t_j \leq \frac{1}{2}T^*$; so adding the two inequalities gives us the bound

$$T_i \leq \frac{3}{2}T^*. \blacksquare$$

11.2 The Center Selection Problem

Like the problem in the previous section, the Center Selection Problem, which we consider here, also relates to the general task of allocating work across multiple servers. The issue at the heart of Center Selection is where best to place the servers; in order to keep the formulation clean and simple, we will not incorporate the notion of load balancing into the problem. The Center Selection Problem also provides an example of a case in which the most natural greedy algorithm can result in an arbitrarily bad solution, but a slightly different greedy method is guaranteed to always result in a near-optimal solution.

The Problem

Consider the following scenario. We have a set S of n sites—say, n little towns in upstate New York. We want to select k centers for building large shopping malls. We expect that people in each of these n towns will shop at one of the malls, and so we want to select the sites of the k malls to be central.

Let us start by defining the input to our problem more formally. We are given an integer k , a set S of n sites (corresponding to the towns), and a distance function. When we consider instances where the sites are points in the plane, the distance function will be the standard Euclidean distance between points, and any point in the plane is an option for placing a center. The algorithm we develop, however, can be applied to more general notions of distance. In applications, distance sometimes means straight-line distance, but can also mean the travel time from point s to point z , or the driving distance (i.e., distance along roads), or even the cost of traveling. We will allow any distance function that satisfies the following natural properties.

- $\text{dist}(s, s) = 0$ for all $s \in S$
- the distance is symmetric: $\text{dist}(s, z) = \text{dist}(z, s)$ for all sites $s, z \in S$
- the triangle inequality: $\text{dist}(s, z) + \text{dist}(z, h) \geq \text{dist}(s, h)$

The first and third of these properties tend to be satisfied by essentially all natural notions of distance. Although there are applications with asymmetric distances, most cases of interest also satisfy the second property. Our greedy algorithm will apply to any distance function that satisfies these three properties, and it will depend on all three.

Next we have to clarify what we mean by the goal of wanting the centers to be “central.” Let C be a set of centers. We assume that the people in a given town will shop at the closest mall. This suggests we define the distance of a site s to the centers as $\text{dist}(s, C) = \min_{c \in C} \text{dist}(s, c)$. We say that C forms an r -cover if each site is within distance at most r from one of the centers—that is, if $\text{dist}(s, C) \leq r$ for all sites $s \in S$. The minimum r for which C is an r -cover will be called the *covering radius* of C and will be denoted by $r(C)$. In other words, the covering radius of a set of centers C is the farthest that anyone needs to travel to get to his or her nearest center. Our goal will be to select a set C of k centers for which $r(C)$ is as small as possible.

Designing and Analyzing the Algorithm

Difficulties with a Simple Greedy Algorithm We now discuss greedy algorithms for this problem. As before, the meaning of “greedy” here is necessarily a little fuzzy; essentially, we consider algorithms that select sites one by one in a myopic fashion—that is, choosing each without explicitly considering where the remaining sites will go.

Probably the simplest greedy algorithm would work as follows. It would put the first center at the best possible location for a single center, then keep adding centers so as to reduce the covering radius, each time, by as much as possible. It turns out that this approach is a bit too simplistic to be effective: there are cases where it can lead to very bad solutions.

To see that this simple greedy approach can be really bad, consider an example with only two sites s and z , and $k = 2$. Assume that s and z are located in the plane, with distance equal to the standard Euclidean distance in the plane, and that any point in the plane is an option for placing a center. Let d be the distance between s and z . Then the best location for a single center c_1 is halfway between s and z , and the covering radius of this one center is $r(\{c_1\}) = d/2$. The greedy algorithm would start with c_1 as the first center. No matter where we add a second center, at least one of s or z will have the center c_1 as closest, and so the covering radius of the set of two centers will still be $d/2$. Note that the optimum solution with $k = 2$ is to select s and z themselves as the centers. This will lead to a covering radius of 0. A more complex example illustrating the same problem can be obtained by having two dense “clusters” of sites, one around s and one around z . Here our proposed greedy algorithm would start by opening a center halfway between the clusters, while the optimum solution would open a separate center for each cluster.

Knowing the Optimal Radius Helps In searching for an improved algorithm, we begin with a useful thought experiment. Suppose for a minute that someone told us what the optimum radius r is. Would this information help? That is, suppose we *know* that there is a set of k centers C^* with radius $r(C^*) \leq r$, and

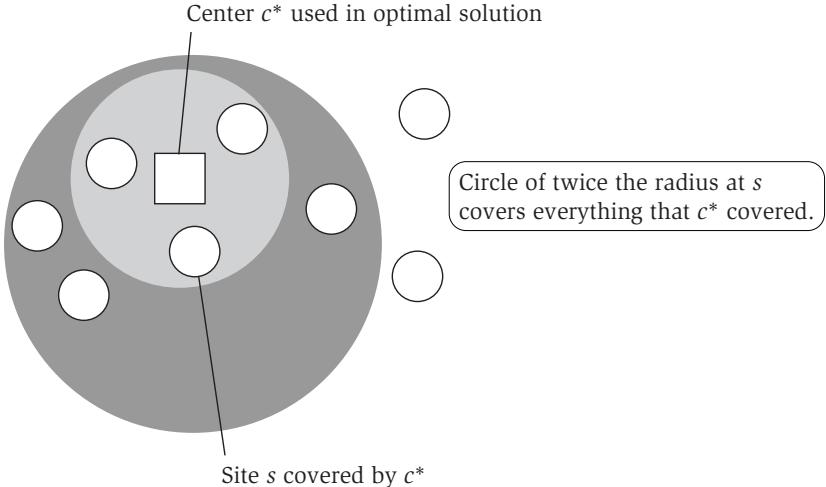


Figure 11.4 Everything covered at radius r by c^* is also covered at radius $2r$ by s .

our job is to find some set of k centers C whose covering radius is not much more than r . It turns out that finding a set of k centers with covering radius at most $2r$ can be done relatively easily.

Here is the idea: We can use the existence of this solution C^* in our algorithm even though we do not know what C^* is. Consider any site $s \in S$. There must be a center $c^* \in C^*$ that covers site s , and this center c^* is at distance at most r from s . Now our idea would be to take this site s as a center in our solution instead of c^* , as we have no idea what c^* is. We would like to make s cover all the sites that c^* covers in the unknown solution C^* . This is accomplished by expanding the radius from r to $2r$. All the sites that were at distance at most r from center c^* are at distance at most $2r$ from s (by the triangle inequality). See Figure 11.4 for a simple illustration of this argument.

```

 $S'$  will represent the sites that still need to be covered
Initialize  $S' = S$ 
Let  $C = \emptyset$ 
While  $S' \neq \emptyset$ 
  Select any site  $s \in S'$  and add  $s$  to  $C$ 
  Delete all sites from  $S'$  that are at distance at most  $2r$  from  $s$ 
EndWhile
If  $|C| \leq k$  then
  Return  $C$  as the selected set of sites
Else

```

Claim (correctly) that there is no set of k centers with
covering radius at most r

EndIf

Clearly, if this algorithm returns a set of at most k centers, then we have what we wanted.

(11.6) *Any set of centers C returned by the algorithm has covering radius $r(C) \leq 2r$.*

Next we argue that if the algorithm fails to return a set of centers, then its conclusion that no set can have covering radius at most r is indeed correct.

(11.7) *Suppose the algorithm selects more than k centers. Then, for any set C^* of size at most k , the covering radius is $r(C^*) > r$.*

Proof. Assume the opposite, that there is a set C^* of at most k centers with covering radius $r(C^*) \leq r$. Each center $c \in C$ selected by the greedy algorithm is one of the original sites in S , and the set C^* has covering radius at most r , so there must be a center $c^* \in C^*$ that is at most a distance of r from c —that is, $\text{dist}(c, c^*) \leq r$. Let us say that such a center c^* is *close* to c . We want to claim that no center c^* in the optimal solution C^* can be close to two different centers in the greedy solution C . If we can do this, we are done: each center $c \in C$ has a close optimal center $c^* \in C^*$, and each of these close optimal centers is distinct. This will imply that $|C^*| \geq |C|$, and since $|C| > k$, this will contradict our assumption that C^* contains at most k centers.

So we just need to show that no optimal center $c^* \in C$ can be close to each of two centers $c, c' \in C$. The reason for this is pictured in Figure 11.5. Each pair of centers $c, c' \in C$ is separated by a distance of more than $2r$, so if c^* were within a distance of at most r from each, then this would violate the triangle inequality, since $\text{dist}(c, c^*) + \text{dist}(c^*, c') \geq \text{dist}(c, c') > 2r$. ■

Eliminating the Assumption That We Know the Optimal Radius Now we return to the original question: How do we select a good set of k centers *without* knowing what the optimal covering radius might be?

It is worth discussing two different answers to this question. First, there are many cases in the design of approximation algorithms where it is conceptually useful to assume that you know the value achieved by an optimal solution. In such situations, you can often start with an algorithm designed under this assumption and convert it into one that achieves a comparable performance guarantee by simply trying out a range of “guesses” as to what the optimal

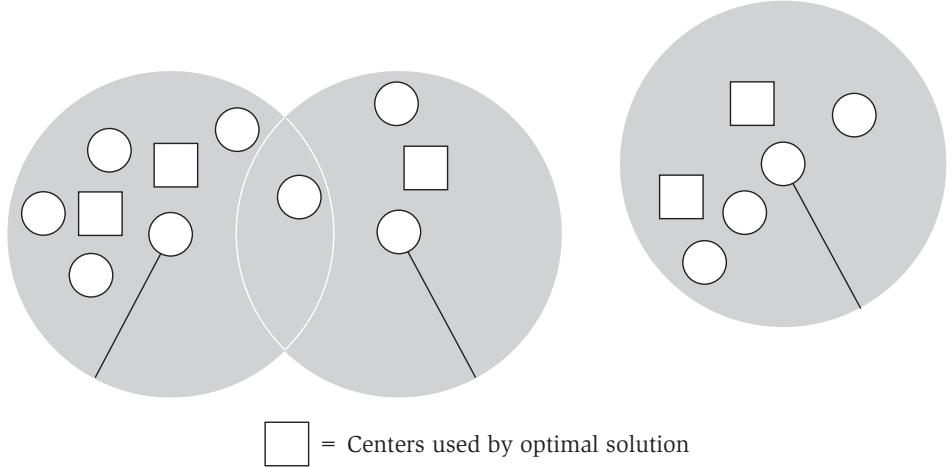


Figure 11.5 The crucial step in the analysis of the greedy algorithm that knows the optimal radius r . No center used by the optimal solution can lie in two different circles, so there must be at least as many optimal centers as there are centers chosen by the greedy algorithm.

value might be. Over the course of the algorithm, this sequence of guesses gets more and more accurate, until an approximate solution is reached.

For the Center Selection Problem, this could work as follows. We can start with some very weak initial guesses about the radius of the optimal solution: We know it is greater than 0, and it is at most the maximum distance r_{max} between any two sites. So we could begin by splitting the difference between these two and running the greedy algorithm we developed above with this value of $r = r_{max}/2$. One of two things will happen, according to the design of the algorithm: Either we find a set of k centers with covering radius at most $2r$, or we conclude that there is no solution with covering radius at most r . In the first case, we can afford to lower our guess on the radius of the optimal solution; in the second case, we need to raise it. This gives us the ability to perform a kind of binary search on the radius: in general, we will iteratively maintain values $r_0 < r_1$ so that we know the optimal radius is greater than r_0 , but we have a solution of radius at most $2r_1$. From these values, we can run the above algorithm with radius $r = (r_0 + r_1)/2$; we will either conclude that the optimal solution has radius greater than $r > r_0$, or obtain a solution with radius at most $2r = (r_0 + r_1) < 2r_1$. Either way, we will have sharpened our estimates on one side or the other, just as binary search is supposed to do. We can stop when we have estimates r_0 and r_1 that are close to each other; at this point, our solution of radius $2r_1$ is close to being a 2-approximation to the optimal radius, since we know the optimal radius is greater than r_0 (and hence close to r_1).

A Greedy Algorithm That Works For the specific case of the Center Selection Problem, there is a surprising way to get around the assumption of knowing the radius, without resorting to the general technique described earlier. It turns out we can run essentially the same greedy algorithm developed earlier without knowing anything about the value of r .

The earlier greedy algorithm, armed with knowledge of r , repeatedly selects one of the original sites s as the next center, making sure that it is at least $2r$ away from all previously selected sites. To achieve essentially the same effect without knowing r , we can simply select the site s that is farthest away from all previously selected centers: If there is any site at least $2r$ away from all previously chosen centers, then this farthest site s must be one of them. Here is the resulting algorithm.

```

Assume  $k \leq |S|$  (else define  $C = S$ )
Select any site  $s$  and let  $C = \{s\}$ 
While  $|C| < k$ 
    Select a site  $s \in S$  that maximizes  $\text{dist}(s, C)$ 
    Add site  $s$  to  $C$ 
EndWhile
Return  $C$  as the selected set of sites

```

(11.8) This greedy algorithm returns a set C of k points such that $r(C) \leq 2r(C^*)$, where C^* is an optimal set of k points.

Proof. Let $r = r(C^*)$ denote the minimum possible radius of a set of k centers. For the proof, we assume that we obtain a set of k centers C with $r(C) > 2r$, and from this we derive a contradiction.

So let s be a site that is more than $2r$ away from every center in C . Consider some intermediate iteration in the execution of the algorithm, where we have thus far selected a set of centers C' . Suppose we are adding the center c' in this iteration. We claim that c' is at least $2r$ away from all sites in C' . This follows as site s is more than $2r$ away from all sites in the larger set C , and we select a site c that is the farthest site from all previously selected centers. More formally, we have the following chain of inequalities:

$$\text{dist}(c', C') \geq \text{dist}(s, C') \geq \text{dist}(s, C) > 2r.$$

It follows that our greedy algorithm is a correct implementation of the first k iterations of the `while` loop of the previous algorithm, which knew the optimal radius r : In each iteration, we are adding a center at distance more than $2r$ from all previously selected centers. But the previous algorithm would

have $S' \neq \emptyset$ after selecting k centers, as it would have $s \in S'$, and so it would go on and select more than k centers and eventually conclude that k centers cannot have covering radius at most r . This contradicts our choice of r , and the contradiction proves that $r(C) \leq 2r$. ■

Note the surprising fact that our final greedy 2-approximation algorithm is a very simple modification of the first greedy algorithm that did not work. Perhaps the most important change is simply that our algorithm always selects sites as centers (i.e., every mall will be built in one of the little towns and not halfway between two of them).

11.3 Set Cover: A General Greedy Heuristic

In this section we will consider a very general problem that we also encountered in Chapter 8, the Set Cover Problem. A number of important algorithmic problems can be formulated as special cases of Set Cover, and hence an approximation algorithm for this problem will be widely applicable. We will see that it is possible to design a greedy algorithm here that produces solutions with a guaranteed approximation factor relative to the optimum, although this factor will be weaker than what we saw for the problems in Sections 11.1 and 11.2.

While the greedy algorithm we design for Set Cover will be very simple, the analysis will be more complex than what we encountered in the previous two sections. There we were able to get by with very simple bounds on the (unknown) optimum solution, while here the task of comparing to the optimum is more difficult, and we will need to use more sophisticated bounds. This aspect of the method can be viewed as our first example of the pricing method, which we will explore more fully in the next two sections.



The Problem

Recall from our discussion of NP-completeness that the Set Cover Problem is based on a set U of n elements and a list S_1, \dots, S_m of subsets of U ; we say that a *set cover* is a collection of these sets whose union is equal to all of U .

In the version of the problem we consider here, each set S_i has an associated *weight* $w_i \geq 0$. The goal is to find a set cover \mathcal{C} so that the total weight

$$\sum_{S_i \in \mathcal{C}} w_i$$

is minimized. Note that this problem is at least as hard as the decision version of Set Cover we encountered earlier; if we set all $w_i = 1$, then the minimum

weight of a set cover is at most k if and only if there is a collection of at most k sets that covers U .

Designing the Algorithm

We will develop and analyze a greedy algorithm for this problem. The algorithm will have the property that it builds the cover one set at a time; to choose its next set, it looks for one that seems to make the most progress toward the goal. What is a natural way to define “progress” in this setting? Desirable sets have two properties: They have small weight w_i , and they cover lots of elements. Neither of these properties alone, however, would be enough for designing a good approximation algorithm. Instead, it is natural to combine these two criteria into the single measure $w_i/|S_i|$ —that is, by selecting S_i , we cover $|S_i|$ elements at a cost of w_i , and so this ratio gives the “cost per element covered,” a very reasonable thing to use as a guide.

Of course, once some sets have already been selected, we are only concerned with how we are doing on the elements still left uncovered. So we will maintain the set R of remaining uncovered elements and choose the set S_i that minimizes $w_i/|S_i \cap R|$.

```

Greedy-Set-Cover:
Start with  $R = U$  and no sets selected
While  $R \neq \emptyset$ 
  Select set  $S_i$  that minimizes  $w_i/|S_i \cap R|$ 
  Delete set  $S_i$  from  $R$ 
EndWhile
Return the selected sets

```

As an example of the behavior of this algorithm, consider what it would do on the instance in Figure 11.6. It would first choose the set containing the four nodes at the bottom (since this has the best weight-to-coverage ratio, $1/4$). It then chooses the set containing the two nodes in the second row, and finally it chooses the sets containing the two individual nodes at the top. It thereby chooses a collection of sets of total weight 4. Because it myopically chooses the best option each time, this algorithm misses the fact that there’s a way to cover everything using a weight of just $2 + 2\epsilon$, by selecting the two sets that each cover a full column.

Analyzing the Algorithm

The sets selected by the algorithm clearly form a set cover. The question we want to address is: How much larger is the weight of this set cover than the weight w^* of an optimal set cover?

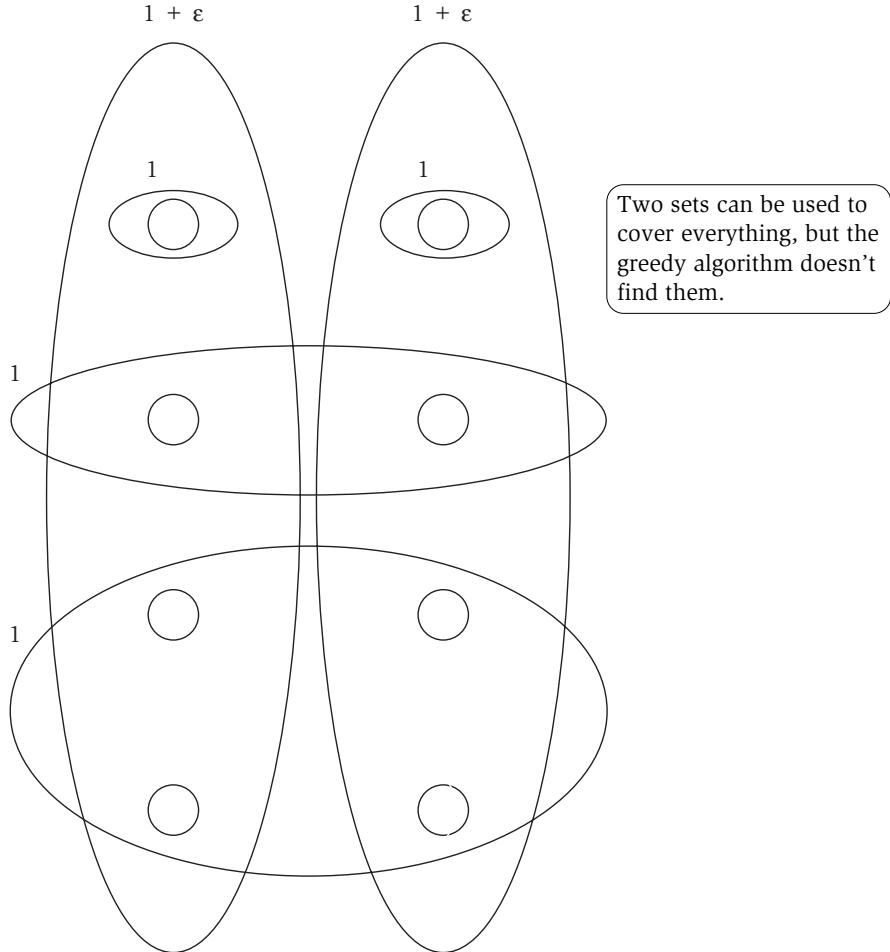


Figure 11.6 An instance of the Set Cover Problem where the weights of sets are either 1 or $1 + \varepsilon$ for some small $\varepsilon > 0$. The greedy algorithm chooses sets of total weight 4 , rather than the optimal solution of weight $2 + 2\varepsilon$.

As in Sections 11.1 and 11.2, our analysis will require a good lower bound on this optimum. In the case of the Load Balancing Problem, we used lower bounds that emerged naturally from the statement of the problem: the average load, and the maximum job size. The Set Cover Problem will turn out to be more subtle; “simple” lower bounds are not very useful, and instead we will use a lower bound that the greedy algorithm implicitly constructs as a by-product.

Recall the intuitive meaning of the ratio $w_i/|S_i \cap R|$ used by the algorithm; it is the “cost paid” for covering each new element. Let’s record this cost paid for

element s in the quantity c_s . We add the following line to the code immediately after selecting the set S_i .

```
Define  $c_s = w_i / |S_i \cap R|$  for all  $s \in S_i \cap R$ 
```

The values c_s do not affect the behavior of the algorithm at all; we view them as a bookkeeping device to help in our comparison with the optimum w^* . As each set S_i is selected, its weight is distributed over the costs c_s of the elements that are newly covered. Thus these costs completely account for the total weight of the set cover, and so we have

(11.9) *If \mathcal{C} is the set cover obtained by Greedy-Set-Cover, then $\sum_{S_i \in \mathcal{C}} w_i = \sum_{s \in U} c_s$.*

The key to the analysis is to ask how much total cost any single set S_k can account for—in other words, to give a bound on $\sum_{s \in S_k} c_s$ relative to the weight w_k of the set, even for sets not selected by the greedy algorithm. Giving an upper bound on the ratio

$$\frac{\sum_{s \in S_k} c_s}{w_k}$$

that holds for every set says, in effect, “To cover a lot of cost, you must use a lot of weight.” We know that the optimum solution must cover the full cost $\sum_{s \in U} c_s$ via the sets it selects; so this type of bound will establish that it needs to use at least a certain amount of weight. This is a lower bound on the optimum, just as we need for the analysis.

Our analysis will use the *harmonic function*

$$H(n) = \sum_{i=1}^n \frac{1}{i}.$$

To understand its asymptotic size as a function of n , we can interpret it as a sum approximating the area under the curve $y = 1/x$. Figure 11.7 shows how it is naturally bounded above by $1 + \int_1^n \frac{1}{x} dx = 1 + \ln n$, and bounded below by $\int_1^{n+1} \frac{1}{x} dx = \ln(n+1)$. Thus we see that $H(n) = \Theta(\ln n)$.

Here is the key to establishing a bound on the performance of the algorithm.

(11.10) *For every set S_k , the sum $\sum_{s \in S_k} c_s$ is at most $H(|S_k|) \cdot w_k$.*

Proof. To simplify the notation, we will assume that the elements of S_k are the first $d = |S_k|$ elements of the set U ; that is, $S_k = \{s_1, \dots, s_d\}$. Furthermore, let us assume that these elements are labeled in the order in which they are assigned a cost c_{s_j} by the greedy algorithm (with ties broken arbitrarily). There

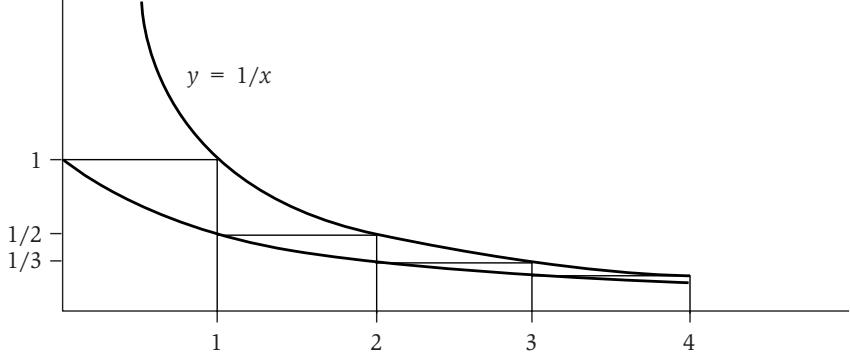


Figure 11.7 Upper and lower bounds for the Harmonic Function $H(n)$.

is no loss of generality in doing this, since it simply involves a renaming of the elements in U .

Now consider the iteration in which element s_j is covered by the greedy algorithm, for some $j \leq d$. At the start of this iteration, $s_j, s_{j+1}, \dots, s_d \in R$ by our labeling of the elements. This implies that $|S_k \cap R|$ is at least $d - j + 1$, and so the average cost of the set S_k is at most

$$\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$

Note that this is not necessarily an equality, since s_j may be covered in the same iteration as some of the other elements $s_{j'}$ for $j' < j$. In this iteration, the greedy algorithm selected a set S_i of minimum average cost; so this set S_i has average cost at most that of S_k . It is the average cost of S_i that gets assigned to s_j , and so we have

$$c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$

We now simply add up these inequalities for all elements $s \in S_k$:

$$\sum_{s \in S_k} c_s = \sum_{j=1}^d c_{s_j} \leq \sum_{j=1}^d \frac{w_k}{d - j + 1} = \frac{w_k}{d} + \frac{w_k}{d-1} + \dots + \frac{w_k}{1} = H(d) \cdot w_k. \blacksquare$$

We now complete our plan to use the bound in (11.10) for comparing the greedy algorithm's set cover to the optimal one. Letting $d^* = \max_i |S_i|$ denote the maximum size of any set, we have the following approximation result.

(11.11) *The set cover \mathcal{C} selected by Greedy-Set-Cover has weight at most $H(d^*)$ times the optimal weight w^* .*

Proof. Let \mathcal{C}^* denote the optimum set cover, so that $w^* = \sum_{S_i \in \mathcal{C}^*} w_i$. For each of the sets in \mathcal{C}^* , (11.10) implies

$$w_i \geq \frac{1}{H(d^*)} \sum_{s \in S_i} c_s.$$

Because these sets form a set cover, we have

$$\sum_{S_i \in \mathcal{C}^*} \sum_{s \in S_i} c_s \geq \sum_{s \in U} c_s.$$

Combining these with (11.9), we obtain the desired bound:

$$w^* = \sum_{S_i \in \mathcal{C}^*} w_i \geq \sum_{S_i \in \mathcal{C}^*} \frac{1}{H(d^*)} \sum_{s \in S_i} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in \mathcal{C}} w_i. \blacksquare$$

Asymptotically, then, the bound in (11.11) says that the greedy algorithm finds a solution within a factor $O(\log d^*)$ of optimal. Since the maximum set size d^* can be a constant fraction of the total number of elements n , this is a worst-case upper bound of $O(\log n)$. However, expressing the bound in terms of d^* shows us that we're doing much better if the largest set is small.

It's interesting to note that this bound is essentially the best one possible, since there are instances where the greedy algorithm can do this badly. To see how such instances arise, consider again the example in Figure 11.6. Now suppose we generalize this so that the underlying set of elements U consists of two tall columns with $n/2$ elements each. There are still two sets, each of weight $1 + \varepsilon$, for some small $\varepsilon > 0$, that cover the columns separately. We also create $\Theta(\log n)$ sets that generalize the structure of the other sets in the figure: there is a set that covers the bottommost $n/2$ nodes, another that covers the next $n/4$, another that covers the next $n/8$, and so forth. Each of these sets will have weight 1.

Now the greedy algorithm will choose the sets of size $n/2, n/4, n/8, \dots$, in the process producing a solution of weight $\Omega(\log n)$. Choosing the two sets that cover the columns separately, on the other hand, yields the optimal solution, with weight $2 + 2\varepsilon$. Through more complicated constructions, one can strengthen this to produce instances where the greedy algorithm incurs a weight that is very close to $H(n)$ times the optimal weight. And in fact, by much more complicated means, it has been shown that no polynomial-time approximation algorithm can achieve an approximation bound much better than $H(n)$ times optimal, unless $P = NP$.

11.4 The Pricing Method: Vertex Cover

We now turn to our second general technique for designing approximation algorithms, the *pricing method*. We will introduce this technique by considering a version of the Vertex Cover Problem. As we saw in Chapter 8, Vertex Cover is in fact a special case of Set Cover, and so we will begin this section by considering the extent to which one can use reductions in the design of approximation algorithms. Following this, we will develop an algorithm with a better approximation guarantee than the general bound that we obtained for Set Cover in the previous section.

The Problem

Recall that a *vertex cover* in a graph $G = (V, E)$ is a set $S \subseteq V$ so that each edge has at least one end in S . In the version of the problem we consider here, each vertex $i \in V$ has a *weight* $w_i \geq 0$, with the weight of a set S of vertices denoted $w(S) = \sum_{i \in S} w_i$. We would like to find a vertex cover S for which $w(S)$ is minimum. When all weights are equal to 1, deciding if there is a vertex cover of weight at most k is the standard decision version of Vertex Cover.

Approximations via Reductions? Before we work on developing an algorithm, we pause to discuss an interesting issue that arises: Vertex Cover is easily reducible to Set Cover, and we have just seen an approximation algorithm for Set Cover. What does this imply about the approximability of Vertex Cover? A discussion of this question brings out some of the subtle ways in which approximation results interact with polynomial-time reductions.

First consider the special case in which all weights are equal to 1—that is, we are looking for a vertex cover of minimum size. We will call this the *unweighted case*. Recall that we showed Set Cover to be NP-complete using a reduction from the decision version of unweighted Vertex Cover. That is,

$$\text{Vertex Cover} \leq_P \text{Set Cover}$$

This reduction says, “If we had a polynomial-time algorithm that solves the Set Cover Problem, then we could use this algorithm to solve the Vertex Cover Problem in polynomial time.” We now have a polynomial-time algorithm for the Set Cover Problem that approximates the solution. Does this imply that we can use it to formulate an approximation algorithm for Vertex Cover?

(11.12) *One can use the Set Cover approximation algorithm to give an $H(d)$ -approximation algorithm for the weighted Vertex Cover Problem, where d is the maximum degree of the graph.*

Proof. The proof is based on the reduction that showed $\text{Vertex Cover} \leq_P \text{Set Cover}$, which also extends to the weighted case. Consider an instance of the weighted Vertex Cover Problem, specified by a graph $G = (V, E)$. We define an

instance of Set Cover as follows. The underlying set U is equal to E . For each node i , we define a set S_i consisting of all edges incident to node i and give this set weight w_i . Collections of sets that cover U now correspond precisely to vertex covers. Note that the maximum size of any S_i is precisely the maximum degree d .

Hence we can use the approximation algorithm for Set Cover to find a vertex cover whose weight is within a factor of $H(d)$ of minimum. ■

This $H(d)$ -approximation is quite good when d is small; but it gets worse as d gets larger, approaching a bound that is logarithmic in the number of vertices. In the following, we will develop a stronger approximation algorithm that comes within a factor of 2 of optimal.

Before turning to the 2-approximation algorithm, we make the following further observation: One has to be very careful when trying to use reductions for designing approximation algorithms. It worked in (11.12), but we made sure to go through an argument for why it worked; it is not the case that every polynomial-time reduction leads to a comparable implication for approximation algorithms.

Here is a cautionary example. We used Independent Set to prove that the Vertex Cover Problem is NP-complete. Specifically, we proved

$$\text{Independent Set} \leq_P \text{Vertex Cover},$$

which states that “if we had a polynomial-time algorithm that solves the Vertex Cover Problem, then we could use this algorithm to solve the Independent Set Problem in polynomial time.” Can we use an approximation algorithm for the minimum-size vertex cover to design a comparably good approximation algorithm for the maximum-size independent set?

The answer is no. Recall that a set I of vertices is independent if and only if its complement $S = V - I$ is a vertex cover. Given a minimum-size vertex cover S^* , we obtain a maximum-size independent set by taking the complement $I^* = V - S$. Now suppose we use an approximation algorithm for the Vertex Cover Problem to get an approximately minimum vertex cover S . The complement $I = V - S$ is indeed an independent set—there’s no problem there. The trouble is when we try to determine our approximation factor for the Independent Set Problem; I can be very far from optimal. Suppose, for example, that the optimal vertex cover S^* and the optimal independent set I^* both have size $|V|/2$. If we invoke a 2-approximation algorithm for the Vertex Cover Problem, we may perfectly well get back the set $S = V$. But, in this case, our “approximately maximum independent set” $I = V - S$ has no elements.



Designing the Algorithm: The Pricing Method

Even though (11.12) gave us an approximation algorithm with a provable guarantee, we will be able to do better. Our approach forms a nice illustration of the *pricing method* for designing approximation algorithms.

The Pricing Method to Minimize Cost The pricing method (also known as the *primal-dual method*) is motivated by an economic perspective. For the case of the Vertex Cover Problem, we will think of the weights on the nodes as *costs*, and we will think of each edge as having to pay for its “share” of the cost of the vertex cover we find. We have actually just seen an analysis of this sort, in the greedy algorithm for Set Cover from Section 11.3; it too can be thought of as a pricing algorithm. The greedy algorithm for Set Cover defined values c_s , the cost the algorithm paid for covering element s . We can think of c_s as the element s ’s “share” of the cost. Statement (11.9) shows that it is very natural to think of the values c_s as cost-shares, as the sum of the cost-shares $\sum_{s \in U} c_s$ is the cost of the set cover \mathcal{C} returned by the algorithm, $\sum_{S_i \in \mathcal{C}} w_i$. The key to proving that the algorithm is an $H(d^*)$ -approximation algorithm was a certain approximate “fairness” property for the cost-shares: (11.10) shows that the elements in a set S_k are charged by at most an $H(|S_k|)$ factor more than the cost of covering them by the set S_k .

In this section, we’ll develop the pricing technique through another application, Vertex Cover. Again, we will think of the weight w_i of the vertex i as the cost for using i in the cover. We will think of each edge e as a separate “agent” who is willing to “pay” something to the node that covers it. The algorithm will not only find a vertex cover S , but also determine prices $p_e \geq 0$ for each edge $e \in E$, so that if each edge $e \in E$ pays the price p_e , this will in total approximately cover the cost of S . These prices p_e are the analogues of c_s from the Set Cover Algorithm.

Thinking of the edges as agents suggests some natural fairness rules for prices, analogous to the property proved by (11.10). First of all, selecting a vertex i covers all edges incident to i , so it would be “unfair” to charge these incident edges in total more than the cost of vertex i . We call prices p_e *fair* if, for each vertex i , the edges adjacent to i do not have to pay more than the cost of the vertex: $\sum_{e=(i,j)} p_e \leq w_i$. Note that the property proved by (11.10) for Set Cover is an approximate fairness condition, while in the Vertex Cover algorithm we’ll actually use the exact fairness defined here. A useful fact about fair prices is that they provide a lower bound on the cost of any solution.

(11.13) For any vertex cover S^* , and any nonnegative and fair prices p_e , we have $\sum_{e \in E} p_e \leq w(S^*)$.

Proof. Consider a vertex cover S^* . By the definition of fairness, we have $\sum_{e=(i,j)} p_e \leq w_i$ for all nodes $i \in S^*$. Adding these inequalities over all nodes in S^* , we get

$$\sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq \sum_{i \in S^*} w_i = w(S^*).$$

Now the expression on the left-hand side is a sum of terms, each of which is some edge price p_e . Since S^* is a vertex cover, each edge e contributes at least one term p_e to the left-hand side. It may contribute more than one copy of p_e to this sum, since it may be covered from both ends by S^* ; but the prices are nonnegative, and so the sum on the left-hand side is at least as large as the sum of all prices p_e . That is,

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} p_e.$$

Combining this with the previous inequality, we get

$$\sum_{e \in E} p_e \leq w(S^*),$$

as desired. ■

The Algorithm The goal of the approximation algorithm will be to find a vertex cover and to set prices at the same time. We can think of the algorithm as being greedy in how it sets the prices. It then uses these prices to drive the way it selects nodes for the vertex cover.

We say that a node i is *tight* (or “paid for”) if $\sum_{e=(i,j)} p_e = w_i$.

```

Vertex-Cover-Approx( $G, w$ ):
  Set  $p_e = 0$  for all  $e \in E$ 
  While there is an edge  $e = (i, j)$  such that neither  $i$  nor  $j$  is tight
    Select such an edge  $e$ 
    Increase  $p_e$  without violating fairness
  EndWhile
  Let  $S$  be the set of all tight nodes
  Return  $S$ 

```

For example, consider the execution of this algorithm on the instance in Figure 11.8. Initially, no node is tight; the algorithm decides to select the edge (a, b) . It can raise the price paid by (a, b) up to 3, at which point the node b becomes tight and it stops. The algorithm then selects the edge (a, d) . It can only raise this price up to 1, since at this point the node a becomes tight (due to the fact that the weight of a is 4, and it is already incident to an edge that is

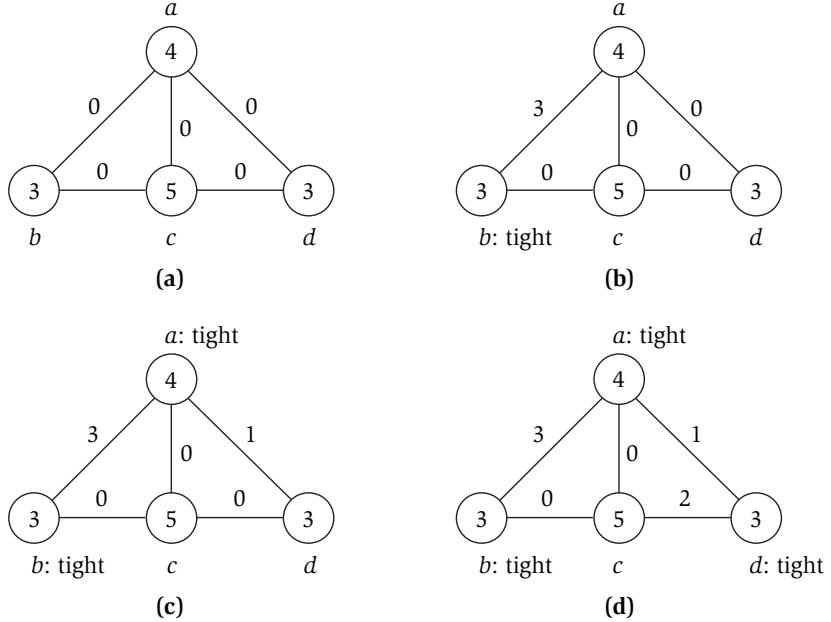


Figure 11.8 Parts (a)–(d) depict the steps in an execution of the pricing algorithm on an instance of the weighted Vertex Cover Problem. The numbers inside the nodes indicate their weights; the numbers annotating the edges indicate the prices they pay as the algorithm proceeds.

paying 3). Finally, the algorithm selects the edge (c, d) . It can raise the price paid by (c, d) up to 2, at which point d becomes tight. We now have a situation where all edges have at least one tight end, so the algorithm terminates. The tight nodes are a , b , and d ; so this is the resulting vertex cover. (Note that this is not the minimum-weight vertex cover; that would be obtained by selecting a and c .)



Analyzing the Algorithm

At first sight, one may have the sense that the vertex cover S is fully paid for by the prices: all nodes in S are tight, and hence the edges adjacent to the node i in S can pay for the cost of i . But the point is that an edge e can be adjacent to more than one node in the vertex cover (i.e., if both ends of e are in the vertex cover), and hence e may have to pay for more than one node. This is the case, for example, with the edges (a, b) and (a, d) at the end of the execution in Figure 11.8.

However, notice that if we take edges for which both ends happened to show up in the vertex cover, and we charge them their price twice, then we're exactly paying for the vertex cover. (In the example, the cost of the vertex

cover is the cost of nodes a , b , and d , which is 10. We can account for this cost exactly by charging (a, b) and (a, d) twice, and (c, d) once.) Now, it's true that this is unfair to some edges, but the amount of unfairness can be bounded: Each edge gets charged its price at most two times (once for each end).

We now make this argument precise, as follows.

(11.14) *The set S and prices p returned by the algorithm satisfy the inequality $w(S) \leq 2 \sum_{e \in E} p_e$.*

Proof. All nodes in S are tight, so we have $\sum_{e=(i,j)} p_e = w_i$ for all $i \in S$. Adding over all nodes in S we get

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e.$$

An edge $e = (i, j)$ can be included in the sum on the right-hand side at most twice (if both i and j are in S), and so we get

$$w(S) = \sum_{i \in S} \sum_{e=(i,j)} p_e \leq 2 \sum_{e \in E} p_e,$$

as claimed. ■

Finally, this factor of 2 carries into an argument that yields the approximation guarantee.

(11.15) *The set S returned by the algorithm is a vertex cover, and its cost is at most twice the minimum cost of any vertex cover.*

Proof. First note that S is indeed a vertex cover. Suppose, by contradiction, that S does not cover edge $e = (i, j)$. This implies that neither i nor j is tight, and this contradicts the fact that the `While` loop of the algorithm terminated.

To get the claimed approximation bound, we simply put together statement (11.14) with (11.13). Let p be the prices set by the algorithm, and let S^* be an optimal vertex cover. By (11.14) we have $2 \sum_{e \in E} p_e \geq w(S)$, and by (11.13) we have $\sum_{e \in E} p_e \leq w(S^*)$.

In other words, the sum of the edge prices is a lower bound on the weight of *any* vertex cover, and twice the sum of the edge prices is an upper bound on the weight of our vertex cover:

$$w(S) \leq 2 \sum_{e \in E} p_e \leq 2w(S^*). \quad \blacksquare$$

11.5 Maximization via the Pricing Method: The Disjoint Paths Problem

We now continue the theme of pricing algorithms with a fundamental problem that arises in network routing: the *Disjoint Paths Problem*. We'll start out by developing a greedy algorithm for this problem and then show an improved algorithm based on pricing.

The Problem

To set up the problem, it helps to recall one of the first applications we saw for the Maximum-Flow Problem: finding disjoint paths in graphs, which we discussed in Chapter 7. There we were looking for edge-disjoint paths all starting at a node s and ending at a node t . How crucial is it to the tractability of this problem that all paths have to start and end at the same node? Using the technique from Section 7.7, one can extend this to find disjoint paths where we are given a set of start nodes S and a set of terminals T , and the goal is to find edge-disjoint paths where paths may start at any node in S and end at any node in T .

Here, however, we will look at a case where each path to be routed has its own designated starting node and ending node. Specifically, we consider the following *Maximum Disjoint Paths Problem*. We are given a directed graph G , together with k pairs of nodes $(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)$ and an integer capacity c . We think of each pair (s_i, t_i) as a *routing request*, which asks for a path from s_i to t_i . A solution to this instance consists of a subset of the requests we will satisfy, $I \subseteq \{1, \dots, k\}$, together with paths that satisfy them while not overloading any one edge: a path P_i for $i \in I$ so that P_i goes from s_i to t_i , and each edge is used by at most c paths. The problem is to find a solution with $|I|$ as large as possible—that is, to satisfy as many requests as possible. Note that the capacity c controls how much “sharing” of edges we allow; when $c = 1$, we are requiring the paths to be fully edge-disjoint, while larger c allows some overlap among the paths.

We have seen in Exercise 39 in Chapter 8 that it is NP-complete to determine whether all k routing requests can be satisfied when the paths are required to be node-disjoint. It is not hard to show that the edge-disjoint version of the problem (corresponding to the case with $c = 1$) is also NP-complete.

Thus it turns out to have been crucial for the application of efficient network flow algorithms that the endpoints of the paths not be explicitly paired up as they are in Maximum Disjoint Paths. To develop this point a little further, suppose we attempted to reduce Maximum Disjoint Paths to a network flow problem by defining the set of sources to be $S = \{s_1, s_2, \dots, s_k\}$, defining the

set of sinks to be $T = \{t_1, t_2, \dots, t_k\}$, setting each edge capacity to be c , and looking for the maximum possible number of disjoint paths starting in S and ending in T . Why wouldn't this work? The problem is that there's no way to tell the flow algorithm that a path starting at $s_i \in S$ *must* end at $t_i \in T$; the algorithm guarantees only that this path will end at *some* node in T . As a result, the paths that come out of the flow algorithm may well not constitute a solution to the instance of Maximum Disjoint Paths, since they might not link a source s_i to its corresponding endpoint t_i .

Disjoint paths problems, where we need to find paths connecting designated pairs of terminal nodes, are very common in networking applications. Just think about paths on the Internet that carry streaming media or Web data, or paths through the phone network carrying voice traffic.¹ Paths sharing edges can interfere with each other, and too many paths sharing a single edge will cause problems in most applications. The maximum allowable amount of sharing will differ from application to application. Requiring the paths to be disjoint is the strongest constraint, eliminating all interference between paths. We'll see, however, that in cases where some sharing is allowed (even just two paths to an edge), better approximation algorithms are possible.

Designing and Analyzing a Greedy Algorithm

We first consider a very simple algorithm for the case when the capacity $c = 1$: that is, when the paths need to be edge-disjoint. The algorithm is essentially greedy, except that it exhibits a preference for short paths. We will show that this simple algorithm is an $O(\sqrt{m})$ -approximation algorithm, where $m = |E|$ is the number of edges in G . This may sound like a rather large factor of approximation, and it is, but there is a strong sense in which it is essentially the best we can do. The Maximum Disjoint Paths Problem is not only NP-complete, but it is also hard to approximate: It has been shown that unless $\mathcal{P} = \mathcal{NP}$, it is impossible for any polynomial-time algorithm to achieve an approximation bound significantly better than $O(\sqrt{m})$ in arbitrary directed graphs.

After developing the greedy algorithm, we will consider a slightly more sophisticated pricing algorithm for the capacitated version. It is interesting

¹ A researcher from the telecommunications industry once gave the following explanation for the distinction between Maximum Disjoint Paths and network flow, and the broken reduction in the previous paragraph. On Mother's Day, traditionally the busiest day of the year for telephone calls, the phone company must solve an enormous disjoint paths problem: ensuring that each source individual s_i is connected by a path through the voice network to his or her mother t_i . Network flow algorithms, finding disjoint paths between a set S and a set T , on the other hand, will ensure only that each person gets their call through to *somebody's* mother.

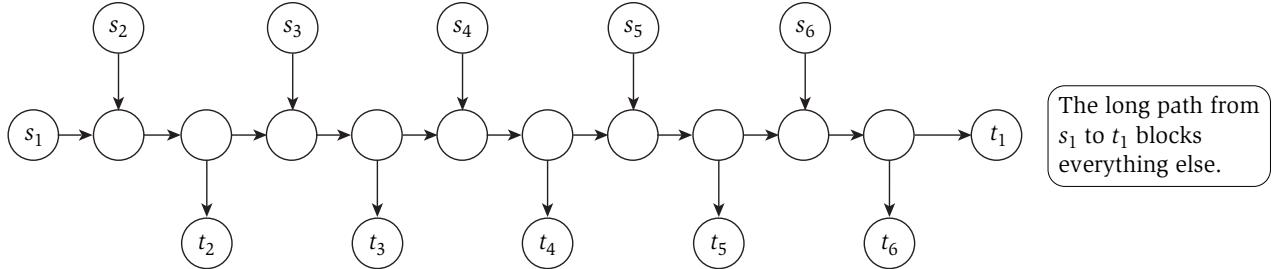


Figure 11.9 A case in which it's crucial that a greedy algorithm for selecting disjoint paths favors short paths over long ones.

to note that the pricing algorithm does much better than the simple greedy algorithm, even when the capacity c is only slightly more than 1.

```

Greedy-Disjoint-Paths:
Set  $I = \emptyset$ 
Until no new path can be found
  Let  $P_i$  be the shortest path (if one exists) that is edge-disjoint
    from previously selected paths, and connects some  $(s_i, t_i)$  pair
    that is not yet connected
  Add  $i$  to  $I$  and select path  $P_i$  to connect  $s_i$  to  $t_i$ 
EndUntil

```

Analyzing the Algorithm The algorithm clearly selects edge-disjoint paths. Assuming the graph G is connected, it must select at least one path. But how does the number of paths selected compare with the maximum possible? A kind of situation we need to worry about is shown in Figure 11.9: One of the paths, from s_1 to t_1 , is very long, so if we select it first, we eliminate up to $\Omega(m)$ other paths.

We now show that the greedy algorithm's preference for short paths not only avoids the problem in this example, but in general it limits the number of other paths that a selected path can interfere with.

(11.16) *The algorithm Greedy-Disjoint-Paths is a $(2\sqrt{m} + 1)$ -approximation algorithm for the Maximum Disjoint Paths Problem.*

Proof. Consider an optimal solution: Let I^* be the set of pairs for which a path was selected in this optimum solution, and let P_i^* for $i \in I^*$ be the selected paths. Let I denote the set of pairs returned by the algorithm, and let P_i for $i \in I$ be the corresponding paths. We need to bound $|I^*|$ in terms of $|I|$. The key to the analysis is to make a distinction between short and long paths and to consider

them separately. We will call a path *long* if it has at least \sqrt{m} edges, and we will call it *short* otherwise. Let I_s^* denote the set of indices in I^* so that the corresponding path P_i^* is short, and let I_s denote the set of indices in I so that the corresponding path P_i is short.

The graph G has m edges, and each long path uses at least \sqrt{m} edges, so there can be at most \sqrt{m} long paths in I^* .

Now consider the short paths in I^* . In order for I^* to be much larger than I , there would have to be many pairs that are connected in I^* but not in I . Thus let us consider pairs that are connected by the optimum using a short path, but are not connected by the greedy algorithm. Since the path P_i^* connecting s_i and t_i in the optimal solution I^* is short, the greedy algorithm would have selected this path, if it had been available, before selecting any long paths. But the greedy algorithm did not connect s_i and t_i at all, and hence one of the edges e along the path P_i^* must occur in a path P_j that was selected earlier by the greedy algorithm. We will say that edge e *blocks* the path P_i^* .

Now the lengths of the paths selected by the greedy algorithm are monotone increasing, since each iteration has fewer options for choosing paths. The path P_j was selected before considering P_i^* and hence it must be shorter: $|P_j| \leq |P_i^*| \leq \sqrt{m}$. So path P_j is short. Since the paths used by the optimum are edge-disjoint, each edge in a path P_j can block at most one path P_i^* . It follows that each short path P_j blocks at most \sqrt{m} paths in the optimal solution, and so we get the bound

$$|I_s^* - I| \leq \sum_{j \in I_s} |P_j| \leq |I_s| \sqrt{m}.$$

We use this to produce a bound on the overall size of the optimal solution. To do this, we view I^* as consisting of three kinds of paths, following the analysis thus far:

- long paths, of which there are at most \sqrt{m} ;
- paths that are also in I ; and
- short paths that are not in I , which we have just bounded by $|I_s| \sqrt{m}$.

Putting this all together, and using the fact that $|I| \geq 1$ whenever at least one set of terminal pairs can be connected, we get the claimed bound:

$$|I^*| \leq \sqrt{m} + |I| + |I_s^* - I| \leq \sqrt{m} + |I| + \sqrt{m}|I_s| \leq (2\sqrt{m} + 1)|I|. \blacksquare$$

This provides an approximation algorithm for the case when the selected paths have to be disjoint. As we mentioned earlier, the approximation bound of $O(\sqrt{m})$ is rather weak, but unless $\mathcal{P} = \mathcal{NP}$, it is essentially the best possible for the case of disjoint paths in arbitrary directed graphs.



Designing and Analyzing a Pricing Algorithm

Not letting any two paths use the same edge is quite extreme; in most applications one can allow a few paths to share an edge. We will now develop an analogous algorithm, based on the pricing method, for the case where $c > 1$ paths may share any edge. In the disjoint case just considered, we viewed all edges as equal and preferred short paths. We can think of this as a simple kind of pricing algorithm: the paths have to pay for using up the edges, and each edge has a unit cost. Here we will consider a pricing scheme in which edges are viewed as more expensive if they have been used already, and hence have less capacity left over. This will encourage the algorithm to “spread out” its paths, rather than piling them up on any single edge. We will refer to the cost of an edge e as its *length* ℓ_e , and define the *length* of a path to be the sum of the lengths of the edges it contains: $\ell(P) = \sum_{e \in P} \ell_e$. We will use a multiplicative parameter β to increase the length of an edge each time an additional path uses it.

```

Greedy-Paths-with-Capacity:
Set  $I = \emptyset$ 
Set edge length  $\ell_e = 1$  for all  $e \in E$ 
Until no new path can be found
    Let  $P_i$  be the shortest path (if one exists) so that adding  $P_i$  to
        the selected set of paths does not use any edge more than  $c$ 
        times, and  $P_i$  connects some  $(s_i, t_i)$  pair not yet connected
    Add  $i$  to  $I$  and select path  $P_i$  to connect  $s_i$  to  $t_i$ 
    Multiply the length of all edges along  $P_i$  by  $\beta$ 
EndUntil

```

Analyzing the Algorithm For the analysis we will focus on the simplest case, when at most two paths may use the same edge—that is, when $c = 2$. We’ll see that, for this case, setting $\beta = m^{1/3}$ will give the best approximation result for this algorithm. Unlike the disjoint paths case (when $c = 1$), it is not known whether the approximation bounds we obtain here for $c > 1$ are close to the best possible for polynomial-time algorithms in general, assuming $\mathcal{P} \neq \mathcal{NP}$.

The key to the analysis in the disjoint case was to distinguish “short” and “long” paths. For the case when $c = 2$, we will consider a path P_i selected by the algorithm to be *short* if the length is less than β^2 . Let I_s denote the set of short paths selected by the algorithm.

Next we want to compare the number of paths selected with the maximum possible. Let I^* be an optimal solution and P_i^* be the set of paths used in this solution. As before, the key to the analysis is to consider the edges that block

the selection of paths in I^* . Long paths can block a lot of other paths, so for now we will focus on the short paths in I_s . As we try to continue following what we did in the disjoint case, we immediately run into a difficulty, however. In that case, the length of a path in I^* was simply the number of edges it contained; but here, the lengths are changing as the algorithm runs, and so it is not clear how to define the length of a path in I^* for purposes of the analysis. In other words, for the analysis, when should we measure this length? (At the beginning of the execution? At the end?)

It turns out that the crucial moment in the algorithm, for purposes of our analysis, is the first point at which there are no short paths left to choose. Let $\bar{\ell}$ be the length function at this point in the execution of the algorithm; we'll use $\bar{\ell}$ to measure the length of paths in I^* . For a path P , we use $\bar{\ell}(P)$ to denote its length, $\sum_{e \in P} \bar{\ell}_e$. We consider a path P_i^* in the optimal solution I^* *short* if $\bar{\ell}(P_i^*) < \beta^2$, and *long* otherwise. Let I_s^* denote the set of short paths in I^* . The first step is to show that there are no short paths connecting pairs that are not connected by the approximation algorithm.

(11.17) *Consider a source-sink pair $i \in I^*$ that is not connected by the approximation algorithm; that is, $i \notin I$. Then $\bar{\ell}(P_i^*) \geq \beta^2$.*

Proof. As long as short paths are being selected, we do not have to worry about explicitly enforcing the requirement that each edge be used by at most $c = 2$ paths: any edge e considered for selection by a third path would already have length $\ell_e = \beta^2$, and hence be long.

Consider the state of the algorithm with length $\bar{\ell}$. By the argument in the previous paragraph, we can imagine the algorithm having run up to this point without caring about the limit of c ; it just selected a short path whenever it could find one. Since the endpoints s_i, t_i of P_i^* are not connected by the greedy algorithm, and since there are no short paths left when the length function reaches $\bar{\ell}$, it must be the case that path P_i^* has length at least β^2 as measured by $\bar{\ell}$. ■

The analysis in the disjoint case used the fact that there are only m edges to limit the number of long paths. Here we consider length $\bar{\ell}$, rather than the number of edges, as the quantity that is being consumed by paths. Hence, to be able to reason about this, we will need a bound on the total length in the graph $\sum_e \bar{\ell}_e$. The sum of the lengths over all edges $\sum_e \ell_e$ starts out at m (length 1 for each edge). Adding a short path to the solution I_s can increase the length by at most β^3 , as the selected path has length at most β^2 , and the lengths of the edges are increased by a β factor along the path. This gives us a useful comparison between the number of short paths selected and the total length.

(11.18) *The set I_s of short paths selected by the approximation algorithm, and the lengths $\bar{\ell}$, satisfy the relation $\sum_e \bar{\ell}_e \leq \beta^3 |I_s| + m$.*

Finally, we prove an approximation bound for this algorithm. We will find that even though we have simply increased the number of paths allowed on each edge from 1 to 2, the approximation guarantee drops by a significant amount that essentially incorporates this change into the exponent: from $O(m^{1/2})$ down to $O(m^{1/3})$.

(11.19) *The algorithm Greedy-Paths-with-Capacity, using $\beta = m^{1/3}$, is a $(4m^{1/3} + 1)$ -approximation algorithm in the case when the capacity $c = 2$.*

Proof. We first bound $|I^* - I|$. By (11.17), we have $\bar{\ell}(P_i^*) \geq \beta^2$ for all $i \in I^* - I$. Summing over all paths in $I^* - I$, we get

$$\sum_{i \in I^* - I} \bar{\ell}(P_i^*) \geq \beta^2 |I^* - I|.$$

On the other hand, each edge is used by at most two paths in the solution I^* , so we have

$$\sum_{i \in I^* - I} \bar{\ell}(P_i^*) \leq \sum_{e \in E} 2\bar{\ell}_e.$$

Combining these bounds with (11.18) we get

$$\begin{aligned} \beta^2 |I^*| &\leq \beta^2 |I^* - I| + \beta^2 |I| \leq \sum_{i \in I^* - I} \bar{\ell}(P_i^*) + \beta^2 |I| \\ &\leq \sum_{e \in E} 2\bar{\ell}_e + \beta^2 |I| \leq 2(\beta^3 |I| + m) + \beta^2 |I|. \end{aligned}$$

Finally, dividing through by β^2 , using $|I| \geq 1$ and setting $\beta = m^{1/3}$, we get that $|I^*| \leq (4m^{1/3} + 1)|I|$. ■

The same algorithm also works for the capacitated Disjoint Path Problem with any capacity $c > 0$. If we choose $\beta = m^{1/(c+1)}$, then the algorithm is a $(2cm^{1/(c+1)} + 1)$ -approximation algorithm. To extend the analysis, one has to consider paths to be short if their length is at most β^c .

(11.20) *The algorithm Greedy-Paths-with-Capacity, using $\beta = m^{1/c+1}$, is a $(2cm^{1/(c+1)} + 1)$ -approximation algorithm when the edge capacities are c .*

11.6 Linear Programming and Rounding: An Application to Vertex Cover

We will start by introducing a powerful technique from operations research: *linear programming*. Linear programming is the subject of entire courses, and

we will not attempt to provide any kind of comprehensive overview of it here. In this section, we will introduce some of the basic ideas underlying linear programming and show how these can be used to approximate NP-hard optimization problems.

Recall that in Section 11.4 we developed a 2-approximation algorithm for the weighted Vertex Cover Problem. As a first application for the linear programming technique, we'll give here a different 2-approximation algorithm that is conceptually much simpler (though slower in running time).

Linear Programming as a General Technique

Our 2-approximation algorithm for the weighted version of Vertex Cover will be based on linear programming. We describe linear programming here not just to give the approximation algorithm, but also to illustrate its power as a very general technique.

So what is linear programming? To answer this, it helps to first recall, from linear algebra, the problem of simultaneous linear equations. Using matrix-vector notation, we have a vector x of unknown real numbers, a given matrix A , and a given vector b ; and we want to solve the equation $Ax = b$. Gaussian elimination is a well-known efficient algorithm for this problem.

The basic Linear Programming Problem can be viewed as a more complex version of this, with inequalities in place of equations. Specifically, consider the problem of determining a vector x that satisfies $Ax \geq b$. By this notation, we mean that each coordinate of the vector Ax should be greater than or equal to the corresponding coordinate of the vector b . Such systems of inequalities define regions in space. For example, suppose $x = (x_1, x_2)$ is a two-dimensional vector, and we have the four inequalities

$$\begin{aligned} x_1 &\geq 0, x_2 \geq 0 \\ x_1 + 2x_2 &\geq 6 \\ 2x_1 + x_2 &\geq 6 \end{aligned}$$

Then the set of solutions is the region in the plane shown in Figure 11.10.

Given a region defined by $Ax \geq b$, linear programming seeks to minimize a linear combination of the coordinates of x , over all x belonging to the region. Such a linear combination can be written $c^t x$, where c is a vector of coefficients, and $c^t x$ denotes the inner product of two vectors. Thus our standard form for Linear Programming, as an optimization problem, will be the following.

Given an $m \times n$ matrix A , and vectors $b \in R^m$ and $c \in R^n$, find a vector $x \in R^n$ to solve the following optimization problem:

$$\min(c^t x \text{ such that } x \geq 0; Ax \geq b).$$

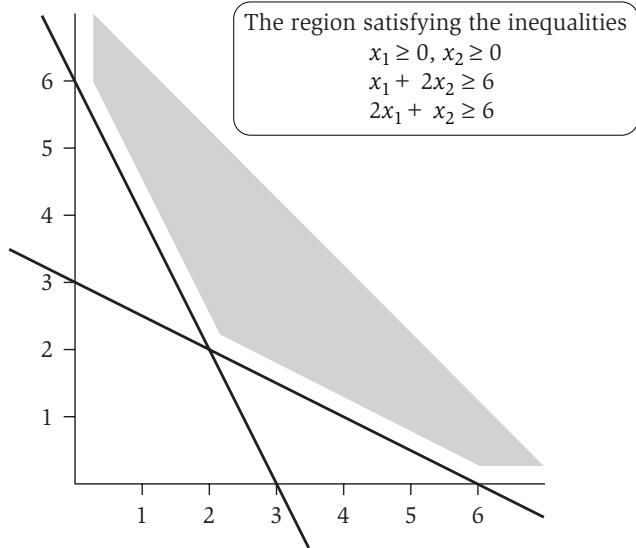


Figure 11.10 The feasible region of a simple linear program.

$c^t x$ is often called the *objective function* of the linear program, and $Ax \geq b$ is called the set of *constraints*. For example, suppose we define the vector c to be $(1.5, 1)$ in the example in Figure 11.10; in other words, we are seeking to minimize the quantity $1.5x_1 + x_2$ over the region defined by the inequalities. The solution to this would be to choose the point $x = (2, 2)$, where the two slanting lines cross; this yields a value of $c^t x = 5$, and one can check that there is no way to get a smaller value.

We can phrase Linear Programming as a decision problem in the following way.

Given a matrix A , vectors b and c , and a bound γ , does there exist x so that $x \geq 0$, $Ax \geq b$, and $c^t x \leq \gamma$?

To avoid issues related to how we represent real numbers, we will assume that the coordinates of the vectors and matrices involved are integers.

The Computational Complexity of Linear Programming The decision version of Linear Programming is in NP . This is intuitively very believable—we just have to exhibit a vector x satisfying the desired properties. The one concern is that even if all the input numbers are integers, such a vector x may not have integer coordinates, and it may in fact require very large precision to specify: How do we know that we'll be able to read and manipulate it in polynomial time? But, in fact, one can show that if there is a solution, then there is one that is rational and needs only a polynomial number of bits to write down; so this is not a problem.

Linear Programming was also known to be in co-NP for a long time, though this is not as easy to see. Students who have taken a linear programming course may notice that this fact follows from linear programming duality.²

For a long time, indeed, Linear Programming was the most famous example of a problem in both NP and co-NP that was not known to have a polynomial-time solution. Then, in 1981, Leonid Khachiyan, who at the time was a young researcher in the Soviet Union, gave a polynomial-time algorithm for the problem. After some initial concern in the U.S. popular press that this discovery might turn out to be a *Sputnik*-like event in the Cold War (it didn't), researchers settled down to understand exactly what Khachiyan had done. His initial algorithm, while polynomial-time, was in fact quite slow and impractical; but since then practical polynomial-time algorithms—so-called *interior point methods*—have also been developed following the work of Narendra Karmarkar in 1984.

Linear programming is an interesting example for another reason as well. The most widely used algorithm for this problem is the *simplex method*. It works very well in practice and is competitive with polynomial-time interior methods on real-world problems. Yet its worst-case running time is known to be exponential; it is simply that this exponential behavior shows up in practice only very rarely. For all these reasons, linear programming has been a very useful and important example for thinking about the limits of polynomial time as a formal definition of efficiency.

For our purposes here, though, the point is that linear programming problems can be solved in polynomial time, and very efficient algorithms exist in practice. You can learn a lot more about all this in courses on linear programming. The question we ask here is this: How can linear programming help us when we want to solve combinatorial problems such as Vertex Cover?

Vertex Cover as an Integer Program

Recall that a *vertex cover* in a graph $G = (V, E)$ is a set $S \subseteq V$ so that each edge has at least one end in S . In the weighted Vertex Cover Problem, each vertex $i \in V$ has a *weight* $w_i \geq 0$, with the weight of a set S of vertices denoted $w(S) = \sum_{i \in S} w_i$. We would like to find a vertex cover S for which $w(S)$ is minimum.

² Those of you who are familiar with duality may also notice that the *pricing method* of the previous sections is motivated by linear programming duality: the prices are exactly the variables in the dual linear program (which explains why pricing algorithms are often referred to as *primal-dual algorithms*).

We now try to formulate a linear program that is in close correspondence with the Vertex Cover Problem. Thus we consider a graph $G = (V, E)$ with a weight $w_i \geq 0$ on each node i . Linear programming is based on the use of vectors of variables. In our case, we will have a *decision variable* x_i for each node $i \in V$ to model the choice of whether to include node i in the vertex cover; $x_i = 0$ will indicate that node i is not in the vertex cover, and $x_i = 1$ will indicate that node i is in the vertex cover. We can create a single n -dimensional vector x in which the i^{th} coordinate corresponds to the i^{th} decision variable x_i .

We use linear inequalities to encode the requirement that the selected nodes form a vertex cover; we use the objective function to encode the goal of minimizing the total weight. For each edge $(i, j) \in E$, it must have one end in the vertex cover, and we write this as the inequality $x_i + x_j \geq 1$. Finally, to express the minimization problem, we write the set of node weights as an n -dimensional vector w , with the i^{th} coordinate corresponding to w_i ; we then seek to minimize $w^t x$. In summary, we have formulated the Vertex Cover Problem as follows.

$$\begin{aligned} (\text{VC.IP}) \quad \text{Min} \quad & \sum_{i \in V} w_i x_i \\ \text{s.t.} \quad & x_i + x_j \geq 1 \quad (i, j) \in E \\ & x_i \in \{0, 1\} \quad i \in V. \end{aligned}$$

We claim that the vertex covers of G are in one-to-one correspondence with the solutions x to this system of linear inequalities in which all coordinates are equal to 0 or 1.

(11.21) *S is a vertex cover in G if and only if the vector x , defined as $x_i = 1$ for $i \in S$, and $x_i = 0$ for $i \notin S$, satisfies the constraints in (VC.IP). Further, we have $w(S) = w^t x$.*

We can put this system into the matrix form we used for linear programming, as follows. We define a matrix A whose columns correspond to the nodes in V and whose rows correspond to the edges in E ; entry $A[e, i] = 1$ if node i is an end of the edge e , and 0 otherwise. (Note that each row has exactly two nonzero entries.) If we use $\vec{1}$ to denote the vector with all coordinates equal to 1, and $\vec{0}$ to denote the vector with all coordinates equal to 0, then the system of inequalities above can be written as

$$\begin{aligned} Ax &\geq \vec{1} \\ \vec{1} &\geq x \geq \vec{0}. \end{aligned}$$

But keep in mind that this is not just an instance of the Linear Programming Problem: We have crucially required that all coordinates in the solution be either 0 or 1. So our formulation suggests that we should solve the problem

$$\min(w^t x \text{ subject to } \vec{1} \geq x \geq \vec{0}, Ax \geq \vec{1}, x \text{ has integer coordinates}).$$

This is an instance of the Linear Programming Problem in which we require the coordinates of x to take integer values; without this extra constraint, the coordinates of x could be arbitrary real numbers. We call this problem *Integer Programming*, as we are looking for integer-valued solutions to a linear program.

Integer Programming is considerably harder than Linear Programming; indeed, our discussion really constitutes a reduction from Vertex Cover to the decision version of Integer Programming. In other words, we have proved

(11.22) Vertex Cover \leq_P Integer Programming.

To show the NP-completeness of Integer Programming, we would still have to establish that the decision version is in \mathcal{NP} . There is a complication here, as with Linear Programming, since we need to establish that there is always a solution x that can be written using a polynomial number of bits. But this can indeed be proven. Of course, for our purposes, the integer program we are dealing with is explicitly constrained to have solutions in which each coordinate is either 0 or 1. Thus it is clearly in \mathcal{NP} , and our reduction from Vertex Cover establishes that even this special case is NP-complete.

Using Linear Programming for Vertex Cover

We have yet to resolve whether our foray into linear and integer programming will turn out to be useful or simply a dead end. Trying to solve the integer programming problem (VC.IP) optimally is clearly not the right way to go, as this is NP-hard.

The way to make progress is to exploit the fact that Linear Programming is not as hard as Integer Programming. Suppose we take (VC.IP) and modify it, dropping the requirement that each $x_i \in \{0, 1\}$ and reverting to the constraint that each x_i is an arbitrary real number between 0 and 1. This gives us an instance of the Linear Programming Problem that we could call (VC.LP), and we can solve it in polynomial time: We can find a set of values $\{x_i^*\}$ between 0 and 1 so that $x_i^* + x_j^* \geq 1$ for each edge (i, j) , and $\sum_i w_i x_i^*$ is minimized. Let x^* denote this vector, and $w_{LP} = w^t x^*$ denote the value of the objective function.

We note the following basic fact.

(11.23) Let S^* denote a vertex cover of minimum weight. Then $w_{LP} \leq w(S^*)$.

Proof. Vertex covers of G correspond to integer solutions of (VC.IP), so the minimum of $\min(w^t x : \vec{1} \geq x \geq 0, Ax \geq 1)$ over all integer x vectors is exactly the minimum-weight vertex cover. To get the minimum of the linear program (VC.LP), we allow x to take arbitrary real-number values—that is, we minimize over many more choices of x —and so the minimum of (VC.LP) is no larger than that of (VC.IP). ■

Note that (11.23) is one of the crucial ingredients we need for an approximation algorithm: a good lower bound on the optimum, in the form of the efficiently computable quantity w_{LP} .

However, w_{LP} can definitely be smaller than $w(S^*)$. For example, if the graph G is a triangle and all weights are 1, then the minimum vertex cover has a weight of 2. But, in a linear programming solution, we can set $x_i = \frac{1}{2}$ for all three vertices, and so get a linear programming solution of weight only $\frac{3}{2}$. As a more general example, consider a graph on n nodes in which each pair of nodes is connected by an edge. Again, all weights are 1. Then the minimum vertex cover has weight $n - 1$, but we can find a linear programming solution of value $n/2$ by setting $x_i = \frac{1}{2}$ for all vertices i .

So the question is: How can solving this linear program help us actually *find* a near-optimal vertex cover? The idea is to work with the values x_i^* and to infer a vertex cover S from them. It is natural that if $x_i^* = 1$ for some node i , then we should put it in the vertex cover S ; and if $x_i^* = 0$, then we should leave it out of S . But what should we do with fractional values in between? What should we do if $x_i^* = .4$ or $x_i^* = .5$? The natural approach here is to *round*. Given a fractional solution $\{x_i^*\}$, we define $S = \{i \in V : x_i^* \geq \frac{1}{2}\}$ —that is, we round values at least $\frac{1}{2}$ up, and those below $\frac{1}{2}$ down.

(11.24) The set S defined in this way is a vertex cover, and $w(S) \leq w_{LP}$.

Proof. First we argue that S is a vertex cover. Consider an edge $e = (i, j)$. We claim that at least one of i and j must be in S . Recall that one of our inequalities is $x_i + x_j \geq 1$. So in any solution x^* that satisfies this inequality, either $x_i^* \geq \frac{1}{2}$ or $x_j^* \geq \frac{1}{2}$. Thus at least one of these two will be rounded up, and i or j will be placed in S .

Now we consider the weight $w(S)$ of this vertex cover. The set S only has vertices with $x_i^* \geq \frac{1}{2}$; thus the linear program “paid” at least $\frac{1}{2}w_i$ for node i , and we only pay w_i : at most twice as much. More formally, we have the following chain of inequalities.

$$w_{LP}w^t x^* = \sum_i w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i = \frac{1}{2}w(S). \blacksquare$$

Thus we have produced a vertex cover S of weight at most $2w_{LP}$. The lower bound in (11.23) showed that the optimal vertex cover has weight at least w_{LP} , and so we have the following result.

(11.25) *The algorithm produces a vertex cover S of at most twice the minimum possible weight.*

* 11.7 Load Balancing Revisited: A More Advanced LP Application

In this section we consider a more general load balancing problem. We will develop an approximation algorithm using the same general outline as the 2-approximation we just designed for Vertex Cover: We solve a corresponding linear program, and then round the solution. However, the algorithm and its analysis here will be significantly more complex than what was needed for Vertex Cover. It turns out that the instance of the Linear Programming Problem we need to solve is, in fact, a flow problem. Using this fact, we will be able to develop a much deeper understanding of what the fractional solutions to the linear program look like, and we will use this understanding in order to round them. For this problem, the only known constant-factor approximation algorithm is based on rounding this linear programming solution.

The Problem

The problem we consider in this section is a significant, but natural, generalization of the Load Balancing Problem with which we began our study of approximation algorithms. There, as here, we have a set J of n jobs, and a set M of m machines, and the goal is to assign each job to a machine so that the maximum load on any machine will be as small as possible. In the simple Load Balancing Problem we considered earlier, each job j can be assigned to any machine i . Here, on the other hand, we will restrict the set of machines that each job may consider; that is, for each job there is just a subset of machines to which it can be assigned. This restriction arises naturally in a number of applications: for example, we may be seeking to balance load while maintaining the property that each job is assigned to a physically nearby machine, or to a machine with an appropriate authorization to process the job.

More formally, each job j has a fixed given size $t_j \geq 0$ and a set of machines $M_j \subseteq M$ that it may be assigned to. The sets M_j can be completely arbitrary. We call an assignment of jobs to machines *feasible* if each job j is assigned to a machine $i \in M_j$. The goal is still to minimize the maximum load on any machine: Using $J_i \subseteq J$ to denote the jobs assigned to a machine $i \in M$ in a feasible assignment, and using $L_i = \sum_{j \in J_i} t_j$ to denote the resulting load,

we seek to minimize $\max_i L_i$. This is the definition of the *Generalized Load Balancing Problem*.

In addition to containing our initial Load Balancing Problem as a special case (setting $M_j = M$ for all jobs j), Generalized Load Balancing includes the Bipartite Perfect Matching Problem as another special case. Indeed, given a bipartite graph with the same number of nodes on each side, we can view the nodes on the left as jobs and the nodes on the right as machines; we define $t_j = 1$ for all jobs j , and define M_j to be the set of machine nodes i such that there is an edge $(i, j) \in E$. There is an assignment of maximum load 1 if and only if there is a perfect matching in the bipartite graph. (Thus, network flow techniques can be used to find the optimum load in this special case.) The fact that Generalized Load Balancing includes both these problems as special cases gives some indication of the challenge in designing an algorithm for it.



Designing and Analyzing the Algorithm

We now develop an approximation algorithm based on linear programming for the Generalized Load Balancing Problem. The basic plan is the same one we saw in the previous section: we'll first formulate the problem as an equivalent linear program where the variables have to take specific discrete values; we'll then relax this to a linear program by dropping this requirement on the values of the variables; and then we'll use the resulting fractional assignment to obtain an actual assignment that is close to optimal. We'll need to be more careful than in the case of the Vertex Cover Problem in rounding the solution to produce the actual assignment.

Integer and Linear Programming Formulations First we formulate the Generalized Load Balancing Problem as a linear program with restrictions on the variable values. We use variables x_{ij} corresponding to each pair (i, j) of machine $i \in M$ and job $j \in J$. Setting $x_{ij} = 0$ will indicate that job j is not assigned to machine i , while setting $x_{ij} = t_j$ will indicate that all the load t_j of job j is assigned to machine i . We can think of x as a single vector with mn coordinates.

We use linear inequalities to encode the requirement that each job is assigned to a machine: For each job j we require that $\sum_i x_{ij} = t_j$. The load of a machine i can then be expressed as $L_i = \sum_j x_{ij}$. We require that $x_{ij} = 0$ whenever $i \notin M_j$. We will use the objective function to encode the goal of finding an assignment that minimizes the maximum load. To do this, we will need one more variable, L , that will correspond to the load. We use the inequalities $\sum_j x_{ij} \leq L$ for all machines i . In summary, we have formulated the following problem.

$$\begin{aligned}
 (\text{GL.IP}) \quad & \min L \\
 & \sum_i x_{ij} = t_j \quad \text{for all } j \in J \\
 & \sum_j x_{ij} \leq L \quad \text{for all } i \in M \\
 & x_{ij} \in \{0, t_j\} \quad \text{for all } j \in J, i \in M_j \\
 & x_{ij} = 0 \quad \text{for all } j \in J, i \notin M_j.
 \end{aligned}$$

First we claim that the feasible assignments are in one-to-one correspondence with the solutions x satisfying the above constraints, and, in an optimal solution to (GL.IP), L is the load of the corresponding assignment.

(11.26) *An assignment of jobs to machines has load at most L if and only if the vector x , defined by setting $x_{ij} = t_j$ whenever job j is assigned to machine i , and $x_{ij} = 0$ otherwise, satisfies the constraints in (GL.IP), with L set to the maximum load of the assignment.*

Next we will consider the corresponding linear program obtained by replacing the requirement that each $x_{ij} \in \{0, t_j\}$ by the weaker requirement that $x_{ij} \geq 0$ for all $j \in J$ and $i \in M_j$. Let (GL.LP) denote the resulting linear program. It would also be natural to add $x_{ij} \leq t_j$. We do not add these inequalities explicitly, as they are implied by the nonnegativity and the equation $\sum_i x_{ij} = t_j$ that is required for each job j .

We immediately see that if there is an assignment with load at most L , then (GL.LP) must have a solution with value at most L . Or, in the contrapositive,

(11.27) *If the optimum value of (GL.LP) is L , then the optimal load is at least $L^* \geq L$.*

We can use linear programming to obtain such a solution (x, L) in polynomial time. Our goal will then be to use x to create an assignment. Recall that the Generalized Load Balancing Problem is NP-hard, and hence we cannot expect to solve it exactly in polynomial time. Instead, we will find an assignment with load at most two times the minimum possible. To be able to do this, we will also need the simple lower bound (11.2), which we used already in the original Load Balancing Problem.

(11.28) *The optimal load is at least $L^* \geq \max_j t_j$.*

Rounding the Solution When There Are No Cycles The basic idea is to round the x_{ij} values to 0 or t_j . However, we cannot use the simple idea of just rounding large values up and small values down. The problem is that the linear programming solution may assign small fractions of a job j to each of

the m machines, and hence for some jobs there may be no large x_{ij} values. The algorithm we develop will be a rounding of x in the weak sense that each job j will be assigned to a machine i with $x_{ij} > 0$, but we may have to round a few really small values up. This weak rounding already ensures that the assignment is feasible, in the sense that we do not assign any job j to a machine i not in M_j (because if $i \notin M_j$, then we have $x_{ij} = 0$).

The key is to understand what the structure of the fractional solution is like and to show that while a few jobs may be spread out to many machines, this cannot happen to too many jobs. To this end, we'll consider the following bipartite graph $G(x) = (V(x), E(x))$: The nodes are $V(x) = M \cup J$, the set of jobs and the set of machines, and there is an edge $(i, j) \in E(x)$ if and only if $x_{ij} > 0$.

We'll show that, given any solution for (GL.LP), we can obtain a new solution x with the same load L , such that $G(x)$ has no cycles. This is the crucial step, as we show that a solution x with no cycles can be used to obtain an assignment with load at most $L + L^*$.

(11.29) *Given a solution (x, L) of (GL.LP) such that the graph $G(x)$ has no cycles, we can use this solution x to obtain a feasible assignment of jobs to machines with load at most $L + L^*$ in $O(mn)$ time.*

Proof. Since the graph $G(x)$ has no cycles, each of its connected components is a tree. We can produce the assignment by considering each component separately. Thus, consider one of the components, which is a tree whose nodes correspond to jobs and machines, as shown in Figure 11.11.

First, root the tree at an arbitrary node. Now consider a job j . If the node corresponding to job j is a leaf of the tree, let machine node i be its parent. Since j has degree 1 in the tree $G(x)$, machine i is the only machine that has been assigned any part of job j , and hence we must have that $x_{ij} = t_j$. Our assignment will assign such a job j to its only neighbor i . For a job j whose corresponding node is not a leaf in $G(x)$, we assign j to an arbitrary child of the corresponding node in the rooted tree.

The method can clearly be implemented in $O(mn)$ time (including the time to set up the graph $G(x)$). It defines a feasible assignment, as the linear program (GL.LP) required that $x_{ij} = 0$ whenever $i \notin M_j$. To finish the proof, we need to show that the load is at most $L + L^*$. Let i be any machine, and let J_i be the set of jobs assigned to machine i . The jobs assigned to machine i form a subset of the neighbors of i in $G(x)$: the set J_i contains those children of node i that are leaves, plus possibly the parent $p(i)$ of node i . To bound the load, we consider the parent $p(i)$ separately. For all other jobs $j \neq p(i)$ assigned to i , we have $x_{ij} = t_j$, and hence we can bound the load using the solution x , as follows.

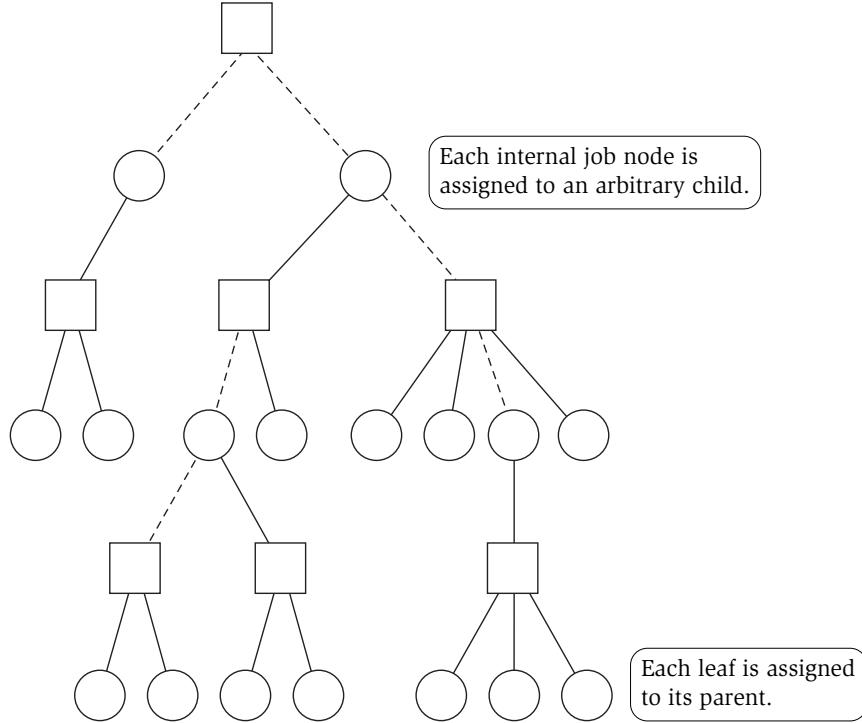


Figure 11.11 An example of a graph $G(x)$ with no cycles, where the squares are machines and the circles are jobs. The solid lines show the resulting assignment of jobs to machines.

$$\sum_{j \in J_i, j \neq p(i)} t_j \leq \sum_{j \in J} x_{ij} \leq L,$$

using the inequality bounding the load in (GL.LP). For the parent $j = p(i)$ of node i , we use $t_j \leq L^*$ by (11.28). Adding the two inequalities, we get that $\sum_{j \in J_i} p_{ij} \leq L + L^*$, as claimed. ■

Now, by (11.27), we know that $L \leq L^*$, so a solution whose load is bounded by $L + L^*$ is also bounded by $2L^*$ —in other words, twice the optimum. Thus we have the following consequence of (11.29).

(11.30) *Given a solution (x, L) of (GL.LP) such that the graph $G(x)$ has no cycles, then we can use this solution x to obtain a feasible assignment of jobs to machines with load at most twice the optimum in $O(mn)$ time.*

Eliminating Cycles from the Linear Programming Solution To wrap up our approximation algorithm, then, we just need to show how to convert

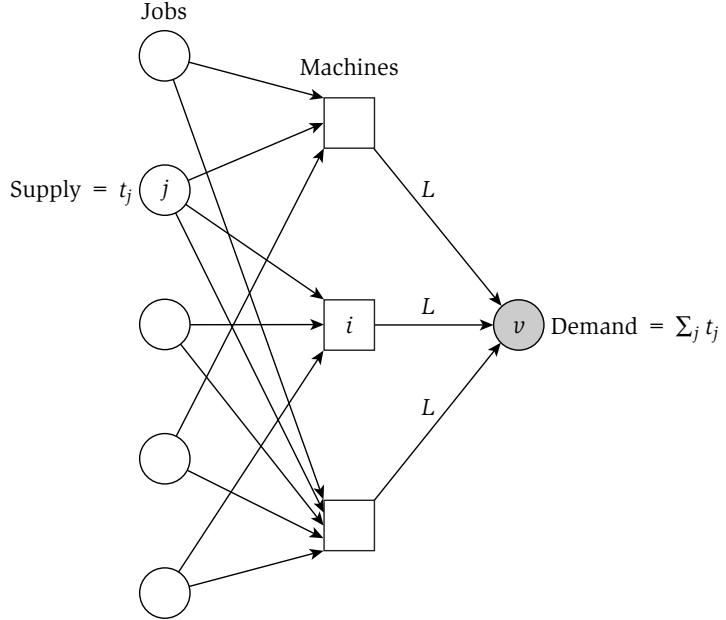


Figure 11.12 The network flow computation used to find a solution to (GL.LP). Edges between the jobs and machines have infinite capacity.

an arbitrary solution of (GL.LP) into a solution x with no cycles in $G(x)$. In the process, we will also show how to obtain a solution to the linear program (GL.LP) using flow computations. More precisely, given a fixed load value L , we show how to use a flow computation to decide if (GL.LP) has a solution with value at most L . For this construction, consider the following directed graph $G = (V, E)$ shown in Figure 11.12. The set of vertices of the flow graph G will be $V = M \cup J \cup \{v\}$, where v is a new node. The nodes $j \in J$ will be sources with supply t_j , and the only demand node is the new sink v , which has demand $\sum_j t_j$. We'll think of the flow in this network as “load” flowing from jobs to the sink v via the machines. We add an edge (j, i) with infinite capacity from job j to machine i if and only if $i \in M_j$. Finally, we add an edge (i, v) for each machine node i with capacity L .

(11.31) *The solutions of this flow problem with capacity L are in one-to-one correspondence with solutions of (GL.LP) with value L , where x_{ij} is the flow value along edge (i, j) , and the flow value on edge (i, v) is the load $\sum_j x_{ij}$ on machine i .*

This statement allows us to solve (GL.LP) using flow computations and a binary search for the optimal value L : we try successive values of L until we find the smallest one for which there is a feasible flow.

Here we'll use the understanding we gained of (GL.LP) from the equivalent flow formulation to modify a solution x to eliminate all cycles from $G(x)$. In terms of the flow we have just defined, $G(x)$ is the undirected graph obtained from G by ignoring the directions of the edges, deleting the sink v and all adjacent edges, and also deleting all edges from J to M that do not carry flow. We'll eliminate all cycles in $G(x)$ in a sequence of at most mn steps, where the goal of a single step is to eliminate at least one edge from $G(x)$ without increasing the load L or introducing any new edges.

(11.32) *Let (x, L) be any solution to (GL.LP) and C be a cycle in $G(x)$. In time linear in the length of the cycle, we can modify the solution x to eliminate at least one edge from $G(x)$ without increasing the load or introducing any new edges.*

Proof. Consider the cycle C in $G(x)$. Recall that $G(x)$ corresponds to the set of edges that carry flow in the solution x . We will modify the solution by *augmenting* the flow along the cycle C , using essentially the procedure `augment` from Section 7.1. The augmentation along a cycle will not change the balance between incoming and outgoing flow at any node; rather, it will eliminate one backward edge from the residual graph, and hence an edge from $G(x)$. Assume that the nodes along the cycle are $i_1, j_1, i_2, j_2, \dots, i_k, j_k$, where i_ℓ is a machine node and j_ℓ is a job node. We'll modify the solution by decreasing the flow along all edges (j_ℓ, i_ℓ) and increasing the flow on the edges $(j_\ell, i_{\ell+1})$ for all $\ell = 1, \dots, k$ (where $k + 1$ is used to denote 1), by the same amount δ . This change will not affect the flow conservation constraints. By setting $\delta = \min_{\ell=1}^k x_{i_\ell j_\ell}$, we ensure that the flow remains feasible and the edge obtaining the minimum is deleted from $G(x)$. ■

We can use the algorithm contained in the proof of (11.32) repeatedly to eliminate all cycles from $G(x)$. Initially, $G(x)$ may have mn edges, so after at most $O(mn)$ iterations, the resulting solution (x, L) will have no cycles in $G(x)$. At this point, we can use (11.30) to obtain a feasible assignment with at most twice the optimal load. We summarize the result by the following statement.

(11.33) *Given an instance of the Generalized Load Balancing Problem, we can find, in polynomial time, a feasible assignment with load at most twice the minimum possible.*

11.8 Arbitrarily Good Approximations: The Knapsack Problem

Often, when you talk to someone faced with an NP-hard optimization problem, they're hoping you can give them something that will produce a solution within, say, 1 percent of the optimum, or at least within a small percentage of optimal. Viewed from this perspective, the approximation algorithms we've seen thus far come across as quite weak: solutions within a factor of 2 of the minimum for Center Selection and Vertex Cover (i.e., 100 percent more than optimal). The Set Cover Algorithm in Section 10.3 is even worse: Its cost is not even within a fixed constant factor of the minimum possible!

Here is an important point underlying this state of affairs: NP-complete problems, as you well know, are all equivalent with respect to polynomial-time solvability; but assuming $\mathcal{P} \neq \mathcal{NP}$, they differ considerably in the extent to which their solutions can be efficiently approximated. In some cases, it is actually possible to prove limits on approximability. For example, if $\mathcal{P} \neq \mathcal{NP}$, then the guarantee provided by our Center Selection Algorithm is the best possible for any polynomial-time algorithm. Similarly, the guarantee provided by the Set Cover Algorithm, however bad it may seem, is very close to the best possible, unless $\mathcal{P} = \mathcal{NP}$. For other problems, such as the Vertex Cover Problem, the approximation algorithm we gave is essentially the best known, but it is an open question whether there could be polynomial-time algorithms with better guarantees. We will not discuss the topic of lower bounds on approximability in this book; while some lower bounds of this type are not so difficult to prove (such as for Center Selection), many are extremely technical.

The Problem

In this section, we discuss an NP-complete problem for which it is possible to design a polynomial-time algorithm providing a very strong approximation. We will consider a slightly more general version of the Knapsack (or Subset Sum) Problem. Suppose you have n items that you consider packing in a knapsack. Each item $i = 1, \dots, n$ has two integer parameters: a weight w_i and a value v_i . Given a knapsack capacity W , the goal of the Knapsack Problem is to find a subset S of items of maximum value subject to the restriction that the total weight of the set should not exceed W . In other words, we wish to maximize $\sum_{i \in S} v_i$ subject to the condition $\sum_{i \in S} w_i \leq W$.

How strong an approximation can we hope for? Our algorithm will take as input the weights and values defining the problem and will also take an extra parameter ϵ , the desired precision. It will find a subset S whose total weight does not exceed W , with value $\sum_{i \in S} v_i$ at most a $(1 + \epsilon)$ factor below the maximum possible. The algorithm will run in polynomial time for any

fixed choice of $\epsilon > 0$; however, the dependence on ϵ will not be polynomial. We call such an algorithm a *polynomial-time approximation scheme*.

You may ask: How could such a strong kind of approximation algorithm be possible in polynomial time when the Knapsack Problem is NP-hard? With integer values, if we get close enough to the optimum value, we must reach the optimum itself! The catch is in the nonpolynomial dependence on the desired precision: for any fixed choice of ϵ , such as $\epsilon = .5$, $\epsilon = .2$, or even $\epsilon = .01$, the algorithm runs in polynomial time, but as we change ϵ to smaller and smaller values, the running time gets larger. By the time we make ϵ small enough to make sure we get the optimum value, it is no longer a polynomial-time algorithm.

Designing the Algorithm

In Section 6.4 we considered algorithms for the Subset Sum Problem, the special case of the Knapsack Problem when $v_i = w_i$ for all items i . We gave a dynamic programming algorithm for this special case that ran in $O(nW)$ time assuming the weights are integers. This algorithm naturally extends to the more general Knapsack Problem (see the end of Section 6.4 for this extension). The algorithm given in Section 6.4 works well when the weights are small (even if the values may be big). It is also possible to extend our dynamic programming algorithm for the case when the values are small, even if the weights may be big. At the end of this section, we give a dynamic programming algorithm for that case running in time $O(n^2v^*)$, where $v^* = \max_i v_i$. Note that this algorithm does not run in polynomial time: It is only pseudo-polynomial, because of its dependence on the size of the values v_i . Indeed, since we proved this problem to be NP-complete in Chapter 8, we don't expect to be able to find a polynomial-time algorithm.

Algorithms that depend on the values in a pseudo-polynomial way can often be used to design polynomial-time approximation schemes, and the algorithm we develop here is a very clean example of the basic strategy. In particular, we will use the dynamic programming algorithm with running time $O(n^2v^*)$ to design a polynomial-time approximation scheme; the idea is as follows. If the values are small integers, then v^* is small and the problem can be solved in polynomial time already. On the other hand, if the values are large, then we do not have to deal with them exactly, as we only want an approximately optimum solution. We will use a rounding parameter b (whose value we'll set later) and will consider the values rounded to an integer multiple of b . We will use our dynamic programming algorithm to solve the problem with the rounded values. More precisely, for each item i , let its rounded value be $\tilde{v}_i = \lceil v_i/b \rceil b$. Note that the rounded and the original value are quite close to each other.

(11.34) For each item i we have $v_i \leq \tilde{v}_i \leq v_i + b$.

What did we gain by the rounding? If the values were big to start with, we did not make them any smaller. However, the rounded values are all integer multiples of a common value b . So, instead of solving the problem with the rounded values \tilde{v}_i , we can change the units; we can divide all values by b and get an equivalent problem. Let $\hat{v}_i = \tilde{v}_i/b = \lceil v_i/b \rceil$ for $i = 1, \dots, n$.

(11.35) The Knapsack Problem with values \tilde{v}_i and the scaled problem with values \hat{v}_i have the same set of optimum solutions, the optimum values differ exactly by a factor of b , and the scaled values are integral.

Now we are ready to state our approximation algorithm. We will assume that all items have weight at most W (as items with weight $w_i > W$ are not in any solution, and hence can be deleted). We also assume for simplicity that ϵ^{-1} is an integer.

```

Knapsack-Approx( $\epsilon$ ) :
  Set  $b = (\epsilon/(2n)) \max_i v_i$ 
  Solve the Knapsack Problem with values  $\hat{v}_i$  (equivalently  $\tilde{v}_i$ )
  Return the set  $S$  of items found

```



Analyzing the Algorithm

First note that the solution found is at least feasible; that is, $\sum_{i \in S} w_i \leq W$. This is true as we have rounded only the values and not the weights. This is why we need the new dynamic programming algorithm described at the end of this section.

(11.36) The set of items S returned by the algorithm has total weight at most W , that is $\sum_{i \in S} w_i \leq W$.

Next we'll prove that this algorithm runs in polynomial time.

(11.37) The algorithm Knapsack-Approx runs in polynomial time for any fixed $\epsilon > 0$.

Proof. Setting b and rounding item values can clearly be done in polynomial time. The time-consuming part of this algorithm is the dynamic programming to solve the rounded problem. Recall that for problems with integer values, the dynamic programming algorithm we use runs in time $O(n^2 v^*)$, where $v^* = \max_i v_i$.

Now we are applying this algorithms for an instance in which each item i has weight w_i and value \hat{v}_i . To determine the running time, we need to

determine $\max_i \hat{v}_i$. The item j with maximum value $v_j = \max_i v_i$ also has maximum value in the rounded problem, so $\max_i \hat{v}_i = \hat{v}_j = \lceil v_j/b \rceil = n\epsilon^{-1}$. Hence the overall running time of the algorithm is $O(n^3\epsilon^{-1})$. Note that this is polynomial time for any fixed $\epsilon > 0$ as claimed; but the dependence on the desired precision ϵ is not polynomial, as the running time includes ϵ^{-1} rather than $\log \epsilon^{-1}$. ■

Finally, we need to consider the key issue: How good is the solution obtained by this algorithm? Statement (11.34) shows that the values \tilde{v}_i we used are close to the real values v_i , and this suggests that the solution obtained may not be far from optimal.

(11.38) *If S is the solution found by the Knapsack-Approx algorithm, and S^* is any other solution satisfying $\sum_{i \in S^*} w_i \leq W$, then we have $(1 + \epsilon) \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$.*

Proof. Let S^* be any set satisfying $\sum_{i \in S^*} w_i \leq W$. Our algorithm finds the optimal solution with values \tilde{v}_i , so we know that

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{i \in S^*} \tilde{v}_i.$$

The rounded values \tilde{v}_i and the real values v_i are quite close by (11.34), so we get the following chain of inequalities.

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} (v_i + b) \leq nb + \sum_{i \in S} v_i,$$

showing that the value $\sum_{i \in S} v_i$ of the solution we obtained is at most nb smaller than the maximum value possible. We wanted to obtain a relative error showing that the value obtained, $\sum_{i \in S} v_i$, is at most a $(1 + \epsilon)$ factor less than the maximum possible, so we need to compare nb to the value $\sum_{i \in S} v_i$.

Let j be the item with largest value; by our choice of b , we have $v_j = 2\epsilon^{-1}nb$ and $v_j = \tilde{v}_j$. By our assumption that each item alone fits in the knapsack ($w_i \leq W$ for all i), we have $\sum_{i \in S} \tilde{v}_i \geq \tilde{v}_j = 2\epsilon^{-1}nb$. Finally, the chain of inequalities above says $\sum_{i \in S} v_i \geq \sum_{i \in S} \tilde{v}_i - nb$, and thus $\sum_{i \in S} v_i \geq (2\epsilon^{-1} - 1)nb$. Hence $nb \leq \epsilon \sum_{i \in S} v_i$ for $\epsilon \leq 1$, and so

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i + nb \leq (1 + \epsilon) \sum_{i \in S} v_i. \blacksquare$$

The New Dynamic Programming Algorithm for the Knapsack Problem

To solve a problem by dynamic programming, we have to define a polynomial set of subproblems. The dynamic programming algorithm we defined when we studied the Knapsack Problem earlier uses subproblems of the form $\text{OPT}(i, w)$: the subproblem of finding the maximum value of any solution using a subset of the items $1, \dots, i$ and a knapsack of weight w . When the weights are large, this is a large set of problems. We need a set of subproblems that work well when the values are reasonably small; this suggests that we should use subproblems associated with values, not weights. We define our subproblems as follows. The subproblem is defined by i and a target value V , and $\overline{\text{OPT}}(i, V)$ is the smallest knapsack weight W so that one can obtain a solution using a subset of items $\{1, \dots, i\}$ with value at least V . We will have a subproblem for all $i = 0, \dots, n$ and values $V = 0, \dots, \sum_{j=1}^i v_j$. If v^* denotes $\max_i v_i$, then we see that the largest V can get in a subproblem is $\sum_{j=1}^n v_j \leq nv^*$. Thus, assuming the values are integral, there are at most $O(n^2v^*)$ subproblems. None of these subproblems is precisely the original instance of Knapsack, but if we have the values of all subproblems $\overline{\text{OPT}}(n, V)$ for $V = 0, \dots, \sum_i v_i$, then the value of the original problem can be obtained easily: it is the largest value V such that $\overline{\text{OPT}}(n, V) \leq W$.

It is not hard to give a recurrence for solving these subproblems. By analogy with the dynamic programming algorithm for Subset Sum, we consider cases depending on whether or not the last item n is included in the optimal solution \mathcal{O} .

- If $n \notin \mathcal{O}$, then $\overline{\text{OPT}}(n, V) = \overline{\text{OPT}}(n - 1, V)$.
- If $n \in \mathcal{O}$ is the only item in \mathcal{O} , then $\overline{\text{OPT}}(n, V) = w_n$.
- If $n \in \mathcal{O}$ is not the only item in \mathcal{O} , then $\overline{\text{OPT}}(n, V) = w_n + \overline{\text{OPT}}(n - 1, V - v_n)$.

These last two options can be summarized more compactly as

- If $n \in \mathcal{O}$, then $\overline{\text{OPT}}(n, V) = w_n + \overline{\text{OPT}}(n - 1, \max(0, V - v_n))$.

This implies the following analogue of the recurrence (6.8) from Chapter 6.

(11.39) *If $V > \sum_{i=1}^{n-1} v_i$, then $\overline{\text{OPT}}(n, V) = w_n + \overline{\text{OPT}}(n - 1, V - v_n)$. Otherwise $\overline{\text{OPT}}(n, V) = \min(\overline{\text{OPT}}(n - 1, V), w_n + \overline{\text{OPT}}(n - 1, \max(0, V - v_n)))$.*

We can then write down an analogous dynamic programming algorithm.

```

Knapsack(n) :
  Array M[0 ... n, 0 ... V]
    
```

```

For i = 0, . . . , n
    M[i, 0] = 0
Endfor
For i = 1, 2, . . . , n
    For V = 1, . . . ,  $\sum_{j=1}^i v_j$ 
        If V >  $\sum_{j=1}^{i-1} v_j$  then
            M[i, V] =  $w_i + M[i - 1, V]$ 
        Else
            M[i, V] = min(M[i - 1, V],  $w_i + M[i - 1, \max(0, V - v_i)]$ )
        Endif
    Endfor
Endfor
Return the maximum value V such that M[n, V] ≤ W

```

(11.40) Knapsack(n) takes $O(n^2v^*)$ time and correctly computes the optimal values of the subproblems.

As was done before, we can trace back through the table M containing the optimal values of the subproblems, to find an optimal solution.

Solved Exercises

Solved Exercise 1

Recall the Shortest-First greedy algorithm for the Interval Scheduling Problem: Given a set of intervals, we repeatedly pick the shortest interval I , delete all the other intervals I' that intersect I , and iterate.

In Chapter 4, we saw that this algorithm does *not* always produce a maximum-size set of nonoverlapping intervals. However, it turns out to have the following interesting approximation guarantee. If s^* is the maximum size of a set of nonoverlapping intervals, and s is the size of the set produced by the Shortest-First Algorithm, then $s \geq \frac{1}{2}s^*$ (that is, Shortest-First is a 2-approximation).

Prove this fact.

Solution Let's first recall the example in Figure 4.1 from Chapter 4, which showed that Shortest-First does not necessarily find an optimal set of intervals. The difficulty is clear: We may select a short interval j while eliminating two longer flanking intervals i and i' . So we have done only half as well as the optimum.

The question is to show that *Shortest-First* could never do worse than this. The issues here are somewhat similar to what came up in the analysis of the

greedy algorithm for the Maximum Disjoint Paths Problem in Section 11.5: Each interval we select may “block” some of the intervals in an optimal solution, and we want to argue that by always selecting the shortest possible interval, these blocking effects are not too severe. In the case of disjoint paths, we analyzed the overlaps among paths essentially edge by edge, since the underlying graph there had an arbitrary structure. Here we can benefit from the highly restricted structure of intervals on a line so as to obtain a stronger bound.

In order for Shortest-First to do less than half as well as the optimum, there would have to be a large optimal solution that overlaps with a much smaller solution chosen by Shortest-First. Intuitively, it seems that the only way this could happen would be to have one of the intervals i in the optimal solution nested completely inside one of the intervals j chosen by Shortest-First. This in turn would contradict the behavior of Shortest-First: Why didn’t it choose this shorter interval i that’s nested inside j ?

Let’s see if we can make this argument precise. Let A denote the set of intervals chosen by Shortest-First, and let \mathcal{O} denote an optimal set of intervals. For each interval $j \in A$, consider the set of intervals in \mathcal{O} that it conflicts with. We claim

(11.41) *Each interval $j \in A$ conflicts with at most two intervals in \mathcal{O} .*

Proof. Assume by way of contradiction that there is an interval in $j \in A$ that conflicts with at least three intervals in $i_1, i_2, i_3 \in \mathcal{O}$. These three intervals do not conflict with one another, as they are part of a single solution \mathcal{O} , so they are ordered sequentially in time. Suppose they are ordered with i_1 first, then i_2 , and then i_3 . Since interval j conflicts with both i_1 and i_3 , the interval i_2 in between must be shorter than j and fit completely inside it. Moreover, since i_2 was never selected by Shortest-First, it must have been available as an option when Shortest-First selected interval j . This is a contradiction, since i_2 is shorter than j . ■

The Shortest-First Algorithm only terminates when every unselected interval conflicts with one of the intervals it selected. So, in particular, each interval in \mathcal{O} is either included in A , or conflicts with an interval in A .

Now we use the following accounting scheme to bound the number of intervals in \mathcal{O} . For each $i \in \mathcal{O}$, we have some interval $j \in A$ “pay” for i , as follows. If i is also in A , then i will pay for itself. Otherwise, we arbitrarily choose an interval $j \in A$ that conflicts with i and have j pay for i . As we just argued, every interval in \mathcal{O} conflicts with some interval in A , so all intervals in \mathcal{O} will be paid for under this scheme. But by (11.41), each interval $j \in A$ conflicts with at most two intervals in \mathcal{O} , and so it will only pay for at most

two intervals. Thus, all intervals in \mathcal{O} are paid for by intervals in A , and in this process each interval in A pays at most twice. It follows that A must have at least half as many intervals as \mathcal{O} .

Exercises

1. Suppose you're acting as a consultant for the Port Authority of a small Pacific Rim nation. They're currently doing a multi-billion-dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port.

Here's a basic sort of problem they face. A ship arrives, with n containers of weight w_1, w_2, \dots, w_n . Standing on the dock is a set of trucks, each of which can hold K units of weight. (You can assume that K and each w_i is an integer.) You can stack multiple containers in each truck, subject to the weight restriction of K ; the goal is to minimize the number of trucks that are needed in order to carry all the containers. This problem is NP-complete (you don't have to prove this).

A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers 1, 2, 3, ... into it until you get to a container that would overflow the weight limit. Now declare this truck "loaded" and send it off; then continue the process with a fresh truck. This algorithm, by considering trucks one at a time, may not achieve the most efficient way to pack the full set of containers into an available collection of trucks.

- (a) Give an example of a set of weights, and a value of K , where this algorithm does not use the minimum possible number of trucks.
(b) Show, however, that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K .
2. At a lecture in a computational biology conference one of us attended a few years ago, a well-known protein chemist talked about the idea of building a "representative set" for a large collection of protein molecules whose properties we don't understand. The idea would be to intensively study the proteins in the representative set and thereby learn (by inference) about all the proteins in the full collection.

To be useful, the representative set must have two properties.

- It should be relatively small, so that it will not be too expensive to study it.

- Every protein in the full collection should be “similar” to some protein in the representative set. (In this way, it truly provides some information about all the proteins.)

More concretely, there is a large set P of proteins. We define similarity on proteins by a *distance function* d : Given two proteins p and q , it returns a number $d(p, q) \geq 0$. In fact, the function $d(\cdot, \cdot)$ most typically used is the *sequence alignment* measure, which we looked at when we studied dynamic programming in Chapter 6. We’ll assume this is the distance being used here. There is a predefined distance cut-off Δ that’s specified as part of the input to the problem; two proteins p and q are deemed to be “similar” to one another if and only if $d(p, q) \leq \Delta$.

We say that a subset of P is a *representative set* if, for every protein p , there is a protein q in the subset that is similar to it—that is, for which $d(p, q) \leq \Delta$. Our goal is to find a representative set that is as small as possible.

- Give a polynomial-time algorithm that approximates the minimum representative set to within a factor of $O(\log n)$. Specifically, your algorithm should have the following property: If the minimum possible size of a representative set is s^* , your algorithm should return a representative set of size at most $O(s^* \log n)$.
 - Note the close similarity between this problem and the Center Selection Problem—a problem for which we considered approximation algorithms in Section 11.2. Why doesn’t the algorithm described there solve the current problem?
3. Suppose you are given a set of positive integers $A = \{a_1, a_2, \dots, a_n\}$ and a positive integer B . A subset $S \subseteq A$ is called *feasible* if the sum of the numbers in S does not exceed B :

$$\sum_{a_i \in S} a_i \leq B.$$

The sum of the numbers in S will be called the *total sum* of S .

You would like to select a feasible subset S of A whose total sum is as large as possible.

Example. If $A = \{8, 2, 4\}$ and $B = 11$, then the optimal solution is the subset $S = \{8, 2\}$.

- Here is an algorithm for this problem.

```

Initially  $S = \emptyset$ 
Define  $T = 0$ 
For  $i = 1, 2, \dots, n$ 
```

```

If  $T + a_i \leq B$  then
     $S \leftarrow S \cup \{a_i\}$ 
     $T \leftarrow T + a_i$ 
Endif
Endfor

```

Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A .

- (b) Give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$. Your algorithm should have a running time of at most $O(n \log n)$.
4. Consider an optimization version of the Hitting Set Problem defined as follows. We are given a set $A = \{a_1, \dots, a_n\}$ and a collection B_1, B_2, \dots, B_m of subsets of A . Also, each element $a_i \in A$ has a *weight* $w_i \geq 0$. The problem is to find a hitting set $H \subseteq A$ such that the total weight of the elements in H , that is, $\sum_{a_i \in H} w_i$, is as small as possible. (As in Exercise 5 in Chapter 8, we say that H is a hitting set if $H \cap B_i$ is not empty for each i .) Let $b = \max_i |B_i|$ denote the maximum size of any of the sets B_1, B_2, \dots, B_m . Give a polynomial-time approximation algorithm for this problem that finds a hitting set whose total weight is at most b times the minimum possible.
5. You are asked to consult for a business where clients bring in jobs each day for processing. Each job has a processing time t_i that is known when the job arrives. The company has a set of ten machines, and each job can be processed on any of these ten machines.

At the moment the business is running the simple Greedy-Balance Algorithm we discussed in Section 11.1. They have been told that this may not be the best approximation algorithm possible, and they are wondering if they should be afraid of bad performance. However, they are reluctant to change the scheduling as they really like the simplicity of the current algorithm: jobs can be assigned to machines as soon as they arrive, without having to defer the decision until later jobs arrive.

In particular, they have heard that this algorithm can produce solutions with makespan as much as twice the minimum possible; but their experience with the algorithm has been quite good: They have been running it each day for the last month, and they have not observed it to produce a makespan more than 20 percent above the average load, $\frac{1}{10} \sum_i t_i$.

To try understanding why they don't seem to be encountering this factor-of-two behavior, you ask a bit about the kind of jobs and loads they see. You find out that the sizes of jobs range between 1 and 50, that is, $1 \leq t_i \leq 50$ for all jobs i ; and the total load $\sum_i t_i$ is quite high each day: it is always at least 3,000.

Prove that on the type of inputs the company sees, the Greedy-Balance Algorithm will always find a solution whose makespan is at most 20 percent above the average load.

6. Recall that in the basic Load Balancing Problem from Section 11.1, we're interested in placing jobs on machines so as to minimize the *makespan*—the maximum load on any one machine. In a number of applications, it is natural to consider cases in which you have access to machines with different amounts of processing power, so that a given job may complete more quickly on one of your machines than on another. The question then becomes: How should you allocate jobs to machines in these more heterogeneous systems?

Here's a basic model that exposes these issues. Suppose you have a system that consists of m *slow* machines and k *fast* machines. The fast machines can perform twice as much work per unit time as the slow machines. Now you're given a set of n jobs; job i takes time t_i to process on a slow machine and time $\frac{1}{2}t_i$ to process on a fast machine. You want to assign each job to a machine; as before, the goal is to minimize the makespan—that is the maximum, over all machines, of the total processing time of jobs assigned to that machine.

Give a polynomial-time algorithm that produces an assignment of jobs to machines with a makespan that is at most three times the optimum.

7. You're consulting for an e-commerce site that receives a large number of visitors each day. For each visitor i , where $i \in \{1, 2, \dots, n\}$, the site has assigned a value v_i , representing the expected revenue that can be obtained from this customer.

Each visitor i is shown one of m possible ads A_1, A_2, \dots, A_m as they enter the site. The site wants a selection of one ad for each customer so that *each* ad is seen, overall, by a set of customers of reasonably large total weight. Thus, given a selection of one ad for each customer, we will define the *spread* of this selection to be the minimum, over $j = 1, 2, \dots, m$, of the total weight of all customers who were shown ad A_j .

Example Suppose there are six customers with values 3, 4, 12, 2, 4, 6, and there are $m = 3$ ads. Then, in this instance, one could achieve a spread of

9 by showing ad A_1 to customers 1, 2, 4, ad A_2 to customer 3, and ad A_3 to customers 5 and 6.

The ultimate goal is to find a selection of an ad for each customer that maximizes the spread. Unfortunately, this optimization problem is NP-hard (you don't have to prove this). So instead, we will try to approximate it.

- (a) Give a polynomial-time algorithm that approximates the maximum spread to within a factor of 2. That is, if the maximum spread is s , then your algorithm should produce a selection of one ad for each customer that has spread at least $s/2$. In designing your algorithm, you may assume that no single customer has a value that is significantly above the average; specifically, if $\bar{v} = \sum_{i=1}^n v_i$ denotes the total value of all customers, then you may assume that no single customer has a value exceeding $\bar{v}/(2m)$.
 - (b) Give an example of an instance on which the algorithm you designed in part (a) does not find an optimal solution (that is, one of maximum spread). Say what the optimal solution is in your sample instance, and what your algorithm finds.
8. Some friends of yours are working with a system that performs real-time scheduling of jobs on multiple servers, and they've come to you for help in getting around an unfortunate piece of legacy code that can't be changed.

Here's the situation. When a batch of jobs arrives, the system allocates them to servers using the simple Greedy-Balance Algorithm from Section 11.1, which provides an approximation to within a factor of 2. In the decade and a half since this part of the system was written, the hardware has gotten faster to the point where, on the instances that the system needs to deal with, your friends find that it's generally possible to compute an optimal solution.

The difficulty is that the people in charge of the system's internals won't let them change the portion of the software that implements the Greedy-Balance Algorithm so as to replace it with one that finds the optimal solution. (Basically, this portion of the code has to interact with so many other parts of the system that it's not worth the risk of something going wrong if it's replaced.)

After grumbling about this for a while, your friends come up with an alternate idea. Suppose they could write a little piece of code that takes the description of the jobs, computes an optimal solution (since they're able to do this on the instances that arise in practice), and then feeds the jobs to the Greedy-Balance Algorithm *in an order that will cause it to allocate them optimally*. In other words, they're hoping to be able to

reorder the input in such a way that when Greedy-Balance encounters the input in this order, it produces an optimal solution.

So their question to you is simply the following: Is this always possible? Their conjecture is,

For every instance of the load balancing problem from Section 11.1, there exists an order of the jobs so that when Greedy-Balance processes the jobs in this order, it produces an assignment of jobs to machines with the minimum possible makespan.

Decide whether you think this conjecture is true or false, and give either a proof or a counterexample.

9. Consider the following maximization version of the 3-Dimensional Matching Problem. Given disjoint sets X , Y , and Z , and given a set $T \subseteq X \times Y \times Z$ of ordered triples, a subset $M \subseteq T$ is a *3-dimensional matching* if each element of $X \cup Y \cup Z$ is contained in at most one of these triples. The *Maximum 3-Dimensional Matching Problem* is to find a 3-dimensional matching M of maximum size. (The size of the matching, as usual, is the number of triples it contains. You may assume $|X| = |Y| = |Z|$ if you want.)

Give a polynomial-time algorithm that finds a 3-dimensional matching of size at least $\frac{1}{3}$ times the maximum possible size.

10. Suppose you are given an $n \times n$ *grid graph* G , as in Figure 11.13.

Associated with each node v is a *weight* $w(v)$, which is a nonnegative integer. You may assume that the weights of all nodes are distinct. Your

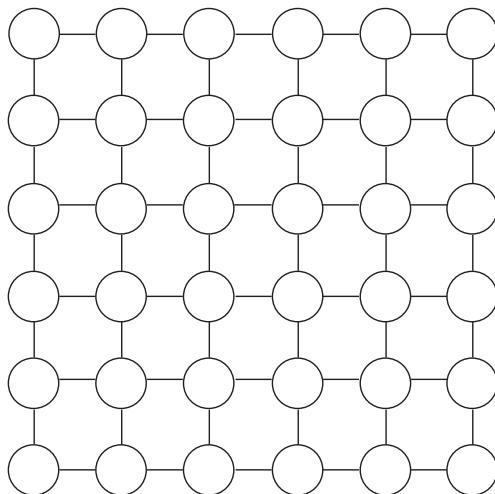


Figure 11.13 A grid graph.

goal is to choose an independent set S of nodes of the grid, so that the sum of the weights of the nodes in S is as large as possible. (The sum of the weights of the nodes in S will be called its *total weight*.)

Consider the following greedy algorithm for this problem.

The "heaviest-first" greedy algorithm:

```

Start with  $S$  equal to the empty set
While some node remains in  $G$ 
  Pick a node  $v_i$  of maximum weight
  Add  $v_i$  to  $S$ 
  Delete  $v_i$  and its neighbors from  $G$ 
Endwhile
Return  $S$ 
```

- (a) Let S be the independent set returned by the “heaviest-first” greedy algorithm, and let T be any other independent set in G . Show that, for each node $v \in T$, either $v \in S$, or there is a node $v' \in S$ so that $w(v) \leq w(v')$ and (v, v') is an edge of G .
 - (b) Show that the “heaviest-first” greedy algorithm returns an independent set of total weight at least $\frac{1}{4}$ times the maximum total weight of any independent set in the grid graph G .
11. Recall that in the Knapsack Problem, we have n items, each with a weight w_i and a value v_i . We also have a weight bound W , and the problem is to select a set of items S of highest possible value subject to the condition that the total weight does not exceed W —that is, $\sum_{i \in S} w_i \leq W$. Here’s one way to look at the approximation algorithm that we designed in this chapter. If we are told there exists a subset \emptyset whose total weight is $\sum_{i \in \emptyset} w_i \leq W$ and whose total value is $\sum_{i \in \emptyset} v_i = V$ for some V , then our approximation algorithm can find a set \mathcal{A} with total weight $\sum_{i \in \mathcal{A}} w_i \leq W$ and total value at least $\sum_{i \in \mathcal{A}} v_i \geq V/(1 + \epsilon)$. Thus the algorithm approximates the best value, while keeping the weights strictly under W . (Of course, returning the set \emptyset is always a valid solution, but since the problem is NP-hard, we don’t expect to always be able to find \emptyset itself; the approximation bound of $1 + \epsilon$ means that other sets \mathcal{A} , with slightly less value, can be valid answers as well.)

Now, as is well known, you can always pack a little bit more for a trip just by “sitting on your suitcase”—in other words, by slightly overflowing the allowed weight limit. This too suggests a way of formalizing the approximation question for the Knapsack Problem, but it’s the following, different, formalization.

Suppose, as before, that you're given n items with weights and values, as well as parameters W and V ; and you're told that there is a subset \mathcal{O} whose total weight is $\sum_{i \in \mathcal{O}} w_i \leq W$ and whose total value is $\sum_{i \in \mathcal{O}} v_i = V$ for some V . For a given fixed $\epsilon > 0$, design a polynomial-time algorithm that finds a subset of items \mathcal{A} such that $\sum_{i \in \mathcal{A}} w_i \leq (1 + \epsilon)W$ and $\sum_{i \in \mathcal{A}} v_i \geq V$. In other words, you want \mathcal{A} to achieve at least as high a total value as the given bound V , but you're allowed to exceed the weight limit W by a factor of $1 + \epsilon$.

Example. Suppose you're given four items, with weights and values as follows:

$$(w_1, v_1) = (5, 3), (w_2, v_2) = (4, 6)$$

$$(w_3, v_3) = (1, 4), (w_4, v_4) = (6, 11)$$

You're also given $W = 10$ and $V = 13$ (since, indeed, the subset consisting of the first three items has total weight at most 10 and has value 13). Finally, you're given $\epsilon = .1$. This means you need to find (via your approximation algorithm) a subset of weight at most $(1 + .1) * 10 = 11$ and value at least 13. One valid solution would be the subset consisting of the first and fourth items, with value $14 \geq 13$. (Note that this is a case where you're able to achieve a value strictly greater than V , since you're allowed to slightly overfill the knapsack.)

12. Consider the following problem. There is a set U of n nodes, which we can think of as users (e.g., these are locations that need to access a service, such as a Web server). You would like to place servers at multiple locations. Suppose you are given a set S possible sites that would be willing to act as locations for the servers. For each site $s \in S$, there is a fee $f_s \geq 0$ for placing a server at that location. Your goal will be to approximately minimize the cost while providing the service to each of the customers. So far this is very much like the Set Cover Problem: The places s are sets, the weight of set s is f_s , and we want to select a collection of sets that covers all users. There is one extra complication: Users $u \in U$ can be served from multiple sites, but there is an associated cost d_{us} for serving user u from site s . When the value d_{us} is very high, we do not want to serve user u from site s ; and in general the service cost d_{us} serves as an incentive to serve customers from “nearby” servers whenever possible.

So here is the question, which we call the Facility Location Problem: Given the sets U and S , and costs f and d , you need to select a subset $A \subseteq S$ at which to place servers (at a cost of $\sum_{s \in A} f_s$), and assign each user u to the active server where it is cheapest to be served, $\min_{s \in A} d_{us}$. The goal

is to minimize the overall cost $\sum_{s \in A} f_s + \sum_{u \in U} \min_{s \in A} d_{us}$. Give an $H(n)$ -approximation for this problem.

(Note that if all service costs d_{us} are 0 or infinity, then this problem is exactly the Set Cover Problem: f_s is the cost of the set named s , and d_{us} is 0 if node u is in set s , and infinity otherwise.)

Notes and Further Reading

The design of approximation algorithms for NP-hard problems is an active area of research, and it is the focus of a book of surveys edited by Hochbaum (1996) and a text by Vazirani (2001).

The greedy algorithm for load balancing and its analysis is due to Graham (1966, 1969); in fact, he proved that when the jobs are first sorted in descending order of size, the greedy algorithm achieves an assignment within a factor $\frac{4}{3}$ of optimal. (In the text, we give a simpler proof for the weaker bound of $\frac{3}{2}$.) Using more complicated algorithms, even stronger approximation guarantees can be proved for this problem (Hochbaum and Shmoys 1987; Hall 1996). The techniques used for these stronger load balancing approximation algorithms are also closely related to the method described in the text for designing arbitrarily good approximations for the Knapsack Problem.

The approximation algorithm for the Center Selection Problem follows the approach of Hochbaum and Shmoys (1985) and Dyer and Frieze (1985). Other geometric location problems of this flavor are discussed by Bern and Eppstein (1996) and in the book of surveys edited by Drezner (1995).

The greedy algorithm for Set Cover and its analysis are due independently to Johnson (1974), Lovász (1975), and Chvatal (1979). Further results for the Set Cover Problem are discussed in the survey by Hochbaum (1996).

As mentioned in the text, the pricing method for designing approximation algorithms is also referred to as the *primal-dual method* and can be motivated using linear programming. This latter perspective is the subject of the survey by Goemans and Williamson (1996). The pricing algorithm to approximate the Weighted Vertex Cover Problem is due to Bar-Yehuda and Even (1981).

The greedy algorithm for the disjoint paths problem is due to Kleinberg and Tardos (1995); the pricing-based approximation algorithm for the case when multiple paths can share an edge is due to Awerbuch, Azar, and Plotkin (1993). Algorithms have been developed for many other variants of the Disjoint Paths Problem; see the book of surveys edited by Korte et al. (1990) for a discussion of cases that can be solved optimally in polynomial time, and Plotkin (1995) and Kleinberg (1996) for surveys of work on approximation.

The linear programming rounding algorithm for the Weighted Vertex Cover Problem is due to Hochbaum (1982). The rounding algorithm for Generalized Load Balancing is due to Lenstra, Shmoys, and Tardos (1990); see the survey by Hall (1996) for other results in this vein. As discussed in the text, these two results illustrate a widely used method for designing approximation algorithms: One sets up an integer programming formulation for the problem, transforms it to a related (but not equivalent) linear programming problem, and then rounds the resulting solution. Vazirani (2001) discusses many further applications of this technique.

Local search and randomization are two other powerful techniques for designing approximation algorithms; we discuss these connections in the next two chapters.

One topic that we do not cover in this book is *inapproximability*. Just as one can prove that a given NP-hard problem can be approximated to within a certain factor in polynomial time, one can also sometimes establish lower bounds, showing that if the problem could be approximated to within better than some factor c in polynomial time, then it could be solved optimally, thereby proving $\mathcal{P} = \mathcal{NP}$. There is a growing body of work that establishes such limits to approximability for many NP-hard problems. In certain cases, these positive and negative results have lined up perfectly to produce an *approximation threshold*, establishing for certain problems that there is a polynomial-time approximation algorithm to within some factor c , and it is impossible to do better unless $\mathcal{P} = \mathcal{NP}$. Some of the early results on inapproximability were not very difficult to prove, but more recent work has introduced powerful techniques that become quite intricate. This topic is covered in the survey by Arora and Lund (1996).

Notes on the Exercises Exercises 4 and 12 are based on results of Dorit Hochbaum. Exercise 11 is based on results of Sartaj Sahni, Oscar Ibarra, and Chul Kim, and of Dorit Hochbaum and David Shmoys.

2 Set Cover

The set cover problem plays the same role in approximation algorithms that the maximum matching problem played in exact algorithms – as a problem whose study led to the development of fundamental techniques for the entire field. For our purpose this problem is particularly useful, since it offers a very simple setting in which many of the basic algorithm design techniques can be explained with great ease. In this chapter, we will cover two combinatorial techniques: the fundamental greedy technique and the technique of layering. In Part II we will explain both the basic LP-based techniques of rounding and the primal–dual schema using this problem.

Among the first strategies one tries when designing an algorithm for an optimization problem is some form of the greedy strategy. Even if this strategy does not work for a specific problem, proving this via a counterexample can provide crucial insights into the structure of the problem.

Perhaps the most natural use of this strategy in approximation algorithms is to the set cover problem. Besides the greedy set cover algorithm, we will also present the technique of layering in this chapter. Because of its generality, the set cover problem has wide applicability, sometimes even in unexpected ways. In this chapter we will illustrate such an application – to the shortest superstring problem (see Chapter 7 for an improved algorithm for the latter problem).

Problem 2.1 (Set cover) Given a universe U of n elements, a collection of subsets of U , $\mathcal{S} = \{S_1, \dots, S_k\}$, and a cost function $c : \mathcal{S} \rightarrow \mathbf{Q}^+$, find a minimum cost subcollection of \mathcal{S} that covers all elements of U .

Define the *frequency* of an element to be the number of sets it is in. A useful parameter is the frequency of the most frequent element. Let us denote this by f . The various approximation algorithms for set cover achieve one of two factors: $O(\log n)$ or f . Clearly, neither dominates the other in all instances. The special case of set cover with $f = 2$ is essentially the vertex cover problem (see Exercise 2.7), for which we gave a factor 2 approximation algorithm in Chapter 1.

2.1 The greedy algorithm

The greedy strategy applies naturally to the set cover problem: iteratively pick the most cost-effective set and remove the covered elements, until all elements are covered. Let C be the set of elements already covered at the beginning of an iteration. During this iteration, define the *cost-effectiveness* of a set S to be the average cost at which it covers new elements, i.e., $c(S)/|S - C|$. Define the *price* of an element to be the average cost at which it is covered. Equivalently, when a set S is picked, we can think of its cost being distributed equally among the new elements covered, to set their prices.

Algorithm 2.2 (Greedy set cover algorithm)

1. $C \leftarrow \emptyset$
2. While $C \neq U$ do
 - Find the most cost-effective set in the current iteration, say S .
 - Let $\alpha = \frac{c(S)}{|S - C|}$, i.e., the cost-effectiveness of S .
 - Pick S , and for each $e \in S - C$, set $\text{price}(e) = \alpha$.
 - $C \leftarrow C \cup S$.
3. Output the picked sets.

Number the elements of U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let e_1, \dots, e_n be this numbering.

Lemma 2.3 *For each $k \in \{1, \dots, n\}$, $\text{price}(e_k) \leq \text{OPT}/(n - k + 1)$.*

Proof: In any iteration, the leftover sets of the optimal solution can cover the remaining elements at a cost of at most OPT . Therefore, among these sets, there must be one having cost-effectiveness of at most $\text{OPT}/|\bar{C}|$. In the iteration in which element e_k was covered, \bar{C} contained at least $n - k + 1$ elements. Since e_k was covered by the most cost-effective set in this iteration, it follows that

$$\text{price}(e_k) \leq \frac{\text{OPT}}{|\bar{C}|} \leq \frac{\text{OPT}}{n - k + 1}.$$

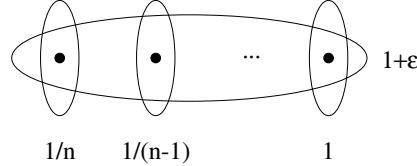
□

From Lemma 2.3 we immediately obtain:

Theorem 2.4 *The greedy algorithm is an H_n factor approximation algorithm for the minimum set cover problem, where $H_n = 1 + \frac{1}{2} + \dots + \frac{1}{n}$.*

Proof: Since the cost of each set picked is distributed among the new elements covered, the total cost of the set cover picked is equal to $\sum_{k=1}^n \text{price}(e_k)$. By Lemma 2.3, this is at most $(1 + \frac{1}{2} + \dots + \frac{1}{n}) \cdot \text{OPT}$. □

Example 2.5 The following is a tight example for Algorithm 2.2:



When run on this instance the greedy algorithm outputs the cover consisting of the n singleton sets, since in each iteration some singleton is the most cost-effective set. Thus, the algorithm outputs a cover of cost

$$\frac{1}{n} + \frac{1}{n-1} + \cdots + 1 = H_n.$$

On the other hand, the optimal cover has a cost of $1 + \varepsilon$. \square

Surprisingly enough, for the minimum set cover problem the obvious algorithm given above is essentially the best one can hope for; see Sections 29.7 and 29.9.

In Chapter 1 we pointed out that finding a good lower bound on OPT is a basic starting point in the design of an approximation algorithm for a minimization problem. At this point the reader may be wondering whether there is any truth to this claim. We will show in Section 13.1 that the correct way to view the greedy set cover algorithm is in the setting of the LP-duality theory – this will not only provide the lower bound on which this algorithm is based, but will also help obtain algorithms for several generalizations of this problem.

2.2 Layering

The algorithm design technique of layering is also best introduced via set cover. We note, however, that this is not a very widely applicable technique. We will give a factor 2 approximation algorithm for vertex cover, assuming arbitrary weights, and leave the problem of generalizing this to a factor f approximation algorithm for set cover, where f is the frequency of the most frequent element (see Exercise 2.13).

The idea in layering is to decompose the given weight function on vertices into convenient functions, called degree-weighted, on a nested sequence of subgraphs of G . For degree-weighted functions, we will show that we will be within twice the optimal even if we pick all vertices in the cover.

Let us introduce some notation. Let $w : V \rightarrow \mathbf{Q}^+$ be the function assigning weights to the vertices of the given graph $G = (V, E)$. We will say that a function assigning vertex weights is *degree-weighted* if there is a constant

$c > 0$ such that the weight of each vertex $v \in V$ is $c \cdot \deg(v)$. The significance of such a weight function is captured in:

Lemma 2.6 *Let $w : V \rightarrow \mathbf{Q}^+$ be a degree-weighted function. Then $w(V) \leq 2 \cdot \text{OPT}$.*

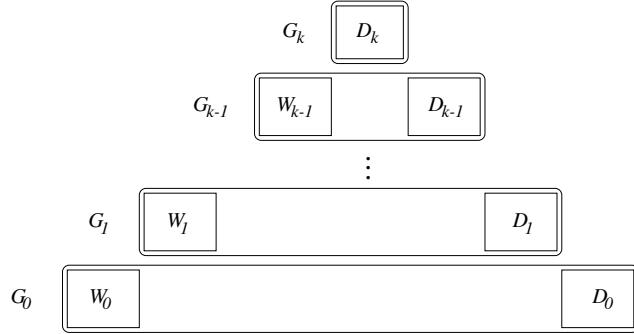
Proof: Let c be the constant such that $w(v) = c \cdot \deg(v)$, and let U be an optimal vertex cover in G . Since U covers all the edges,

$$\sum_{v \in U} \deg(v) \geq |E|.$$

Therefore, $w(U) \geq c|E|$. Now, since $\sum_{v \in V} \deg(v) = 2|E|$, $w(V) = 2c|E|$. The lemma follows. \square

Let us define the *largest degree-weighted function in w* as follows: remove all degree zero vertices from the graph, and over the remaining vertices, compute $c = \min\{w(v)/\deg(v)\}$. Then, $t(v) = c \cdot \deg(v)$ is the desired function. Define $w'(v) = w(v) - t(v)$ to be the *residual weight function*.

The algorithm for decomposing w into degree-weighted functions is as follows. Let $G_0 = G$. Remove degree zero vertices from G_0 , say this set is D_0 , and compute the largest degree-weighted function in w . Let W_0 be vertices of zero residual weight; these vertices are included in the vertex cover. Let G_1 be the graph induced on $V - (D_0 \cup W_0)$. Now, the entire process is repeated on G_1 w.r.t. the residual weight function. The process terminates when all vertices are of degree zero; let G_k denote this graph. The process is schematically shown in the following figure.



Let t_0, \dots, t_{k-1} be the degree-weighted functions defined on graphs G_0, \dots, G_{k-1} . The vertex cover chosen is $C = W_0 \cup \dots \cup W_{k-1}$. Clearly, $V - C = D_0 \cup \dots \cup D_k$.

Theorem 2.7 *The layer algorithm achieves an approximation guarantee of factor 2 for the vertex cover problem, assuming arbitrary vertex weights.*

Proof: We need to show that set C is a vertex cover for G and $w(C) \leq 2 \cdot \text{OPT}$. Assume, for contradiction, that C is not a vertex cover for G . Then

there must be an edge (u, v) with $u \in D_i$ and $v \in D_j$, for some i, j . Assume $i \leq j$. Therefore, (u, v) is present in G_i , contradicting the fact that u is a degree zero vertex.

Let C^* be an optimal vertex cover. For proving the second part, consider a vertex $v \in C$. If $v \in W_j$, its weight can be decomposed as

$$w(v) = \sum_{i \leq j} t_i(v).$$

Next, consider a vertex $v \in V - C$. If $v \in D_j$, a lower bound on its weight is given by

$$w(v) \geq \sum_{i < j} t_i(v).$$

The important observation is that in each layer i , $C^* \cap G_i$ is a vertex cover for G_i , since G_i is a vertex-induced graph. Therefore, by Lemma 2.6, $t_i(C \cap G_i) \leq 2 \cdot t_i(C^* \cap G_i)$. By the decomposition of weights given above, we get

$$w(C) = \sum_{i=0}^{k-1} t_i(C \cap G_i) \leq 2 \sum_{i=0}^{k-1} t_i(C^* \cap G_i) \leq 2 \cdot w(C^*).$$

□

Example 2.8 A tight example is provided by the family of complete bipartite graphs, $K_{n,n}$, with all vertices of unit weight. The layering algorithm will pick all $2n$ vertices of $K_{n,n}$ in the cover, whereas the optimal cover picks only one side of the bipartition. □

2.3 Application to shortest superstring

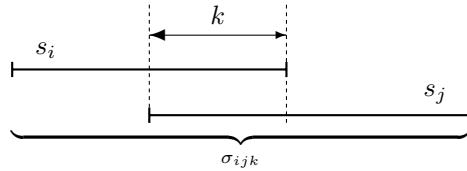
The following algorithm is given primarily to demonstrate the wide applicability of set cover. A constant factor approximation algorithm for shortest superstring will be given in Chapter 7.

Let us first provide motivation for this problem. The human DNA can be viewed as a very long string over a four-letter alphabet. Scientists are attempting to decipher this string. Since it is very long, several overlapping short segments of this string are first deciphered. Of course, the locations of these segments on the original DNA are not known. It is hypothesized that the shortest string which contains these segments as substrings is a good approximation to the original DNA string.

Problem 2.9 (Shortest superstring) Given a finite alphabet Σ , and a set of n strings, $S = \{s_1, \dots, s_n\} \subseteq \Sigma^+$, find a shortest string s that contains each s_i as a substring. Without loss of generality, we may assume that no string s_i is a substring of another string s_j , $j \neq i$.

This problem is **NP-hard**. Perhaps the first algorithm that comes to mind for finding a short superstring is the following greedy algorithm. Define the *overlap* of two strings $s, t \in \Sigma^*$ as the maximum length of a suffix of s that is also a prefix of t . The algorithm maintains a set of strings T ; initially $T = S$. At each step, the algorithm selects from T two strings that have maximum overlap and replaces them with the string obtained by overlapping them as much as possible. After $n - 1$ steps, T will contain a single string. Clearly, this string contains each s_i as a substring. This algorithm is conjectured to have an approximation factor of 2. To see that the approximation factor of this algorithm is no better than 2, consider an input consisting of 3 strings: ab^k , $b^k c$, and b^{k+1} . If the first two strings are selected in the first iteration, the greedy algorithm produces the string $ab^k cb^{k+1}$. This is almost twice as long as the shortest superstring, $ab^{k+1}c$.

We will obtain a $2H_n$ factor approximation algorithm, using the greedy set cover algorithm. The set cover instance, denoted by \mathcal{S} , is constructed as follows. For $s_i, s_j \in S$ and $k > 0$, if the last k symbols of s_i are the same as the first k symbols of s_j , let σ_{ijk} be the string obtained by overlapping these k positions of s_i and s_j :



Let M be the set that consists of the strings σ_{ijk} , for all valid choices of i, j, k . For a string $\pi \in \Sigma^+$, define $\text{set}(\pi) = \{s \in S \mid s \text{ is a substring of } \pi\}$. The universal set of the set cover instance \mathcal{S} is S , and the specified subsets of S are $\text{set}(\pi)$, for each string $\pi \in S \cup I$. The cost of $\text{set}(\pi)$ is $|\pi|$, i.e., the length of string π .

Let $\text{OPT}_{\mathcal{S}}$ and OPT denote the cost of an optimal solution to \mathcal{S} and the length of the shortest superstring of S , respectively. As shown in Lemma 2.11, $\text{OPT}_{\mathcal{S}}$ and OPT are within a factor of 2 of each other, and so an approximation algorithm for set cover can be used to obtain an approximation algorithm for shortest superstring. The complete algorithm is:

Algorithm 2.10 (Shortest superstring via set cover)

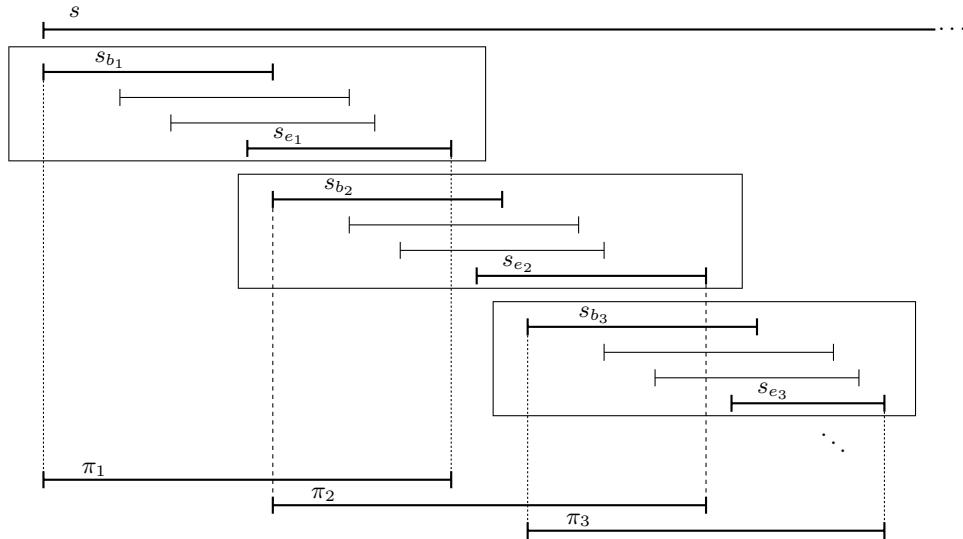
1. Use the greedy set cover algorithm to find a cover for the instance \mathcal{S} . Let $\text{set}(\pi_1), \dots, \text{set}(\pi_k)$ be the sets picked by this cover.
2. Concatenate the strings π_1, \dots, π_k , in any order.
3. Output the resulting string, say s .

Lemma 2.11 $\text{OPT} \leq \text{OPT}_{\mathcal{S}} \leq 2 \cdot \text{OPT}$.

Proof: Consider an optimal set cover, say $\{\text{set}(\pi_i) | 1 \leq i \leq l\}$, and obtain a string, say s , by concatenating the strings π_i , $1 \leq i \leq l$, in any order. Clearly, $|s| = \text{OPT}_{\mathcal{S}}$. Since each string of S is a substring of some π_i , $1 \leq i \leq l$, it is also a substring of s . Hence $\text{OPT}_{\mathcal{S}} = |s| \geq \text{OPT}$.

To prove the second inequality, let s be a shortest superstring of s_1, \dots, s_n , $|s| = \text{OPT}$. It suffices to produce *some* set cover of cost at most $2 \cdot \text{OPT}$.

Consider the leftmost occurrence of the strings s_1, \dots, s_n in string s . Since no string among s_1, \dots, s_n is a substring of another, these n leftmost occurrences start at distinct places in s . For the same reason, they also end at distinct places. Renumber the n strings in the order in which their leftmost occurrences start. Again, since no string is a substring of another, this is also the order in which they end.



We will partition the ordered list of strings s_1, \dots, s_n in groups as described below. Each group will consist of a contiguous set of strings from this

list. Let b_i and e_i denote the index of the first and last string in the i th group ($b_i = e_i$ is allowed). Thus, $b_1 = 1$. Let e_1 be the largest index of a string that overlaps with s_1 (there exists at least one such string, namely s_1 itself). In general, if $e_i < n$ we set $b_{i+1} = e_i + 1$ and denote by e_{i+1} the largest index of a string that overlaps with $s_{b_{i+1}}$. Eventually, we will get $e_t = n$ for some $t \leq n$.

For each pair of strings (s_{b_i}, s_{e_i}) , let $k_i > 0$ be the length of the overlap between their leftmost occurrences in s (this may be different from their maximum overlap). Let $\pi_i = \sigma_{b_i e_i k_i}$. Clearly, $\{\text{set}(\pi_i) | 1 \leq i \leq t\}$ is a solution for \mathcal{S} , of cost $\sum_i |\pi_i|$.

The critical observation is that π_i does not overlap π_{i+2} . We will prove this claim for $i = 1$; the same argument applies to an arbitrary i . Assume, for contradiction, that π_1 overlaps π_3 . Then the occurrence of s_{b_3} in s overlaps the occurrence of s_{e_1} . However, s_{b_3} does not overlap s_{b_2} (otherwise, s_{b_3} would have been put in the second group). This implies that s_{e_1} ends later than s_{b_2} , contradicting the property of endings of strings established earlier.

Because of this observation, each symbol of s is covered by at most two of the π_i 's. Hence $\text{OPT}_{\mathcal{S}} \leq \sum_i |\pi_i| \leq 2 \cdot \text{OPT}$. \square

The size of the universal set in the set cover instance \mathcal{S} is n , the number of strings in the given shortest superstring instance. This fact, Lemma 2.11, and Theorem 2.4 immediately give the following theorem.

Theorem 2.12 *Algorithm 2.10 is a $2H_n$ factor algorithm for the shortest superstring problem, where n is the number of strings in the given instance.*

2.4 Exercises

2.1 Given an undirected graph $G = (V, E)$, the *cardinality maximum cut problem* asks for a partition of V into sets S and \bar{S} so that the number of edges running between these sets is maximized. Consider the following greedy algorithm for this problem. Here v_1 and v_2 are arbitrary vertices in G , and for $A \subset V$, $d(v, A)$ denotes the number of edges running between vertex v and set A .

Algorithm 2.13

1. Initialization:
 $A \leftarrow \{v_1\}$
 $B \leftarrow \{v_2\}$
2. For $v \in V - \{v_1, v_2\}$ do:
if $d(v, A) \geq d(v, B)$ then $B \leftarrow B \cup \{v\}$,
else $A \leftarrow A \cup \{v\}$.
3. Output A and B .

Show that this is a factor $1/2$ approximation algorithm and give a tight example. What is the upper bound on OPT that you are using? Give examples of graphs for which this upper bound is as bad as twice OPT. Generalize the problem and the algorithm to weighted graphs.

2.2 Consider the following algorithm for the maximum cut problem, based on the technique of *local search*. Given a partition of V into sets, the basic step of the algorithm, called *flip*, is that of moving a vertex from one side of the partition to the other. The following algorithm finds a *locally optimal solution* under the flip operation, i.e., a solution which cannot be improved by a single flip.

The algorithm starts with an arbitrary partition of V . While there is a vertex such that flipping it increases the size of the cut, the algorithm flips such a vertex. (Observe that a vertex qualifies for a flip if it has more neighbors in its own partition than in the other side.) The algorithm terminates when no vertex qualifies for a flip. Show that this algorithm terminates in polynomial time, and achieves an approximation guarantee of $1/2$.

2.3 Consider the following generalization of the maximum cut problem.

Problem 2.14 (MAX k -CUT) Given an undirected graph $G = (V, E)$ with nonnegative edge costs, and an integer k , find a partition of V into sets S_1, \dots, S_k so that the total cost of edges running between these sets is maximized.

Give a greedy algorithm for this problem that achieves a factor of $(1 - \frac{1}{k})$. Is the analysis of your algorithm tight?

2.4 Give a greedy algorithm for the following problem achieving an approximation guarantee of factor $1/4$.

Problem 2.15 (Maximum directed cut) Given a directed graph $G = (V, E)$ with nonnegative edge costs, find a subset $S \subset V$ so as to maximize the total cost of edges out of S , i.e., $\text{cost}(\{(u \rightarrow v) \mid u \in S \text{ and } v \in \bar{S}\})$.

2.5 (N. Vishnoi) Use the algorithm in Exercise 2.2 and the fact that the vertex cover problem is polynomial time solvable for bipartite graphs to give a factor $\lceil \log_2 \Delta \rceil$ algorithm for vertex cover, where Δ is the degree of the vertex having highest degree.

Hint: Let H denote the subgraph consisting of edges in the maximum cut found by Algorithm 2.13. Clearly, H is bipartite, and for any vertex v , $\deg_H(v) \geq (1/2)\deg_G(v)$.

2.6 (Wigderson [257]) Consider the following problem.

Problem 2.16 (Vertex coloring) Given an undirected graph $G = (V, E)$, color its vertices with the minimum number of colors so that the two endpoints of each edge receive distinct colors.

1. Give a greedy algorithm for coloring G with $\Delta + 1$ colors, where Δ is the maximum degree of a vertex in G .
2. Give an algorithm for coloring a 3-colorable graph with $O(\sqrt{n})$ colors.

Hint: For any vertex v , the induced subgraph on its neighbors, $N(v)$, is bipartite, and hence optimally colorable. If v has degree $> \sqrt{n}$, color $v \cup N(v)$ using 3 distinct colors. Continue until every vertex has degree $\leq \sqrt{n}$. Then use the algorithm in the first part.

2.7 Let 2SC denote the restriction of set cover to instances having $f = 2$. Show that 2SC is equivalent to the vertex cover problem, with arbitrary costs, under approximation factor preserving reductions.

2.8 Prove that Algorithm 2.2 achieves an approximation factor of H_k , where k is the cardinality of the largest specified subset of U .

2.9 Give a greedy algorithm that achieves an approximation guarantee of H_n for set multicover, which is a generalization of set cover in which an integral coverage requirement is also specified for each element and sets can be picked multiple numbers of times to satisfy all coverage requirements. Assume that the cost of picking α copies of set S_i is $\alpha \cdot \text{cost}(S_i)$.

2.10 By giving an appropriate tight example, show that the analysis of Algorithm 2.2 cannot be improved even for the cardinality set cover problem, i.e., if all specified sets have unit cost.

Hint: Consider running the greedy algorithm on a vertex cover instance.

2.11 Consider the following algorithm for the weighted vertex cover problem. For each vertex v , $t(v)$ is initialized to its weight, and when $t(v)$ drops to 0, v is picked in the cover. $c(e)$ is the amount charged to edge e .

Algorithm 2.17

1. Initialization:
 $C \leftarrow \emptyset$
 $\forall v \in V, t(v) \leftarrow w(v)$
 $\forall e \in E, c(e) \leftarrow 0$
2. While C is not a vertex cover do:
 Pick an uncovered edge, say (u, v) . Let $m = \min(t(u), t(v))$.
 $t(u) \leftarrow t(u) - m$
 $t(v) \leftarrow t(v) - m$
 $c(u, v) \leftarrow m$
 Include in C all vertices having $t(v) = 0$.
3. Output C .

Show that this is a factor 2 approximation algorithm.

Hint: Show that the total amount charged to edges is a lower bound on OPT and that the weight of cover C is at most twice the total amount charged to edges.

2.12 Consider the layering algorithm for vertex cover. Another weight function for which we have a factor 2 approximation algorithm is the constant function – by simply using the factor 2 algorithm for the cardinality vertex cover problem. Can layering be made to work by using this function instead of the degree-weighted function?

2.13 Use layering to get a factor f approximation algorithm for set cover, where f is the frequency of the most frequent element. Provide a tight example for this algorithm.

2.14 A *tournament* is a directed graph $G = (V, E)$, such that for each pair of vertices, $u, v \in V$, exactly one of (u, v) and (v, u) is in E . A *feedback vertex set* for G is a subset of the vertices of G whose removal leaves an acyclic graph. Give a factor 3 algorithm for the problem of finding a minimum feedback vertex set in a directed graph.

Hint: Show that it is sufficient to “kill” all length 3 cycles. Use the factor f set cover algorithm.

2.15 (Hochbaum [125]) Consider the following problem.

Problem 2.18 (Maximum coverage) Given a universal set U of n elements, with nonnegative weights specified, a collection of subsets of U , S_1, \dots, S_l , and an integer k , pick k sets so as to maximize the weight of elements covered.

Show that the obvious algorithm, of greedily picking the best set in each iteration until k sets are picked, achieves an approximation factor of

$$1 - \left(1 - \frac{1}{k}\right)^k > 1 - \frac{1}{e}.$$

2.16 Using set cover, obtain approximation algorithms for the following variants of the shortest superstring problem (here s^R is the reverse of string s):

1. Find the shortest string that contains, for each string $s_i \in S$, both s_i and s_i^R as substrings.

Hint: The universal set for the set cover instance will contain $2n$ elements, s_i and s_i^R , for $1 \leq i \leq n$.

2. Find the shortest string that contains, for each string $s_i \in S$, either s_i or s_i^R as a substring.

Hint: Define $\text{set}(\pi) = \{s \in S \mid s \text{ or } s^R \text{ is a substring of } \pi\}$. Choose the strings π appropriately.

2.5 Notes

Algorithm 2.2 is due to Johnson [150], Lovász [192], and Chvátal [48]. The hardness result for set cover, showing that this algorithm is essentially the best possible, is due to Feige [80], improving on the result of Lund and Yannakakis [199]. The application to shortest superstring is due to Li [187].

3 Steiner Tree and TSP

In this chapter, we will present constant factor algorithms for two fundamental problems, metric Steiner tree and metric TSP. The reasons for considering the metric case of these problems are quite different. For Steiner tree, this is the core of the problem – the rest of the problem reduces to this case. For TSP, without this restriction, the problem admits no approximation factor, assuming $\mathbf{P} \neq \mathbf{NP}$. The algorithms, and their analyses, are similar in spirit, which is the reason for presenting these problems together.

3.1 Metric Steiner tree

The Steiner tree problem was defined by Gauss in a letter he wrote to Schumacher (reproduced on the cover of this book). Today, this problem occupies a central place in the field of approximation algorithms. The problem has a wide range of applications, all the way from finding minimum length interconnection of terminals in VLSI design to constructing phylogeny trees in computational biology. This problem and its generalizations will be studied extensively in this book, see Chapters 22 and 23.

Problem 3.1 (Steiner tree) Given an undirected graph $G = (V, E)$ with nonnegative edge costs and whose vertices are partitioned into two sets, *required* and *Steiner*, find a minimum cost tree in G that contains all the required vertices and any subset of the Steiner vertices.

We will first show that the core of this problem lies in its restriction to instances in which the edge costs satisfy the *triangle inequality*, i.e., G is a complete undirected graph, and for any three vertices u , v , and w , $\text{cost}(u, v) \leq \text{cost}(u, w) + \text{cost}(v, w)$. Let us call this restriction the *metric Steiner tree problem*.

Theorem 3.2 *There is an approximation factor preserving reduction from the Steiner tree problem to the metric Steiner tree problem.*

Proof: We will transform, in polynomial time, an instance I of the Steiner tree problem, consisting of graph $G = (V, E)$, to an instance I' of the metric Steiner tree problem as follows. Let G' be the complete undirected graph on

vertex set V . Define the cost of edge (u, v) in G' to be the cost of a shortest $u-v$ path in G . G' is called the *metric closure* of G . The partition of V into required and Steiner vertices in I' is the same as in I .

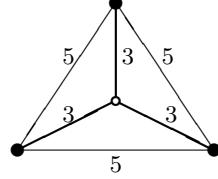
For any edge $(u, v) \in E$, its cost in G' is no more than its cost in G . Therefore, the cost of an optimal solution in I' does not exceed the cost of an optimal solution in I .

Next, given a Steiner tree T' in I' , we will show how to obtain, in polynomial time, a Steiner tree T in I of at most the same cost. The cost of an edge (u, v) in G' corresponds to the cost of a path in G . Replace each edge of T' by the corresponding path to obtain a subgraph of G . Clearly, in this subgraph, all the required vertices are connected. However, this subgraph may, in general, contain cycles. If so, remove edges to obtain tree T . This completes the approximation factor preserving reduction. \square

As a consequence of Theorem 3.2, any approximation factor established for the metric Steiner tree problem carries over to the entire Steiner tree problem.

3.1.1 MST-based algorithm

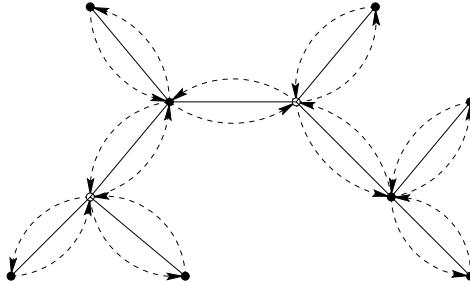
Let R denote the set of required vertices. Clearly, a minimum spanning tree (MST) on R is a feasible solution for this problem. Since the problem of finding an MST is in **P** and the metric Steiner tree problem is **NP-hard**, we cannot expect the MST on R to always give an optimal Steiner tree; below is an example in which the MST is strictly costlier.



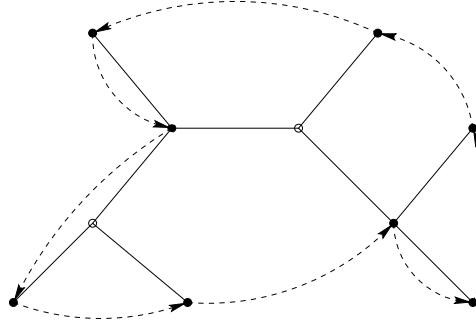
Even so, an MST on R is not much more costly than an optimal Steiner tree:

Theorem 3.3 *The cost of an MST on R is within $2 \cdot \text{OPT}$.*

Proof: Consider a Steiner tree of cost OPT . By doubling its edges we obtain an Eulerian graph connecting all vertices of R and, possibly, some Steiner vertices. Find an Euler tour of this graph, for example by traversing the edges in DFS (depth first search) order:



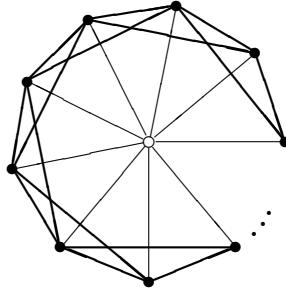
The cost of this Euler tour is $2 \cdot \text{OPT}$. Next obtain a Hamiltonian cycle on the vertices of R by traversing the Euler tour and “short-cutting” Steiner vertices and previously visited vertices of R :



Because of triangle inequality, the shortcuts do not increase the cost of the tour. If we delete one edge of this Hamiltonian cycle, we obtain a path that spans R and has cost at most $2 \cdot \text{OPT}$. This path is also a spanning tree on R . Hence, the MST on R has cost at most $2 \cdot \text{OPT}$. \square

Theorem 3.3 gives a straightforward factor 2 algorithm for the metric Steiner tree problem: simply find an MST on the set of required vertices. As in the case of set cover, the “correct” way of viewing this algorithm is in the setting of LP-duality theory. In Chapters 22 and 23 we will see that LP-duality provides the lower bound on which this algorithm is based and also helps solve generalizations of this problem.

Example 3.4 For a tight example, consider a graph with n required vertices and one Steiner vertex. An edge between the Steiner vertex and a required vertex has cost 1, and an edge between two required vertices has cost 2 (not all edges of cost 2 are shown below). In this graph, any MST on R has cost $2(n - 1)$, while $\text{OPT} = n$.



□

3.2 Metric TSP

The following is a well-studied problem in combinatorial optimization.

Problem 3.5 (Traveling salesman problem (TSP)) Given a complete graph with nonnegative edge costs, find a minimum cost cycle visiting every vertex exactly once.

In its full generality, TSP cannot be approximated, assuming $\mathbf{P} \neq \mathbf{NP}$.

Theorem 3.6 *For any polynomial time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $\mathbf{P} = \mathbf{NP}$.*

Proof: Assume, for a contradiction, that there is a factor $\alpha(n)$ polynomial time approximation algorithm, \mathcal{A} , for the general TSP problem. We will show that \mathcal{A} can be used for deciding the Hamiltonian cycle problem (which is **NP-hard**) in polynomial time, thus implying $\mathbf{P} = \mathbf{NP}$.

The central idea is a reduction from the Hamiltonian cycle problem to TSP, that transforms a graph G on n vertices to an edge-weighted complete graph G' on n vertices such that

- if G has a Hamiltonian cycle, then the cost of an optimal TSP tour in G' is n , and
- if G does not have a Hamiltonian cycle, then an optimal TSP tour in G' is of cost $> \alpha(n) \cdot n$.

Observe that when run on graph G' , algorithm \mathcal{A} must return a solution of cost $\leq \alpha(n) \cdot n$ in the first case, and a solution of cost $> \alpha(n) \cdot n$ in the second case. Thus, it can be used for deciding whether G contains a Hamiltonian cycle.

The reduction is simple. Assign a weight of 1 to edges of G , and a weight of $\alpha(n) \cdot n$ to nonedges, to obtain G' . Now, if G has a Hamiltonian cycle, then the corresponding tour in G' has cost n . On the other hand, if G has

no Hamiltonian cycle, any tour in G' must use an edge of cost $\alpha(n) \cdot n$, and therefore has cost $> \alpha(n) \cdot n$. \square

Notice that in order to obtain such a strong nonapproximability result, we had to assign edge costs that violate triangle inequality. If we restrict ourselves to graphs in which edge costs satisfy triangle inequality, i.e., consider *metric TSP*, the problem remains **NP**-complete, but it is no longer hard to approximate.

3.2.1 A simple factor 2 algorithm

We will first present a simple factor 2 algorithm. The lower bound we will use for obtaining this factor is the cost of an MST in G . This is a lower bound because deleting any edge from an optimal solution to TSP gives us a spanning tree of G .

Algorithm 3.7 (Metric TSP – factor 2)

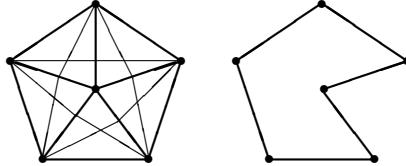
1. Find an MST, T , of G .
2. Double every edge of the MST to obtain an Eulerian graph.
3. Find an Eulerian tour, \mathcal{T} , on this graph.
4. Output the tour that visits vertices of G in the order of their first appearance in \mathcal{T} . Let \mathcal{C} be this tour.

Notice that Step 4 is similar to the “short-cutting” step in Theorem 3.3.

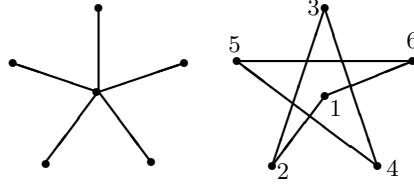
Theorem 3.8 *Algorithm 3.7 is a factor 2 approximation algorithm for metric TSP.*

Proof: As noted above, $\text{cost}(T) \leq \text{OPT}$. Since \mathcal{T} contains each edge of T twice, $\text{cost}(\mathcal{T}) = 2 \cdot \text{cost}(T)$. Because of triangle inequality, after the “short-cutting” step, $\text{cost}(\mathcal{C}) \leq \text{cost}(\mathcal{T})$. Combining these inequalities we get that $\text{cost}(\mathcal{C}) \leq 2 \cdot \text{OPT}$. \square

Example 3.9 A tight example for this algorithm is given by a complete graph on n vertices with edges of cost 1 and 2. We present the graph for $n = 6$ below, where thick edges have cost 1 and remaining edges have cost 2. For arbitrary n the graph has $2n - 2$ edges of cost 1, with these edges forming the union of a star and an $n - 1$ cycle; all remaining edges have cost 2. The optimal TSP tour has cost n , as shown below for $n = 6$:



Suppose that the MST found by the algorithm is the spanning star created by edges of cost 1. Moreover, suppose that the Euler tour constructed in Step 3 visits vertices in order shown below for $n = 6$:



Then the tour obtained after short-cutting contains $n - 2$ edges of cost 2 and has a total cost of $2n - 2$. Asymptotically, this is twice the cost of the optimal TSP tour. \square

3.2.2 Improving the factor to 3/2

Algorithm 3.7 first finds a low cost Euler tour spanning the vertices of G , and then short-cuts this tour to find a traveling salesman tour. Is there a cheaper Euler tour than that found by doubling an MST? Recall that a graph has an Euler tour iff all its vertices have even degrees. Thus, we only need to be concerned about the vertices of odd degree in the MST. Let V' denote this set of vertices. $|V'|$ must be even since the sum of degrees of all vertices in the MST is even. Now, if we add to the MST a minimum cost perfect matching on V' , every vertex will have an even degree, and we get an Eulerian graph. With this modification, the algorithm achieves an approximation guarantee of $3/2$.

Algorithm 3.10 (Metric TSP – factor 3/2)

1. Find an MST of G , say T .
2. Compute a minimum cost perfect matching, M , on the set of odd-degree vertices of T . Add M to T and obtain an Eulerian graph.
3. Find an Euler tour, \mathcal{T} , of this graph.
4. Output the tour that visits vertices of G in order of their first appearance in \mathcal{T} . Let \mathcal{C} be this tour.

Interestingly, the proof of this algorithm is based on a second lower bound on OPT.

Lemma 3.11 *Let $V' \subseteq V$, such that $|V'|$ is even, and let M be a minimum cost perfect matching on V' . Then, $\text{cost}(M) \leq \text{OPT}/2$.*

Proof: Consider an optimal TSP tour of G , say τ . Let τ' be the tour on V' obtained by short-cutting τ . By the triangle inequality, $\text{cost}(\tau') \leq$

$\text{cost}(\tau)$. Now, τ' is the union of two perfect matchings on V' , each consisting of alternate edges of τ . Thus, the cheaper of these matchings has cost $\leq \text{cost}(\tau')/2 \leq \text{OPT}/2$. Hence the optimal matching also has cost at most $\text{OPT}/2$. \square

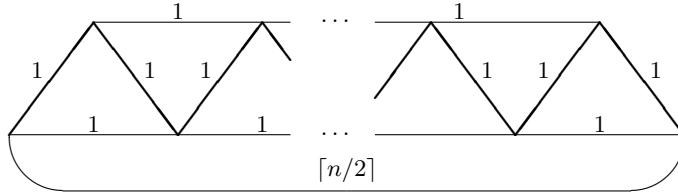
Theorem 3.12 *Algorithm 3.10 achieves an approximation guarantee of $3/2$ for metric TSP.*

Proof: The cost of the Euler tour,

$$\text{cost}(\mathcal{T}) \leq \text{cost}(T) + \text{cost}(M) \leq \text{OPT} + \frac{1}{2}\text{OPT} = \frac{3}{2}\text{OPT},$$

where the first inequality follows by using the two lower bounds on OPT . Using the triangle inequality, $\text{cost}(\mathcal{C}) \leq \text{cost}(\mathcal{T})$, and the theorem follows. \square

Example 3.13 A tight example for this algorithm is given by the following graph on n vertices, with n odd:



Thick edges represent the MST found in step 1. This MST has only two odd vertices, and by adding the edge joining them we obtain a traveling salesman tour of cost $(n - 1) + \lceil n/2 \rceil$. In contrast, the optimal tour has cost n . \square

Finding a better approximation algorithm for metric TSP is currently one of the outstanding open problems in this area. Many researchers have conjectured that an approximation factor of $4/3$ may be achievable.

3.3 Exercises

3.1 The hardness of the Steiner tree problem lies in determining the optimal subset of Steiner vertices that need to be included in the tree. Show this by proving that if this set is provided, then the optimal Steiner tree can be computed in polynomial time.

Hint: Find an MST on the union of this set and the set of required vertices.

3.2 Let $G = (V, E)$ be a graph with nonnegative edge costs. S , the *senders* and R , the *receivers*, are disjoint subsets of V . The problem is to find a minimum cost subgraph of G that has a path connecting each receiver to a

sender (any sender suffices). Partition the instances into two cases: $S \cup R = V$ and $S \cup R \neq V$. Show that these two cases are in **P** and **NP-hard**, respectively. For the second case, give a factor 2 approximation algorithm.

Hint: Add a new vertex which is connected to each sender by a zero cost edge. Consider the new vertex and all receivers as required and the remaining vertices as Steiner, and find a minimum cost Steiner tree.

3.3 Give an approximation factor preserving reduction from the set cover problem to the following problem, thereby showing that it is unlikely to have a better approximation guarantee than $O(\log n)$.

Problem 3.14 (Directed Steiner tree) $G = (V, E)$ is a directed graph with nonnegative edge costs. The vertex set V is partitioned into two sets, *required* and *Steiner*. One of the required vertices, r , is special. The problem is to find a minimum cost tree in G rooted into r that contains all the required vertices and any subset of the Steiner vertices.

Hint: Construct a three layer graph: layer 1 contains a required vertex corresponding to each element, layer 2 contains a Steiner vertex corresponding to each set, and layer 3 contains r .

3.4 (Hoogeveen [130]) Consider variants on the metric TSP problem in which the object is to find a simple path containing all the vertices of the graph. Three different problems arise, depending on the number (0, 1, or 2) of endpoints of the path that are specified. Obtain the following approximation algorithms.

- If zero or one endpoints are specified, obtain a $3/2$ factor algorithm.
- If both endpoints are specified, obtain a $5/3$ factor algorithm.

Hint: Use the idea behind Algorithm 3.10.

3.5 (Papadimitriou and Yannakakis [219]) Let G be a complete undirected graph in which all edge lengths are either 1 or 2 (clearly, G satisfies the triangle inequality). Give a $4/3$ factor algorithm for TSP in this special class of graphs.

Hint: Start by finding a minimum 2-matching in G . A 2-matching is a subset S of edges so that every vertex has exactly 2 edges of S incident at it.

3.6 (Frieze, Galbiati, and Maffioli [89]) Give an $O(\log n)$ factor approximation algorithm for the following problem.

Problem 3.15 (Asymmetric TSP) We are given a directed graph G on vertex set V , with a nonnegative cost specified for edge $(u \rightarrow v)$, for each pair $u, v \in V$. The edge costs satisfy *the directed triangle inequality*, i.e., for any three vertices u , v , and w , $\text{cost}(u \rightarrow v) \leq \text{cost}(u \rightarrow w) + \text{cost}(w \rightarrow v)$. The problem is to find a minimum cost cycle visiting every vertex exactly once.

Hint: Use the fact that a minimum cost cycle cover (i.e., disjoint cycles covering all the vertices) can be found in polynomial time. Shrink the cycles and recurse.

3.7 Let $G = (V, E)$ be a graph with edge costs satisfying the triangle inequality, and $V' \subseteq V$ be a set of even cardinality. Prove or disprove: The cost of a minimum cost perfect matching on V' is bounded above by the cost of a minimum cost perfect matching on V .

3.8 Given n points in \mathbf{R}^2 , define the optimal Euclidean Steiner tree to be a minimum length tree containing all n points and any other subset of points from \mathbf{R}^2 . Prove that each of the additional points must have degree three, with all three angles being 120° .

3.9 (Rao, Sadayappan, Hwang, and Shor [230]) This exercise develops a factor 2 approximation algorithm for the following problem.

Problem 3.16 (Rectilinear Steiner arborescence) Let p_1, \dots, p_n be points given in \mathbf{R}^2 in the positive quadrant. A path from the origin to point p_i is said to be *monotone* if it consists of segments traversing in the positive x direction or the positive y direction (informally, going right or up). The problem is to find a minimum length tree containing monotone paths from the origin to each of the n points; such a tree is called *rectilinear Steiner arborescence*.

For point p , define x_p and y_p to be its x and y coordinates, and $|p|_1 = |x_p| + |y_p|$. Say that point p *dominates* point q if $x_p \leq x_q$ and $y_p \leq y_q$. For sets of points A and B , we will say that A *dominates* B if for each point $b \in B$, there is a point $a \in A$ such that a dominates b . For points p and q , define $\text{dom}(p, q) = (x, y)$, where $x = \min(x_p, x_q)$ and $y = \min(y_p, y_q)$. If p dominates q , define $\text{segments}(p, q)$ to be a monotone path from p to q . Consider the following algorithm.

Algorithm 3.17 (Rectilinear Steiner arborescence)

1. $T \leftarrow \emptyset$.
2. $P \leftarrow \{p_1, \dots, p_n\} \cup \{(0, 0)\}$.
3. while $|P| > 1$ do:
 - Pick $p, q = \arg \max_{p, q \in P} (|\text{dom}(p, q)|_1)$.
 - $P \leftarrow (P - \{p, q\}) \cup \{\text{dom}(p, q)\}$.
 - $T \leftarrow T \cup \text{segments}(\text{dom}(p, q), p) \cup \text{segments}(\text{dom}(p, q), q)$.
4. Output T .

For $z \geq 0$, define ℓ_z to be the line $x + y = z$. For a rectilinear Steiner arborescence T , let $T(z) = |T \cap \ell_z|$. Prove that the length of T is

$$\int_{z=0}^{\infty} T(z) dz.$$

Also, for every $x \geq 0$ define $P_z = \{p \in P \text{ s.t. } |p|_1 > z\}$, and

$$N(z) = \min\{|C| : C \subset \ell_z \text{ and } C \text{ dominates } P_z\}.$$

Prove that

$$\int_{z=0}^{\infty} N(z) dz$$

is a lower bound on OPT.

Use these facts to show that Algorithm 3.17 achieves an approximation guarantee of 2.

3.10 (I. Măndoiu) This exercise develops a factor 9 approximation algorithm for the following problem, which finds applications in VLSI clock routing.

Problem 3.18 (Rectilinear zero-skew tree) Given a set S of points in the rectilinear plane, find a minimum length *zero-skew tree* (ZST) for S , i.e., a rooted tree T embedded in the rectilinear plane such that points in S are leaves of T and all root-to-leaf paths in T have equal length. By *length* of a path we mean the sum of the lengths of edges on it.

- Let T be an arbitrary zero-skew tree, and let R' denote the common length of all root-to-leaf paths. For $r \geq 0$, let $T(r)$ denote the number of points of T that are at a length of $R' - r$ from the root. Prove that the length of T is

$$\int_0^{R'} T(r) dr$$

- A closed ℓ_1 ball of radius r centered at point p is the set of all points whose ℓ_1 -distance from p is $\leq r$. Let R denote the radius of the smallest ℓ_1 -ball that contains all points of S . For $r \geq 0$, let $N(r)$ denote the minimum number of closed ℓ_1 -balls of radius r needed to cover all points of S . Prove that

$$\int_0^R N(r) dr$$

is a lower bound on the length of the optimum ZST.

3. Consider the following algorithm. First, compute R and find a radius R ℓ_1 -ball enclosing all points of S . The center of this ball is chosen as the root of the resulting ZST. This ball can be partitioned into 4 balls, called its *quadrants*, of radius $R/2$ each. The root can be connected to the center of any of these balls by an edge of length $R/2$. These balls can be further partitioned into 4 balls each of radius $R/4$, and so on.

The ZST is constructed recursively, starting with the ball of radius R . The center of the current ball is connected to the centers of each of its quadrants that has a point of S . The algorithm then recurses on each of these quadrants. If the current ball contains exactly one point of S , then this ball is not partitioned into quadrants. Let r' be the radius of this ball, c its center, and $p \in S$ the point in it. Clearly, the ℓ_1 distance between c and p is $\leq r'$. Connect c to p by a rectilinear path of length exactly r' .

Show that for $0 \leq r \leq R$, $T(r) \leq 9N(r)$. Hence, show that this is a factor 9 approximation algorithm.

3.4 Notes

The Steiner tree problem has its origins in a problem posed by Fermat, and was defined by Gauss in a letter he wrote to his student Schumacher on March 21, 1836. Parts of the letter are reproduced on the cover of this book. Courant and Robbins [55] popularized this problem under the name of Steiner, a well known 19th century geometer. See Hwang, Richards, and Winter [133] and Schreiber [236] for the fascinating history of this problem.

The factor 2 Steiner tree algorithm was discovered independently by Choukhmane [44], Iwainsky, Canuto, Taraszow, and Villa [136], Kou, Markowsky, and Berman [177], and Plesník [221]. The factor $3/2$ metric TSP algorithm is due to Christofides [45], and Theorem 3.6 is due to Sahni and Gonzalez [232]. The lower bound in Exercise 3.10 is from Charikar, Kleinberg, Kumar, Rajagopalan, Sahai, and Tomkins [41]. The best factor known for the rectilinear zero-skew tree problem, due to Zelikovsky and Măndoiu [263], is 3.

Given n points on the Euclidean plane, the minimum spanning tree on these points is within a factor of $2/\sqrt{3}$ of the minimum Steiner tree (which is allowed to use any set of points on the plane as Steiner points). This was shown by Du and Hwang [63], thereby settling the conjecture of Gilbert and Pollak [100].

4 Multiway Cut and k -Cut

The theory of cuts occupies a central place in the study of exact algorithms. In this chapter, we will present approximation algorithms for natural generalizations of the minimum cut problem. These generalizations are **NP-hard**.

Given a connected, undirected graph $G = (V, E)$ with an assignment of weights to edges, $w : E \rightarrow \mathbf{R}^+$, a *cut* is defined by a partition of V into two sets, say V' and $V - V'$, and consists of all edges that have one endpoint in each partition. Clearly, the removal of the cut from G disconnects G . Given *terminals* $s, t \in V$, consider a partition of V that separates s and t . The cut defined by such a partition will be called an *s–t cut*. The problems of finding a minimum weight cut and a minimum weight *s–t* cut can be efficiently solved using a maximum flow algorithm. Let us generalize these two notions:

Problem 4.1 (Multiway cut) Given a set of terminals $S = \{s_1, s_2, \dots, s_k\} \subseteq V$, a *multiway cut* is a set of edges whose removal disconnects the terminals from each other. The multiway cut problem asks for the minimum weight such set.

Problem 4.2 (Minimum k -cut) A set of edges whose removal leaves k connected components is called a *k -cut*. The k -cut problem asks for a minimum weight k -cut.

The problem of finding a minimum weight multiway cut is **NP-hard** for any fixed $k \geq 3$. Observe that the case $k = 2$ is precisely the minimum *s–t* cut problem. The minimum k -cut problem is polynomial time solvable for fixed k ; however, it is **NP-hard** if k is specified as part of the input. In this chapter, we will obtain factor $2 - 2/k$ approximation algorithms for both problems. In Chapter 19 we will improve the guarantee for the multiway cut problem to $3/2$.

4.1 The multiway cut problem

Define an *isolating cut* for s_i to be a set of edges whose removal disconnects s_i from the rest of the terminals.

Algorithm 4.3 (Multiway cut)

1. For each $i = 1, \dots, k$, compute a minimum weight isolating cut for s_i , say C_i .
2. Discard the heaviest of these cuts, and output the union of the rest, say C .

Each computation in Step 1 can be accomplished by identifying the terminals in $S - \{s_i\}$ into a single node, and finding a minimum cut separating this node from s_i ; this takes one max-flow computation. Clearly, removing C from the graph disconnects every pair of terminals, and so is a multiway cut.

Theorem 4.4 *Algorithm 4.3 achieves an approximation guarantee of $2 - 2/k$.*

Proof: Let A be an optimal multiway cut in G . We can view A as the union of k cuts as follows: The removal of A from G will create k connected components, each having one terminal (since A is a minimum weight multiway cut, no more than k components will be created). Let A_i be the cut separating the component containing s_i from the rest of the graph. Then $A = \bigcup_{i=1}^k A_i$.

Since each edge of A is incident at two of these components, each edge will be in two of the cuts A_i . Hence,

$$\sum_{i=1}^k w(A_i) = 2w(A).$$

Clearly, A_i is an isolating cut for s_i . Since C_i is a minimum weight isolating cut for s_i , $w(C_i) \leq w(A_i)$. Notice that this already gives a factor 2 algorithm, by taking the union of all k cuts C_i . Finally, since C is obtained by discarding the heaviest of the cuts C_i ,

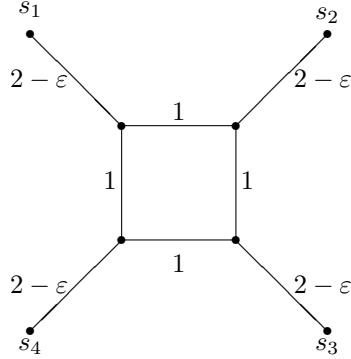
$$w(C) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^k w(C_i) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^k w(A_i) = 2 \left(1 - \frac{1}{k}\right) w(A).$$

□

Once again, Algorithm 4.3 is not based on a lower bounding scheme. Exercise 19.2 gives an algorithm with the same guarantee using an LP-relaxation as the lower bound. The use of LP-relaxations is fruitful for this problem as well. Section 19.1 gives an algorithm with an improved guarantee, using another LP-relaxation.

Example 4.5 A tight example for this algorithm is given by a graph on $2k$ vertices consisting of a k -cycle and a distinct terminal attached to each vertex of the cycle. The edges of the cycle have weight 1 and edges attaching terminals to the cycle have weight $2 - \varepsilon$ for a small fraction $\varepsilon > 0$.

For example, the graph corresponding to $k = 4$ is:



For each terminal s_i , the minimum weight isolating cuts for s_i is given by the edge incident to s_i . So, the cut C returned by the algorithm has weight $(k - 1)(2 - \varepsilon)$. On the other hand, the optimal multiway cut is given by the cycle edges, and has weight k . \square

4.2 The minimum k -cut problem

A natural algorithm for finding a k -cut is as follows. Starting with G , compute a minimum cut in each connected component and remove the lightest one; repeat until there are k connected components. This algorithm does achieve a guarantee of $2 - 2/k$, however, the proof is quite involved. Instead we will use the *Gomory–Hu tree representation of minimum cuts* to give a simpler algorithm achieving the same guarantee.

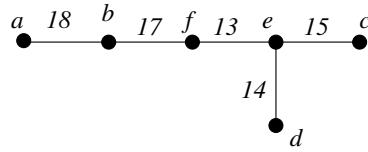
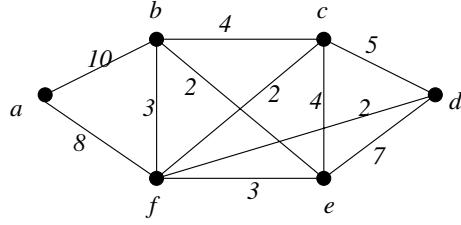
Minimum cuts, as well as sub-optimal cuts, in undirected graphs have several interesting structural properties, as opposed to cuts in directed graphs (the algorithm of Section 28.2 is based on exploiting some of these properties). The existence of Gomory–Hu trees is one of the remarkable consequences of these properties.

Let T be a tree on vertex set V ; the edges of T need not be in E . Let e be an edge in T . Its removal from T creates two connected components. Let S and \bar{S} be the vertex sets of these components. The cut defined in graph G by the partition (S, \bar{S}) is the *cut associated with e in G* . Define a weight function w' on the edges of T . Tree T will be said to be a Gomory–Hu tree for G if

1. for each pair of vertices $u, v \in V$, the weight of a minimum u – v cut in G is the same as that in T .
2. for each edge $e \in T$, $w'(e)$ is the weight of the cut associated with e in G , and

A Gomory–Hu tree encodes, in a succinct manner, a minimum u – v cut in G , for each pair of vertices $u, v \in V$ as follows. A minimum u – v cut in T is given by a minimum weight edge on the unique path from u to v in T , say e . By the properties stated above, the cut associated with e in G is a minimum u – v cut, and has weight $w'(e)$. So, for the $\binom{n}{2}$ pairs of vertices $u, v \in V$, we need only $n - 1$ cuts, those encoded by the edges of a Gomory–Hu tree, to give minimum u – v cuts in G .

The following figure shows a weighted graph and its associated Gomory–Hu tree. Exercise 4.6 shows how to construct a Gomory–Hu tree for an undirected graph, using only $n - 1$ max-flow computations.



We will need the following lemma.

Lemma 4.6 *Let S be the union of cuts in G associated with l edges of T . Then, the removal of S from G leaves a graph with at least $l + 1$ components.*

Proof: Removing the corresponding l edges from T leaves exactly $l + 1$ connected components, say with vertex sets V_1, V_2, \dots, V_{l+1} . Clearly, removing S from G will disconnect each pair V_i and V_j . Hence we must get at least $l + 1$ connected components. \square

As a consequence of Lemma 4.6, the union of $k - 1$ cuts picked from T will form a k -cut in G . The complete algorithm is given below.

Algorithm 4.7 (Minimum k -cut)

1. Compute a Gomory–Hu tree T for G .
2. Output the union of the lightest $k - 1$ cuts of the $n - 1$ cuts associated with edges of T in G ; let C be this union.

By Lemma 4.6, the removal of C from G will leave at least k components. If more than k components are created, throw back some of the removed edges until there are exactly k components.

Theorem 4.8 *Algorithm 4.7 achieves an approximation factor of $2 - 2/k$.*

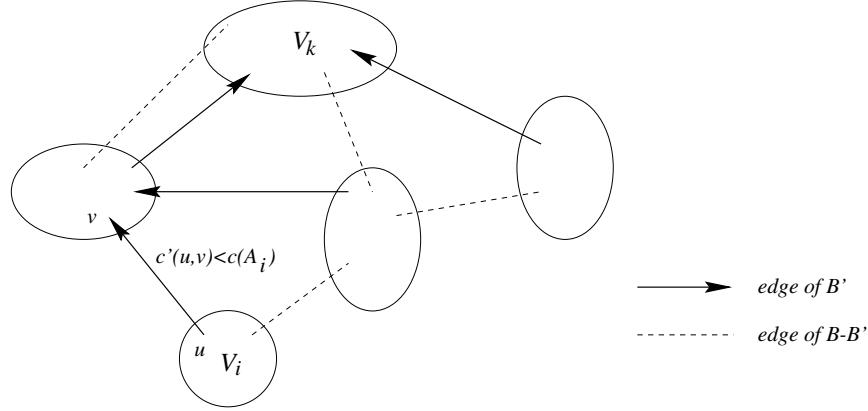
Proof: Let A be an optimal k -cut in G . As in Theorem 4.4, we can view A as the union of k cuts: Let V_1, V_2, \dots, V_k be the k components formed by removing A from G , and let A_i denote the cut separating V_i from the rest of the graph. Then $A = A_1 \cup \dots \cup A_k$, and, since each edge of A lies in two of these cuts,

$$\sum_{i=1}^k w(A_i) = 2w(A).$$

Without loss of generality assume that A_k is the heaviest of these cuts. The idea behind the rest of the proof is to show that there are $k-1$ cuts defined by the edges of T whose weights are dominated by the weight of the cuts A_1, A_2, \dots, A_{k-1} . Since the algorithm picks the lightest $k-1$ cuts defined by T , the theorem follows.

The $k-1$ cuts are identified as follows. Let B be the set of edges of T that connect across two of the sets V_1, V_2, \dots, V_k . Consider the graph on vertex set V and edge set B , and shrink each of the sets V_1, V_2, \dots, V_k to a single vertex. This shrunk graph must be connected (since T was connected). Throw edges away until a tree remains. Let $B' \subseteq B$ be the left over edges, $|B'| = k-1$. The edges of B' define the required $k-1$ cuts.

Next, root this tree at V_k (recall that A_k was assumed to be the heaviest cut among the cuts A_i). This helps in defining a correspondence between the edges in B' and the sets V_1, V_2, \dots, V_{k-1} : each edge corresponds to the set it comes out of in the rooted tree.



Suppose edge $(u, v) \in B'$ corresponds to set V_i in this manner. The weight of a minimum $u-v$ cut in G is $w'(u, v)$. Since A_i is a $u-v$ cut in G ,

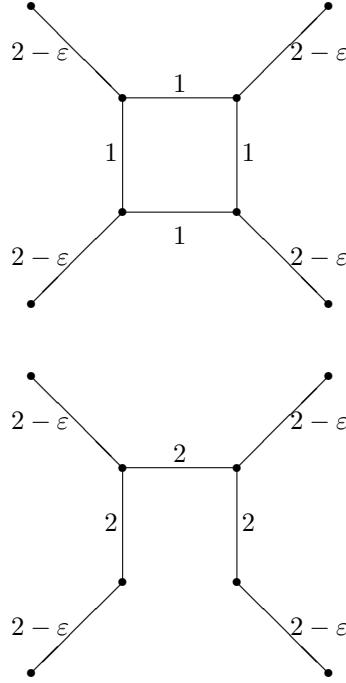
$$w(A_i) \geq w'(u, v).$$

Thus each cut among A_1, A_2, \dots, A_{k-1} is at least as heavy as the cut defined in G by the corresponding edge of B' . This, together with the fact that C is the union of the lightest $k-1$ cuts defined by T , gives:

$$w(C) \leq \sum_{e \in B'} w'(e) \leq \sum_{i=1}^{k-1} w(A_i) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^k w(A_i) = 2 \left(1 - \frac{1}{k}\right) w(A).$$

□

Example 4.9 The tight example given above for multiway cuts on $2k$ vertices also serves as a tight example for the k -cut algorithm (of course, there is no need to mark vertices as terminals). Below we give the example for $k=4$, together with its Gomory–Hu tree.



The lightest $k-1$ cuts in the Gomory–Hu tree have weight $2 - \varepsilon$ each, corresponding to picking edges of weight $2 - \varepsilon$ of G . So, the k -cut returned

by the algorithm has weight $(k - 1)(2 - \varepsilon)$. On the other hand, the optimal k -cut picks all edges of weight 1, and has weight k . \square

4.3 Exercises

4.1 Show that Algorithm 4.3 can be used as a subroutine for finding a k -cut within a factor of $2 - 2/k$ of the minimum k -cut. How many subroutine calls are needed?

4.2 A natural greedy algorithm for computing a multiway cut is the following. Starting with G , compute minimum s_i - s_j cuts for all pairs s_i, s_j that are still connected and remove the lightest of these cuts; repeat this until all pairs s_i, s_j are disconnected. Prove that this algorithm also achieves a guarantee of $2 - 2/k$.

The next 4 exercises provide background and an algorithm for finding Gomory–Hu trees.

4.3 Let $G = (V, E)$ be a graph and $w : E \rightarrow \mathbf{R}^+$ be an assignment of nonnegative weights to its edges. For $u, v \in V$ let $f(u, v)$ denote the weight of a minimum u - v cut in G .

1. Let $u, v, w \in V$, and suppose $f(u, v) \leq f(u, w) \leq f(v, w)$. Show that $f(u, v) = f(u, w)$, i.e., the two smaller numbers are equal.
2. Show that among the $\binom{n}{2}$ values $f(u, v)$, for all pairs $u, v \in V$, there are at most $n - 1$ distinct values.
3. Show that for $u, v, w \in V$,

$$f(u, v) \geq \min\{f(u, w), f(w, v)\}.$$

4. Show that for $u, v, w_1, \dots, w_r \in V$

$$f(u, w) \geq \min\{f(u, w_1), f(w_1, w_2), \dots, f(w_r, v)\} \quad (4.1)$$

4.4 Let T be a tree on vertex set V with weight function w' on its edges. We will say that T is a *flow equivalent tree* if it satisfies the first of the two Gomory–Hu conditions. i.e., for each pair of vertices $u, v \in V$, the weight of a minimum u - v cut in G is the same as that in T . Let K be the complete graph on V . Define the weight of each edge (u, v) in K to be $f(u, v)$. Show that any maximum weight spanning tree in K is a flow equivalent tree for G .

Hint: For $u, v \in V$, let u, w_1, \dots, w_r, v be the unique path from u to v in T . Use (4.1) and the fact that since T is a maximum weight spanning tree, $f(u, v) \leq \min\{f(u, w_1), \dots, f(w_r, v)\}$.

4.5 Let (A, \bar{A}) be a minimum $s-t$ cut such that $s \in A$. Let x and y be any two vertices in A . Consider the graph G' obtained by collapsing all vertices of \bar{A} to a single vertex $v_{\bar{A}}$. The weight of any edge $(a, v_{\bar{A}})$ in G' is defined to be the sum of the weights of edges (a, b) where $b \in \bar{A}$. Clearly, any cut in G' defines a cut in G . Show that a minimum $x-y$ cut in G' defines a minimum $x-y$ cut in G .

4.6 Now we are ready to state the Gomory–Hu algorithm. The algorithm maintains a partition of V , (S_1, S_2, \dots, S_t) , and a spanning tree T on the vertex set $\{S_1, \dots, S_t\}$. Let w' be the function assigning weights to the edges of T . Tree T satisfies the following invariant.

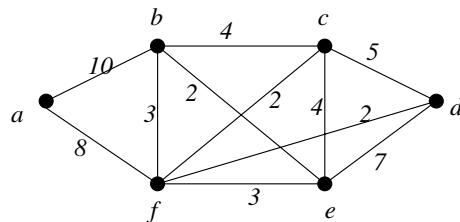
Invariant: For any edge (S_i, S_j) in T there are vertices a and b in S_i and S_j respectively, such that $w'(S_i, S_j) = f(a, b)$, and the cut defined by edge (S_i, S_j) is a minimum $a-b$ cut in G .

The algorithm starts with the trivial partition V , and proceeds in $n - 1$ iterations. In each iteration, it selects a set S_i in the partition such that $|S_i| \geq 2$ and refines the partition by splitting S_i , and finding a tree on the refined partition satisfying the invariant. This is accomplished as follows. Let x and y be two distinct vertices in S_i . Root the current tree T at S_i , and consider the subtrees rooted at the children of S_i . Each of these subtrees is collapsed into a single vertex, to obtain graph G' (besides these collapsed vertices, G' contains all vertices of S_i). A minimum $x-y$ cut is found in G' . Let (A, B) be the partition of the vertices of G' defining this cut, with $x \in A$ and $y \in B$, and let w_{xy} be the weight of this cut. Compute $S_i^x = S_i \cap A$ and $S_i^y = S_i \cap B$, the two sets into which S_i splits.

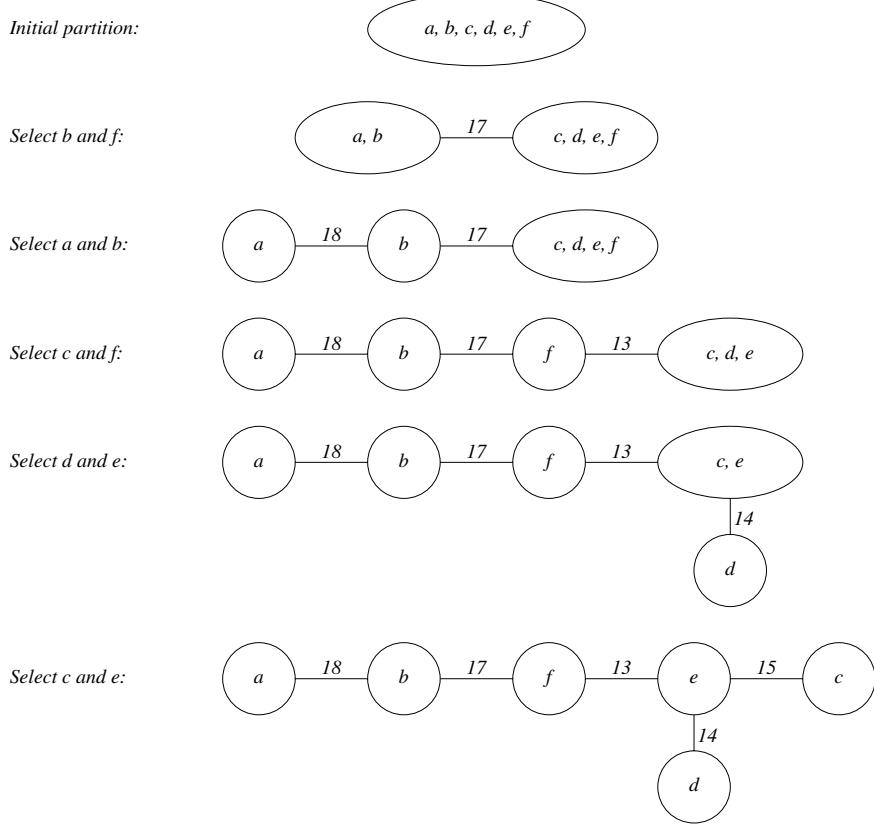
The algorithm updates the partition and the tree as follows. It refines the partition by replacing S_i with two sets S_i^x and S_i^y . The new tree has the edge (S_i^x, S_i^y) , with weight w_{xy} . Consider a subtree T' that was incident at S_i in T . Assume w.l.o.g. that the node corresponding to T' lies in A . Then, T' is connected by an edge to S_i^x . The weight of this connecting edge is the same as the weight of the edge connecting T' to S_i . All edges in T' retain their weights.

Show that the new tree satisfies the invariant. Hence show that the algorithm terminates (when the partition consists of singleton vertices) with a Gomory–Hu tree for G .

Consider the graph:



The execution of the Gomory–Hu algorithm is demonstrated below:



4.7 Prove that if the Gomory–Hu tree for an edge-weighted undirected graph G contains all $n - 1$ distinct weights, then G can have only one minimum weight cut.

4.4 Notes

Algorithm 4.3 is due to Dahlhaus, Johnson, Seymour, Papadimitriou and Yannakakis [57]. Algorithm 4.7 is due to Saran and Vazirani [233]; the proof given here is due to R. Ravi. For Gomory–Hu trees see Gomory and Hu [110].

5 k -Center

Consider the following application. Given a set of cities, with intercity distances specified, pick k cities for locating warehouses in so as to minimize the maximum distance of a city from its closest warehouse. We will study this problem, called the k -center problem, and its weighted version, under the restriction that the edge costs satisfy the triangle inequality. Without this restriction, the k -center problem cannot be approximated within factor $\alpha(n)$, for any computable function $\alpha(n)$, assuming $\mathbf{P} \neq \mathbf{NP}$ (see Exercise 5.1).

We will introduce the algorithmic technique of *parametric pruning* for solving this problem. In Chapter 17 we will use this technique in a linear programming setting.

Problem 5.1 (Metric k -center) Let $G = (V, E)$ be a complete undirected graph with edge costs satisfying the triangle inequality, and k be a positive integer. For any set $S \subseteq V$ and vertex $v \in V$, define $\text{connect}(v, S)$ to be the cost of the cheapest edge from v to a vertex in S . The problem is to find a set $S \subseteq V$, with $|S| = k$, so as to minimize $\max_v \{\text{connect}(v, S)\}$.

5.1 Parametric pruning applied to metric k -center

If we know the cost of an optimal solution, we may be able to prune away irrelevant parts of the input and thereby simplify the search for a good solution. However, as stated in Chapter 1, computing the cost of an optimal solution is precisely the difficult core of **NP-hard** **NP**-optimization problems. The technique of parametric pruning gets around this difficulty as follows. A parameter t is chosen, which can be viewed as a “guess” on the cost of an optimal solution. For each value of t , the given instance I is pruned by removing parts that will not be used in any solution of cost $> t$. Denote the pruned instance by $I(t)$. The algorithm consists of two steps. In the first step, the family of instances $I(t)$ is used for computing a lower bound on OPT , say t^* . In the second step, a solution is found in instance $I(\alpha \cdot t^*)$, for a suitable choice of α .

A restatement of the k -center problem shows how parametric pruning applies naturally to it. Sort the edges of G in nondecreasing order of cost, i.e., $\text{cost}(e_1) \leq \text{cost}(e_2) \leq \dots \leq \text{cost}(e_m)$, and let $G_i = (V, E_i)$, where $E_i =$

$\{e_1, e_2, \dots, e_i\}$. A *dominating set* in an undirected graph $H = (U, F)$ is a subset $S \subseteq U$ such that every vertex in $U - S$ is adjacent to a vertex in S . Let $\text{dom}(H)$ denote the size of a minimum cardinality dominating set in H . Computing $\text{dom}(H)$ is **NP-hard**. The k -center problem is equivalent to finding the smallest index i such that G_i has a dominating set of size at most k , i.e., G_i contains k stars spanning all vertices, where a *star* is the graph $K_{1,p}$, with $p \geq 1$. If i^* is the smallest such index, then $\text{cost}(e_{i^*})$ is the cost of an optimal k -center. We will denote this by OPT . We will work with the family of graphs G_1, \dots, G_m .

Define the *square of graph* H to be the graph containing an edge (u, v) whenever H has a path of length at most two between u and v , $u \neq v$. We will denote it by H^2 . The following structural result gives a method for lower bounding OPT .

Lemma 5.2 *Given a graph H , let I be an independent set in H^2 . Then, $|I| \leq \text{dom}(H)$.*

Proof: Let D be a minimum dominating set in H . Then, H contains $|D|$ stars spanning all vertices. Since each of these stars will be a clique in H^2 , H^2 contains $|D|$ cliques spanning all vertices. Clearly, I can pick at most one vertex from each clique, and the lemma follows. \square

The k -center algorithm is:

Algorithm 5.3 (Metric k -center)

1. Construct $G_1^2, G_2^2, \dots, G_m^2$.
2. Compute a maximal independent set, M_i , in each graph G_i^2 .
3. Find the smallest index i such that $|M_i| \leq k$, say j .
4. Return M_j .

The lower bound on which this algorithm is based is:

Lemma 5.4 *For j as defined in the algorithm, $\text{cost}(e_j) \leq \text{OPT}$.*

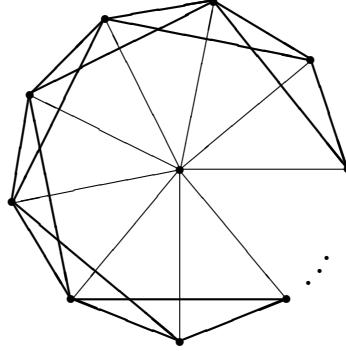
Proof: For every $i < j$ we have that $|M_i| > k$. Now, by Lemma 5.2, $\text{dom}(G_i) > k$, and so $i^* > i$. Hence, $j \leq i^*$. \square

Theorem 5.5 *Algorithm 5.3 achieves an approximation factor of 2 for the metric k -center problem.*

Proof: The key observation is that a maximal independent set, I , in a graph is also a dominating set (for, if some vertex v is not dominated by I , then $I \cup \{v\}$ must also be an independent set, contradicting I 's maximality). Thus,

there exist stars in G_j^2 , centered on the vertices of M_j , covering all vertices. By the triangle inequality, each edge used in constructing these stars has cost at most $2 \cdot \text{cost}(e_j)$. The theorem follows from Lemma 5.4. \square

Example 5.6 A tight example for the previous algorithm is given by a wheel graph on $n + 1$ vertices, where all edges incident to the center vertex have cost 1, and the rest of the edges have cost 2:



(Here, thin edges have cost 1 and thick edges have cost 2; not all edges of cost 2 are shown.)

For $k = 1$, the optimal solution is the center of the wheel, and $\text{OPT} = 1$. The algorithm will compute index $j = n$. Now, G_n^2 is a clique and, if a peripheral vertex is chosen as the maximal independent set, then the cost of the solution found is 2. \square

Next, we will show that 2 is essentially the best approximation factor achievable for the metric k -center problem.

Theorem 5.7 *Assuming $\mathbf{P} \neq \mathbf{NP}$, there is no polynomial time algorithm achieving a factor of $2 - \varepsilon$, $\varepsilon > 0$, for the metric k -center problem.*

Proof: We will show that such an algorithm can solve the dominating set problem in polynomial time. The idea is similar to that of Theorem 3.6 and involves giving a reduction from the dominating set problem to metric k -center. Let $G = (V, E)$, k be an instance of the dominating set problem. Construct a complete graph $G' = (V, E')$ with edge costs given by

$$\text{cost}(u, v) = \begin{cases} 1, & \text{if } (u, v) \in E, \\ 2, & \text{if } (u, v) \notin E. \end{cases}$$

Clearly, G' satisfies the triangle inequality. This reduction satisfies the conditions:

- if $\text{dom}(G) \leq k$, then G' has a k -center of cost 1, and

- if $\text{dom}(G) > k$, then the optimum cost of a k -center in G' is 2.

In the first case, when run on G' , the $(2 - \varepsilon)$ -approximation algorithm must give a solution of cost 1, since it cannot use an edge of cost 2. Hence, using this algorithm, we can distinguish between the two possibilities, thus solving the dominating set problem. \square

5.2 The weighted version

We will use the technique of parametric pruning to obtain a factor 3 approximation algorithm for the following generalization of the metric k -center problem.

Problem 5.8 (Metric weighted k -center) In addition to a cost function on edges, we are given a weight function on vertices, $w : V \rightarrow R^+$, and a bound $W \in R^+$. The problem is to pick $S \subseteq V$ of total weight at most W , minimizing the same objective function as before, i.e.,

$$\max_{v \in V} \left\{ \min_{u \in S} \{ \text{cost}(u, v) \} \right\}.$$

Let $\text{wdom}(G)$ denote the weight of a minimum weight dominating set in G . Then, with respect to the graphs G_i defined above, we need to find the smallest index i such that $\text{wdom}(G_i) \leq W$. If i^* is this index, then the cost of the optimal solution is $\text{OPT} = \text{cost}(e_{i^*})$.

Given a vertex weighted graph H , let I be an independent set in H^2 . For each $u \in I$, let $s(u)$ denote a lightest neighbor of u in H , where u is also considered a neighbor of itself. (Notice that the neighbor is picked in H and not in H^2 .) Let $S = \{s(u) \mid u \in I\}$. The following fact, analogous to Lemma 5.2, will be used to derive a lower bound on OPT :

Lemma 5.9 $w(S) \leq \text{wdom}(H)$.

Proof: Let D be a minimum weight dominating set of H . Then there exists a set of disjoint stars in H , centered on the vertices of D and covering all the vertices. Since each of these stars becomes a clique in H^2 , I can pick at most one vertex from each of them. Thus, each vertex in I has the center of the corresponding star available as a neighbor in H . Hence, $w(S) \leq w(D)$. \square

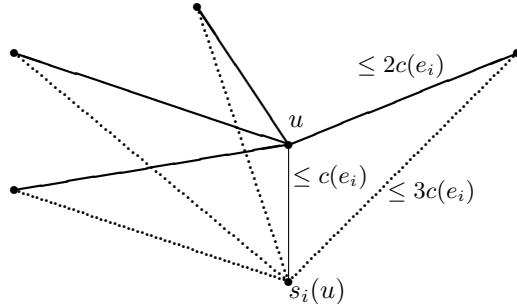
The algorithm is given below. In it, $s_i(u)$ will denote a lightest neighbor of u in G_i ; for this definition, u will also be considered a neighbor of itself.

Algorithm 5.10 (Metric weighted k -center)

1. Construct $G_1^2, G_2^2, \dots, G_m^2$.
2. Compute a maximal independent set, M_i , in each graph G_i^2 .
3. Compute $S_i = \{s_i(u) \mid u \in M_i\}$.
4. Find the minimum index i such that $w(S_i) \leq W$, say j .
5. Return S_j .

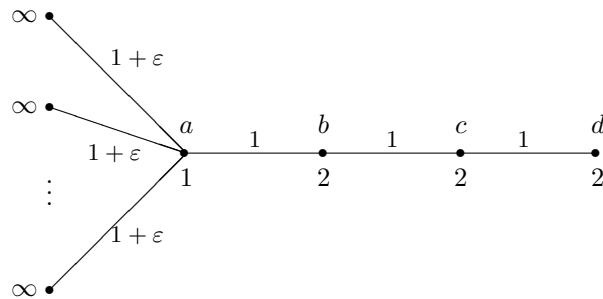
Theorem 5.11 Algorithm 5.10 achieves an approximation factor of 3 for the weighted k -center problem.

Proof: By Lemma 5.9, $\text{cost}(e_j)$ is a lower bound on OPT; the argument is identical to that in Lemma 5.4 and is omitted here. Since M_j is a dominating set in G_j^2 , we can cover V with stars of G_j^2 centered in vertices of M_j . By the triangle inequality these stars use edges of cost at most $2 \cdot \text{cost}(e_j)$.



Each star center is adjacent to a vertex in S_j , using an edge of cost at most $\text{cost}(e_j)$. Move each of the centers to the adjacent vertex in S_j and redefine the stars. Again, by the triangle inequality, the largest edge cost used in constructing the final stars is at most $3 \cdot \text{cost}(e_j)$. \square

Example 5.12 A tight example is provided by the following graph on $n+4$ vertices. Vertex weights and edge costs are as marked; all missing edges have a cost given by the shortest path.



It is not difficult to see that for $W = 3$ the optimum cost of a k -center is $1 + \varepsilon$: a k -center achieving this cost is $\{a, c\}$. For any $i < n + 3$, the set S_i computed by the algorithm will contain a vertex of infinite weight. Suppose that, for $i = n + 3$, the algorithm chooses $M_{n+3} = \{b\}$ as a maximal independent set. Then $S_{n+3} = \{a\}$, and this is the output of the algorithm. The cost of this solution is 3. \square

5.3 Exercises

5.1 Show that if the edge costs do not satisfy the triangle inequality, then the k -center problem cannot be approximated within factor $\alpha(n)$ for any computable function $\alpha(n)$.

Hint: Put together ideas from Theorems 3.6 and 5.7.

5.2 Consider Step 2 of Algorithm 5.3, in which a maximal independent set is found in G_i^2 . Perhaps a more natural choice would have been to find a minimal dominating set. Modify Algorithm 5.3 so that M_i is picked to be a minimal dominating set in G_i^2 . Show that this modified algorithm does not achieve an approximation guarantee of 2 for the k -center problem. What approximation factor can you establish for this algorithm?

Hint: With this modification, the lower bounding method does not work, since Lemma 5.2 does not hold if I is picked to be a minimal dominating set in H^2 .

5.3 (Gonzalez [111]) Consider the following problem.

Problem 5.13 (Metric k -cluster) Let $G = (V, E)$ be a complete undirected graph with edge costs satisfying the triangle inequality, and let k be a positive integer. The problem is to partition V into sets V_1, \dots, V_k so as to minimize the costliest edge between two vertices in the same set, i.e., minimize

$$\max_{1 \leq i \leq k, u, v \in V_i} \text{cost}(u, v).$$

1. Give a factor 2 approximation algorithm for this problem, together with a tight example.
2. Show that this problem cannot be approximated within a factor of $2 - \varepsilon$, for any $\varepsilon > 0$, unless $\mathbf{P} = \mathbf{NP}$.

5.4 (Khuller, Pless, and Sussmann [169]) The *fault-tolerant* version of the metric k -center problem has an additional input, $\alpha \leq k$, which specifies the

number of centers that each city should be connected to. The problem again is to pick k centers so that the length of the longest edge used is minimized.

A set $S \subseteq V$ in an undirected graph $H = (V, E)$ is an α -dominating set if each vertex $v \in V$ is adjacent to at least α vertices in S (assuming that a vertex is adjacent to itself). Let $\text{dom}_\alpha(H)$ denote the size of a minimum cardinality α -dominating set in H .

1. Let I be an independent set in H^2 . Show that $\alpha|I| \leq \text{dom}_\alpha(H)$.
2. Give a factor 3 approximation algorithm for the fault-tolerant k -center problem.

Hint: Compute a maximal independent set M_i in G_i^2 , for $1 \leq i \leq m$. Find the smallest index i such that $|M_i| \leq \lfloor \frac{k}{\alpha} \rfloor$, and moreover, the degree of each vertex of M_i in G_i is $\geq \alpha - 1$.

5.5 (Khuller, Pless, and Sussmann [169]) Consider a modification of the problem of Exercise 5.4 in which vertices of S have no connectivity requirements and only vertices of $V - S$ have connectivity requirements. Each vertex of $V - S$ needs to be connected to α vertices in S . The object again is to pick S , $|S| = k$, so that the length of the longest edge used is minimized.

The algorithm for this problem works on each graph G_i . It starts with $S_i = \emptyset$. Vertex $v \in V - S_i$ is said to be j -connected if it is adjacent to j vertices in S_i , using edges of G_i^2 . While there is a vertex $v \in V - S_i$ that is not k -connected, pick the vertex with minimum connectivity, and include it in S_i . Finally, find the minimum index i such that $|S_i| \leq k$, say l . Output S_l . Prove that this is a factor 2 approximation algorithm.

5.4 Notes

Both k -center algorithms presented in this chapter are due to Hochbaum and Shmoys [127], and Theorem 5.7 is due to Hsu and Nemhauser [132].

Chapter 1

Introduction



A squirrel, a platypus and a hamster walk into a bar...

Imagine that you are an exceptionally tech-savvy security guard of a bar in an undisclosed small town on the west coast of Norway. Every Friday, half of the inhabitants of the town go out, and the bar you work at is well known for its nightly brawls. This of course results in an excessive amount of work for you; having to throw out intoxicated guests is tedious and rather unpleasant labor. Thus you decide to take preemptive measures. As the town is small, you know everyone in it, and you also know who will be likely to fight with whom if they are admitted to the bar. So you wish to plan ahead, and only admit people if they will not be fighting with anyone else at the bar. At the same time, the management wants to maximize profit and is not too happy if you on any given night reject more than k people at the door. Thus, you are left with the following optimization problem. You have a list of all of the n people who will come to the bar, and for each pair of people a prediction of whether or not they will fight if they both are admitted. You need to figure out whether it is possible to admit everyone except for at most k troublemakers, such that no fight breaks out among the admitted guests. Let us call this problem the BAR FIGHT PREVENTION problem. Figure 1.1 shows an instance of the problem and a solution for $k = 3$. One can easily check that this instance has no solution with $k = 2$.

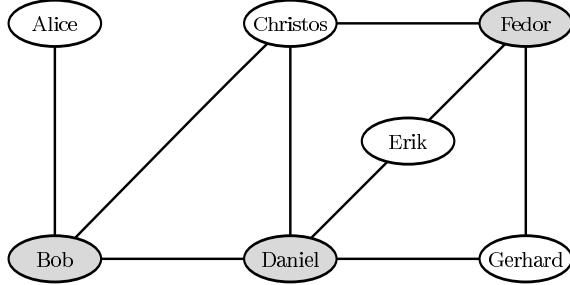


Fig. 1.1: An instance of the BAR FIGHT PREVENTION problem with a solution for $k = 3$. An edge between two guests means that they will fight if both are admitted

Efficient algorithms for BAR FIGHT PREVENTION

Unfortunately, BAR FIGHT PREVENTION is a classic NP-complete problem (the reader might have heard of it under the name VERTEX COVER), and so the best way to solve the problem is by trying all possibilities, right? If there are $n = 1000$ people planning to come to the bar, then you can quickly code up the brute-force solution that tries each of the $2^{1000} \approx 1.07 \cdot 10^{301}$ possibilities. Sadly, this program won't terminate before the guests arrive, probably not even before the universe implodes on itself. Luckily, the number k of guests that should be rejected is not that large, $k \leq 10$. So now the program only needs to try $\binom{1000}{10} \approx 2.63 \cdot 10^{23}$ possibilities. This is much better, but still quite infeasible to do in one day, even with access to supercomputers.

So should you give up at this point, and resign yourself to throwing guests out after the fights break out? Well, at least you can easily identify some peaceful souls to accept, and some troublemakers you need to refuse at the door for sure. Anyone who does not have a potential conflict with anyone else can be safely moved to the list of people to accept. On the other hand, if some guy will fight with at least $k + 1$ other guests you have to reject him — as otherwise you will have to reject all of his $k + 1$ opponents, thereby upsetting the management. If you identify such a troublemaker (in the example of Fig. 1.1, Daniel is such a troublemaker), you immediately strike him from the guest list, and decrease the number k of people you can reject by one.¹

If there is no one left to strike out in this manner, then we know that each guest will fight with at most k other guests. Thus, rejecting any single guest will resolve at most k potential conflicts. And so, if there are more than k^2

¹ The astute reader may observe that in Fig. 1.1, after eliminating Daniel and setting $k = 2$, Fedor still has three opponents, making it possible to eliminate him and set $k = 1$. Then Bob, who is in conflict with Alice and Christos, can be eliminated, resolving all conflicts.

potential conflicts, you know that there is no way to ensure a peaceful night at the bar by rejecting only k guests at the door. As each guest who has not yet been moved to the accept or reject list participates in at least one and at most k potential conflicts, and there are at most k^2 potential conflicts, there are at most $2k^2$ guests whose fate is yet undecided. Trying all possibilities for these will need approximately $\binom{2k^2}{k} \leq \binom{200}{10} \approx 2.24 \cdot 10^{16}$ checks, which is feasible to do in less than a day on a modern supercomputer, but quite hopeless on a laptop.

If it is safe to admit anyone who does not participate in any potential conflict, what about those who participate in exactly one? If Alice has a conflict with Bob, but with no one else, then it is always a good idea to admit Alice. Indeed, you cannot accept both Alice and Bob, and admitting Alice cannot be any worse than admitting Bob: if Bob is in the bar, then Alice has to be rejected for sure and potentially some other guests as well. Therefore, it is safe to accept Alice, reject Bob, and decrease k by one in this case. This way, you can always decide the fate of any guest with only one potential conflict. At this point, each guest you have not yet moved to the accept or reject list participates in at least two and at most k potential conflicts. It is easy to see that with this assumption, having at most k^2 unresolved conflicts implies that there are only at most k^2 guests whose fate is yet undecided, instead of the previous upper bound of $2k^2$. Trying all possibilities for which of those to refuse at the door requires $\binom{k^2}{k} \leq \binom{100}{10} \approx 1.73 \cdot 10^{13}$ checks. With a clever implementation, this takes less than half a day on a laptop, so if you start the program in the morning you'll know who to refuse at the door by the time the bar opens. Therefore, instead of using brute force to go through an enormous search space, we used simple observations to reduce the search space to a manageable size. This algorithmic technique, using reduction rules to decrease the size of the instance, is called *kernelization*, and will be the subject of Chapter 2 (with some more advanced examples appearing in Chapter 9).

It turns out that a simple observation yields an even faster algorithm for BAR FIGHT PREVENTION. The crucial point is that every conflict has to be resolved, and that the only way to resolve a conflict is to refuse at least one of the two participants. Thus, as long as there is at least one unresolved conflict, say between Alice and Bob, we proceed as follows. Try moving Alice to the reject list and run the algorithm recursively to check whether the remaining conflicts can be resolved by rejecting at most $k - 1$ guests. If this succeeds you already have a solution. If it fails, then move Alice back onto the undecided list, move Bob to the reject list and run the algorithm recursively to check whether the remaining conflicts can be resolved by rejecting at most $k - 1$ additional guests (see Fig. 1.2). If this recursive call also fails to find a solution, then you can be sure that there is no way to avoid a fight by rejecting at most k guests.

What is the running time of this algorithm? All it does is to check whether all conflicts have been resolved, and if not, it makes two recursive calls. In

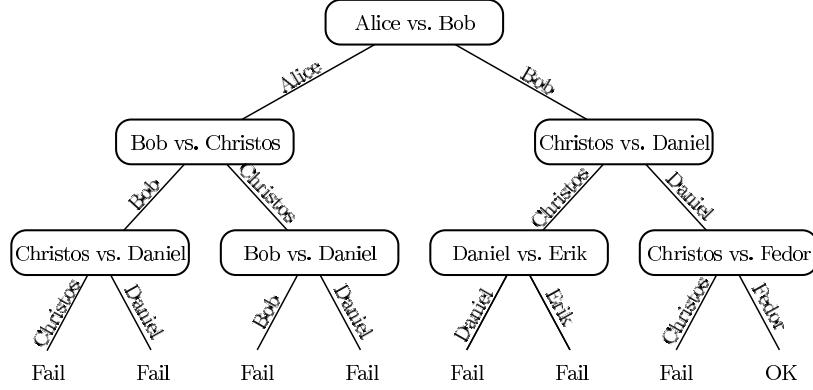


Fig. 1.2: The search tree for BAR FIGHT PREVENTION with $k = 3$. In the leaves marked with “Fail”, the parameter k is decreased to zero, but there are still unresolved conflicts. The rightmost branch of the search tree finds a solution: after rejecting Bob, Daniel, and Fedor, no more conflicts remain

both of the recursive calls the value of k decreases by 1, and when k reaches 0 all the algorithm has to do is to check whether there are any unresolved conflicts left. Hence there is a total of 2^k recursive calls, and it is easy to implement each recursive call to run in linear time $\mathcal{O}(n + m)$, where m is the total number of possible conflicts. Let us recall that we already achieved the situation where every undecided guest has at most k conflicts with other guests, so $m \leq nk/2$. Hence the total number of operations is approximately $2^k \cdot n \cdot k \leq 2^{10} \cdot 10,000 = 10,240,000$, which takes a fraction of a second on today’s laptops. Or cell phones, for that matter. You can now make the BAR FIGHT PREVENTION app, and celebrate with a root beer. This simple algorithm is an example of another algorithmic paradigm: the technique of *bounded search trees*. In Chapter 3 we will see several applications of this technique to various problems.

The algorithm above runs in time $\mathcal{O}(2^k \cdot k \cdot n)$, while the naive algorithm that tries every possible subset of k people to reject runs in time $\mathcal{O}(n^k)$. Observe that if k is considered to be a constant (say $k = 10$), then both algorithms run in polynomial time. However, as we have seen, there is a quite dramatic difference between the running times of the two algorithms. The reason is that even though the naive algorithm is a polynomial-time algorithm for every fixed value of k , the exponent of the polynomial depends on k . On the other hand, the final algorithm we designed runs in linear time for every fixed value of k ! This difference is what parameterized algorithms and complexity is all about. In the $\mathcal{O}(2^k \cdot k \cdot n)$ -time algorithm, the combinatorial explosion is restricted to the parameter k : the running time is exponential

in k , but depends only polynomially (actually, linearly) on n . Our goal is to find algorithms of this form.

Algorithms with running time $f(k) \cdot n^c$, for a constant c independent of both n and k , are called *fixed-parameter algorithms*, or FPT algorithms. Typically the goal in parameterized algorithmics is to design FPT algorithms, trying to make both the $f(k)$ factor and the constant c in the bound on the running time as small as possible. FPT algorithms can be put in contrast with less efficient XP algorithms (for *slice-wise polynomial*), where the running time is of the form $f(k) \cdot n^{g(k)}$, for some functions f, g . There is a tremendous difference in the running times $f(k) \cdot n^{g(k)}$ and $f(k) \cdot n^c$.

In parameterized algorithmics, k is simply a *relevant secondary measurement* that encapsulates some aspect of the input instance, be it the size of the solution sought after, or a number describing how “structured” the input instance is.

A negative example: vertex coloring

Not every choice for what k measures leads to FPT algorithms. Let us have a look at an example where it does not. Suppose the management of the hypothetical bar you work at doesn’t want to refuse anyone at the door, but still doesn’t want any fights. To achieve this, they buy $k - 1$ more bars across the street, and come up with the following brilliant plan. Every night they will compile a list of the guests coming, and a list of potential conflicts. Then you are to split the guest list into k groups, such that no two guests with a potential conflict between them end up in the same group. Then each of the groups can be sent to one bar, keeping everyone happy. For example, in Fig. 1.1, we may put Alice and Christos in the first bar, Bob, Erik, and Gerhard in the second bar, and Daniel and Fedor in the third bar.

We model this problem as a graph problem, representing each person as a vertex, and each conflict as an edge between two vertices. A partition of the guest list into k groups can be represented by a function that assigns to each vertex an integer between 1 and k . The objective is to find such a function that, for every edge, assigns different numbers to its two endpoints. A function that satisfies these constraints is called a *proper k -coloring* of the graph. Not every graph has a proper k -coloring. For example, if there are $k + 1$ vertices with an edge between every pair of them, then each of these vertices needs to be assigned a unique integer. Hence such a graph does not have a proper k -coloring. This gives rise to a computational problem, called VERTEX COLORING. Here we are given as input a graph G and an integer k , and we need to decide whether G has a proper k -coloring.

It is well known that VERTEX COLORING is NP-complete, so we do not hope for a polynomial-time algorithm that works in all cases. However, it is fair to assume that the management does not want to own more than $k = 5$ bars on the same street, so we will gladly settle for a $\mathcal{O}(2^k \cdot n^c)$ -time algorithm for some constant c , mimicking the success we had with our first problem. Unfortunately, deciding whether a graph G has a proper 5-coloring is NP-complete, so any $f(k) \cdot n^c$ -time algorithm for VERTEX COLORING for any function f and constant c would imply that P = NP; indeed, suppose such an algorithm existed. Then, given a graph G , we can decide whether G has a proper 5-coloring in time $f(5) \cdot n^c = \mathcal{O}(n^c)$. But then we have a polynomial-time algorithm for an NP-hard problem, implying P = NP. Observe that even an XP algorithm with running time $f(k) \cdot n^{g(k)}$ for any functions f and g would imply that P = NP by an identical argument.

A hard parameterized problem: finding cliques

The example of VERTEX COLORING illustrates that parameterized algorithms are not all-powerful: there are parameterized problems that do not seem to admit FPT algorithms. But very importantly, in this specific example, we could explain very precisely why we are not able to design efficient algorithms, even when the number of bars is small. From the perspective of an algorithm designer such insight is very useful; she can now stop wasting time trying to design efficient algorithms based only on the fact that the number of bars is small, and start searching for other ways to attack the problem instances. If we are trying to make a polynomial-time algorithm for a problem and failing, it is quite likely that this is because the problem is NP-hard. Is the theory of NP-hardness the right tool also for giving negative evidence for fixed-parameter tractability? In particular, if we are trying to make an $f(k) \cdot n^c$ -time algorithm and fail to do so, is it because the problem is NP-hard for some fixed constant value of k , say $k = 100$? Let us look at another example problem.

Now that you have a program that helps you decide who to refuse at the door and who to admit, you are faced with a different problem. The people in the town you live in have friends who might get upset if their friend is refused at the door. You are quite skilled at martial arts, and you can handle at most $k - 1$ angry guys coming at you, but probably not k . What you are most worried about are groups of at least k people where everyone in the group is friends with everyone else. These groups tend to have an “all for one and one for all” mentality — if one of them gets mad at you, they all do. Small as the town is, you know exactly who is friends with whom, and you want to figure out whether there is a group of at least k people where everyone is friends with everyone else. You model this as a graph problem where every person is a vertex and two vertices are connected by an edge if the corresponding persons are friends. What you are looking for is a *clique* on k vertices, that

is, a set of k vertices with an edge between every pair of them. This problem is known as the CLIQUE problem. For example, if we interpret now the edges of Fig. 1.1 as showing friendships between people, then Bob, Christos, and Daniel form a clique of size 3.

There is a simple $\mathcal{O}(n^k)$ -time algorithm to check whether a clique on at least k vertices exists; for each of the $\binom{n}{k} = \mathcal{O}\left(\frac{n^k}{k^2}\right)$ subsets of vertices of size k , we check in time $\mathcal{O}(k^2)$ whether every pair of vertices in the subset is adjacent. Unfortunately, this XP algorithm is quite hopeless to run for $n = 1000$ and $k = 10$. Can we design an FPT algorithm for this problem? So far, no one has managed to find one. Could it be that this is because finding a k -clique is NP-hard for some fixed value of k ? Suppose the problem was NP-hard for $k = 100$. We just gave an algorithm for finding a clique of size 100 in time $\mathcal{O}(n^{100})$, which is polynomial time. We would then have a polynomial-time algorithm for an NP-hard problem, implying that P = NP. So we cannot expect to be able to use NP-hardness in this way in order to rule out an FPT algorithm for CLIQUE. More generally, it seems very difficult to use NP-hardness in order to explain why a problem that does have an XP algorithm does not admit an FPT algorithm.

Since NP-hardness is insufficient to differentiate between problems with $f(k) \cdot n^{g(k)}$ -time algorithms and problems with $f(k) \cdot n^c$ -time algorithms, we resort to stronger complexity theoretical assumptions. The theory of W[1]-hardness (see Chapter 13) allows us to prove (under certain complexity assumptions) that even though a problem is polynomial-time solvable for every fixed k , the parameter k has to appear in the exponent of n in the running time, that is, the problem is not FPT. This theory has been quite successful for identifying which parameterized problems are FPT and which are unlikely to be. Besides this qualitative classification of FPT versus W[1]-hard, more recent developments give us also (an often surprisingly tight) quantitative understanding of the time needed to solve a parameterized problem. Under reasonable assumptions about the hardness of CNF-SAT (see Chapter 14), it is possible to show that there is no $f(k) \cdot n^c$, or even a $f(k) \cdot n^{o(k)}$ -time algorithm for finding a clique on k vertices. Thus, up to constant factors in the exponent, the naive $\mathcal{O}(n^k)$ -time algorithm is optimal! Over the past few years, it has become a rule, rather than an exception, that whenever we are unable to significantly improve the running time of a parameterized algorithm, we are able to show that the existing algorithms are asymptotically optimal, under reasonable assumptions. For example, under the same assumptions that we used to rule out an $f(k) \cdot n^{o(k)}$ -time algorithm for solving CLIQUE, we can also rule out a $2^{o(k)} \cdot n^{\mathcal{O}(1)}$ -time algorithm for the BAR FIGHT PREVENTION problem from the beginning of this chapter.

Any algorithmic theory is incomplete without an accompanying complexity theory that establishes intractability of certain problems. There

Problem	Good news	Bad news
BAR FIGHT PREVENTION	$\mathcal{O}(2^k \cdot k \cdot n)$ -time algorithm	NP-hard (probably not in P)
CLIQUE with Δ	$\mathcal{O}(2^\Delta \cdot \Delta^2 \cdot n)$ -time algorithm	NP-hard (probably not in P)
CLIQUE with k	$n^{\mathcal{O}(k)}$ -time algorithm	W[1]-hard (probably not FPT)
VERTEX COLORING		NP-hard for $k = 3$ (probably not XP)

Fig. 1.3: Overview of the problems in this chapter

is such a complexity theory providing lower bounds on the running time required to solve parameterized problems.

Finding cliques — with a different parameter

OK, so there probably is no algorithm for solving CLIQUE with running time $f(k) \cdot n^{o(k)}$. But what about those scary groups of people that might come for you if you refuse the wrong person at the door? They do not care at all about the computational hardness of CLIQUE, and neither do their fists. What can you do? Well, in Norway most people do not have too many friends. In fact, it is quite unheard of that someone has more than $\Delta = 20$ friends. That means that we are trying to find a k -clique in a graph of maximum degree Δ . This can be done quite efficiently: if we guess one vertex v in the clique, then the remaining vertices in the clique must be among the Δ neighbors of v . Thus we can try all of the 2^Δ subsets of the neighbors of v , and return the largest clique that we found. The total running time of this algorithm is $\mathcal{O}(2^\Delta \cdot \Delta^2 \cdot n)$, which is quite feasible for $\Delta = 20$. Again it is possible to use complexity theoretic assumptions on the hardness of CNF-SAT to show that this algorithm is asymptotically optimal, up to multiplicative constants in the exponent.

What the algorithm above shows is that the CLIQUE problem is FPT when the parameter is the maximum degree Δ of the input graph. At the same time CLIQUE is probably not FPT when the parameter is the solution size k . Thus, the classification of the problem into “tractable” or “intractable” crucially depends on the choice of parameter. This makes a lot of sense; the more we know about our input instances, the more we can exploit algorithmically!

The art of parameterization

For typical optimization problems, one can immediately find a relevant parameter: the size of the solution we are looking for. In some cases, however, it is not completely obvious what we mean by the size of the solution. For example, consider the variant of BAR FIGHT PREVENTION where we want to reject at most k guests such that the number of conflicts is reduced to at most ℓ (as we believe that the bouncers at the bar can handle ℓ conflicts, but not more). Then we can parameterize either by k or by ℓ . We may even parameterize by both: then the goal is to find an FPT algorithm with running time $f(k, \ell) \cdot n^c$ for some computable function f depending only on k and ℓ . Thus the theory of parameterization and FPT algorithms can be extended to considering a set of parameters at the same time. Formally, however, one can express parameterization by k and ℓ simply by defining the value $k + \ell$ to be the parameter: an $f(k, \ell) \cdot n^c$ algorithm exists if and only if an $f(k + \ell) \cdot n^c$ algorithm exists.

The parameters k and ℓ in the extended BAR FIGHT PREVENTION example of the previous paragraph are related to the *objective* of the problem: they are parameters explicitly given in the input, defining the properties of the solution we are looking for. We get more examples of this type of parameter if we define variants of BAR FIGHT PREVENTION where we need to reject at most k guests such that, say, the number of conflicts decreases by p , or such that each accepted guest has conflicts with at most d other accepted guests, or such that the average number of conflicts per guest is at most a . Then the parameters p , d , and a are again explicitly given in the input, telling us what kind of solution we need to find. The parameter Δ (maximum degree of the graph) in the CLIQUE example is a parameter of a very different type: it is not given explicitly in the input, but it is a *measure* of some property of the input instance. We defined and explored this particular measure because we believed that it is typically small in the input instances we care about: this parameter expresses some structural property of typical instances. We can identify and investigate any number of such parameters. For example, in problems involving graphs, we may parameterize by any structural parameter of the graph at hand. Say, if we believe that the problem is easy on planar graphs and the instances are “almost planar”, then we may explore the parameterization by the genus of the graph (roughly speaking, a graph has genus g if it can be drawn without edge crossings on a sphere with g holes in it). A large part of Chapter 7 (and also Chapter 11) is devoted to parameterization by treewidth, which is a very important parameter measuring the “tree-likeness” of the graph. For problems involving satisfiability of Boolean formulas, we can have such parameters as the number of variables, or clauses, or the number of clauses that need to be satisfied, or that are allowed not to be satisfied. For problems involving a set of strings, one can parameterize by the maximum length of the strings, by the size of the alphabet, by the maximum number of distinct symbols appearing in each string, etc. In

a problem involving a set of geometric objects (say, points in space, disks, or polygons), one may parameterize by the maximum number of vertices of each polygon or the dimension of the space where the problem is defined. For each problem, with a bit of creativity, one can come up with a large number of (combinations of) parameters worth studying.

For the same problem there can be multiple choices of parameters. Selecting the right parameter(s) for a particular problem is an art.

Parameterized complexity allows us to study how different parameters influence the complexity of the problem. A successful parameterization of a problem needs to satisfy two properties. First, we should have some reason to believe that the selected parameter (or combination of parameters) is typically small on input instances in some application. Second, we need efficient algorithms where the combinatorial explosion is restricted to the parameter(s), that is, we want the problem to be FPT with this parameterization. Finding good parameterizations is an art on its own and one may spend quite some time on analyzing different parameterizations of the same problem. However, in this book we focus more on explaining algorithmic techniques via carefully chosen illustrative examples, rather than discussing every possible aspect of a particular problem. Therefore, even though different parameters and parameterizations will appear throughout the book, we will not try to give a complete account of all known parameterizations and results for any concrete problem.

1.1 Formal definitions

We finish this chapter by leaving the realm of pub jokes and moving to more serious matters. Before we start explaining the techniques for designing parameterized algorithms, we need to introduce formal foundations of parameterized complexity. That is, we need to have rigorous definitions of what a parameterized problem is, and what it means that a parameterized problem belongs to a specific complexity class.

Definition 1.1. A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is called the *parameter*.

For example, an instance of CLIQUE parameterized by the solution size is a pair (G, k) , where we expect G to be an undirected graph encoded as a string over Σ , and k is a positive integer. That is, a pair (G, k) belongs to the CLIQUE parameterized language if and only if the string G correctly encodes an undirected graph, which we will also denote by G , and moreover the graph

G contains a clique on k vertices. Similarly, an instance of the CNF-SAT problem (satisfiability of propositional formulas in CNF), parameterized by the number of variables, is a pair (φ, n) , where we expect φ to be the input formula encoded as a string over Σ and n to be the number of variables of φ . That is, a pair (φ, n) belongs to the CNF-SAT parameterized language if and only if the string φ correctly encodes a CNF formula with n variables, and the formula is satisfiable.

We define the size of an instance (x, k) of a parameterized problem as $|x| + k$. One interpretation of this convention is that, when given to the algorithm on the input, the parameter k is encoded in unary.

Definition 1.2. A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *fixed-parameter tractable* (FPT) if there exists an algorithm \mathcal{A} (called a *fixed-parameter algorithm*), a computable function $f: \mathbb{N} \rightarrow \mathbb{N}$, and a constant c such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm \mathcal{A} correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called FPT.

Before we go further, let us make some remarks about the function f in this definition. Observe that we assume f to be computable, as otherwise we would quickly run into trouble when developing complexity theory for fixed-parameter tractability. For technical reasons, it will be convenient to assume, from now on, that f is also nondecreasing. Observe that this assumption has no influence on the definition of fixed-parameter tractability as stated in Definition 1.2 since for every computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ there exists a computable nondecreasing function \bar{f} that is never smaller than f : we can simply take $\bar{f}(k) = \max_{i=0,1,\dots,k} f(i)$. Also, for standard algorithmic results it is always the case that the bound on the running time is a nondecreasing function of the complexity measure, so this assumption is indeed satisfied in practice. However, the assumption about f being nondecreasing is formally needed in various situations, for example when performing reductions.

We now define the complexity class XP.

Definition 1.3. A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *slice-wise polynomial* (XP) if there exists an algorithm \mathcal{A} and two computable functions $f, g: \mathbb{N} \rightarrow \mathbb{N}$ such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm \mathcal{A} correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^{g(k)}$. The complexity class containing all slice-wise polynomial problems is called XP.

Again, we shall assume that the functions f, g in this definition are nondecreasing.

The definition of a parameterized problem, as well as the definitions of the classes FPT and XP, can easily be generalized to encompass multiple parameters. In this setting we simply allow k to be not just one nonnegative

integer, but a vector of d nonnegative integers, for some fixed constant d . Then the functions f and g in the definitions of the complexity classes FPT and XP can depend on all these parameters.

Just as “polynomial time” and “polynomial-time algorithm” usually refer to time polynomial in the input size, the terms “FPT time” and “FPT algorithms” refer to time $f(k)$ times a polynomial in the input size. Here f is a computable function of k and the degree of the polynomial is independent of both n and k . The same holds for “XP time” and “XP algorithms”, except that here the degree of the polynomial is allowed to depend on the parameter k , as long as it is upper bounded by $g(k)$ for some computable function g .

Observe that, given some parameterized problem L , the algorithm designer has essentially two different optimization goals when designing FPT algorithms for L . Since the running time has to be of the form $f(k) \cdot n^c$, one can:

- optimize the *parametric dependence* of the running time, i.e., try to design an algorithm where function f grows as slowly as possible; or
- optimize the *polynomial factor* in the running time, i.e., try to design an algorithm where constant c is as small as possible.

Both these goals are equally important, from both a theoretical and a practical point of view. Unfortunately, keeping track of and optimizing both factors of the running time can be a very difficult task. For this reason, most research on parameterized algorithms concentrates on optimizing one of the factors, and putting more focus on each of them constitutes one of the two dominant trends in parameterized complexity. Sometimes, when we are not interested in the exact value of the polynomial factor, we use the *\mathcal{O}^* -notation*, which suppresses factors polynomial in the input size. More precisely, a running time $\mathcal{O}^*(f(k))$ means that the running time is upper bounded by $f(k) \cdot n^{\mathcal{O}(1)}$, where n is the input size.

The theory of parameterized complexity has been pioneered by Downey and Fellows over the last two decades [148] [149] [150] [151] [153]. The main achievement of their work is a comprehensive complexity theory for parameterized problems, with appropriate notions of reduction and completeness. The primary goal is to understand the qualitative difference between fixed-parameter tractable problems, and problems that do not admit such efficient algorithms. The theory contains a rich “positive” toolkit of techniques for developing efficient parameterized algorithms, as well as a corresponding “negative” toolkit that supports a theory of parameterized intractability. This textbook is mostly devoted to a presentation of the positive toolkit: in Chapters 2 through 12 we present various algorithmic techniques for designing fixed-parameter tractable algorithms. As we have argued, the process of algorithm design has to use both toolkits in order to be able to conclude that certain research directions are pointless. Therefore, in Part III we give an introduction to lower bounds for parameterized problems.

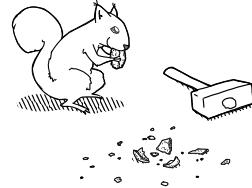
Bibliographic notes

Downey and Fellows laid the foundation of parameterized complexity in the series of papers [149, 150, 151]. The classic reference on parameterized complexity is the book of Downey and Fellows [153]. The new edition of this book [154] is a comprehensive overview of the state of the art in many areas of parameterized complexity. The book of Flum and Grohe [189] is an extensive introduction to the area with a strong emphasis on the complexity viewpoint. An introduction to basic algorithmic techniques in parameterized complexity up to 2006 is given in the book of Niedermeier [376]. The recent book [51] contains a collection of surveys on different areas of parameterized complexity.

Chapter 2

Kernelization

Kernelization is a systematic approach to study polynomial-time preprocessing algorithms. It is an important tool in the design of parameterized algorithms. In this chapter we explain basic kernelization techniques such as crown decomposition, the expansion lemma, the sunflower lemma, and linear programming. We illustrate these techniques by obtaining kernels for VERTEX COVER, FEEDBACK ARC SET IN TOURNAMENTS, EDGE CLIQUE COVER, MAXIMUM SATISFIABILITY, and d -HITTING SET.



Preprocessing (data reduction or kernelization) is used universally in almost every practical computer implementation that aims to deal with an NP-hard problem. The goal of a preprocessing subroutine is to solve efficiently the “easy parts” of a problem instance and reduce it (shrink it) to its computationally difficult “core” structure (the *problem kernel* of the instance). In other words, the idea of this method is to reduce (but not necessarily solve) the given problem instance to an equivalent “smaller sized” instance in time polynomial in the input size. A slower exact algorithm can then be run on this smaller instance.

How can we measure the effectiveness of such a preprocessing subroutine? Suppose we define a useful preprocessing algorithm as one that runs in polynomial time and replaces an instance I with an equivalent instance that is at least one bit smaller. Then the existence of such an algorithm for an NP-hard problem would imply $P = NP$, making it unlikely that such an algorithm can be found. For a long time, there was no other suggestion for a formal definition of useful preprocessing, leaving the mathematical analysis of polynomial-time preprocessing algorithms largely neglected. But in the language of parameterized complexity, we can formulate a definition of useful preprocessing by demanding that large instances with a small parameter should be shrunk, while instances that are small compared to their parameter

do not have to be processed further. These ideas open up the “lost continent” of polynomial-time algorithms called *kernelization*.

In this chapter we illustrate some commonly used techniques to design kernelization algorithms through concrete examples. The next section, Section 2.1, provides formal definitions. In Section 2.2 we give kernelization algorithms based on so-called natural reduction rules. Section 2.3 introduces the concepts of crown decomposition and the expansion lemma, and illustrates it on MAXIMUM SATISFIABILITY. Section 2.5 studies tools based on linear programming and gives a kernel for VERTEX COVER. Finally, we study the sunflower lemma in Section 2.6 and use it to obtain a polynomial kernel for d -HITTING SET.

2.1 Formal definitions

We now turn to the formal definition that captures the notion of kernelization. A *data reduction rule*, or simply, reduction rule, for a parameterized problem Q is a function $\phi: \Sigma^* \times \mathbb{N} \rightarrow \Sigma^* \times \mathbb{N}$ that maps an instance (I, k) of Q to an equivalent instance (I', k') of Q such that ϕ is computable in time polynomial in $|I|$ and k . We say that two instances of Q are *equivalent* if $(I, k) \in Q$ if and only if $(I', k') \in Q$; this property of the reduction rule ϕ , that it translates an instance to an equivalent one, is sometimes referred to as the *safeness* or *soundness* of the reduction rule. In this book, we stick to the phrases: *a rule is safe* and *the safeness of a reduction rule*.

The general idea is to design a *preprocessing algorithm* that consecutively applies various data reduction rules in order to shrink the instance size as much as possible. Thus, such a preprocessing algorithm takes as input an instance $(I, k) \in \Sigma^* \times \mathbb{N}$ of Q , works in polynomial time, and returns an equivalent instance (I', k') of Q . In order to formalize the requirement that the output instance has to be small, we apply the main principle of Parameterized Complexity: The complexity is measured in terms of the parameter. Consequently, the *output size* of a preprocessing algorithm \mathcal{A} is a function $\text{size}_{\mathcal{A}}: \mathbb{N} \rightarrow \mathbb{N} \cup \{\infty\}$ defined as follows:

$$\text{size}_{\mathcal{A}}(k) = \sup\{|I'| + k' : (I', k') = \mathcal{A}(I, k), I \in \Sigma^*\}.$$

In other words, we look at all possible instances of Q with a fixed parameter k , and measure the supremum of the sizes of the output of \mathcal{A} on these instances. Note that this supremum may be infinite; this happens when we do not have any bound on the size of $\mathcal{A}(I, k)$ in terms of the input parameter k only. *Kernelization algorithms* are exactly these preprocessing algorithms whose output size is finite and bounded by a computable function of the parameter.

Definition 2.1 (Kernelization, kernel). A *kernelization algorithm*, or simply a *kernel*, for a parameterized problem Q is an algorithm \mathcal{A} that, given

an instance (I, k) of Q , works in polynomial time and returns an equivalent instance (I', k') of Q . Moreover, we require that $\text{size}_{\mathcal{A}}(k) \leq g(k)$ for some computable function $g: \mathbb{N} \rightarrow \mathbb{N}$.

The size requirement in this definition can be reformulated as follows: There exists a computable function $g(\cdot)$ such that whenever (I', k') is the output for an instance (I, k) , then it holds that $|I'| + k' \leq g(k)$. If the upper bound $g(\cdot)$ is a polynomial (linear) function of the parameter, then we say that Q admits a *polynomial (linear) kernel*. We often abuse the notation and call the output of a kernelization algorithm the “reduced” equivalent instance, also a kernel.

In the course of this chapter, we will often encounter a situation when in some boundary cases we are able to completely resolve the considered problem instance, that is, correctly decide whether it is a yes-instance or a no-instance. Hence, for clarity, we allow the reductions (and, consequently, the kernelization algorithm) to return a yes/no answer instead of a reduced instance. Formally, to fit into the introduced definition of a kernel, in such cases the kernelization algorithm should instead return a constant-size trivial yes-instance or no-instance. Note that such instances exist for every parameterized language except for the empty one and its complement, and can be therefore hardcoded into the kernelization algorithm.

Recall that, given an instance (I, k) of Q , the size of the kernel is defined as the number of *bits* needed to encode the reduced equivalent instance I' plus the parameter value k' . However, when dealing with problems on graphs, hypergraphs, or formulas, often we would like to emphasize other aspects of output instances. For example, for a graph problem Q , we could say that Q admits a kernel with $\mathcal{O}(k^3)$ vertices and $\mathcal{O}(k^5)$ edges to emphasize the upper bound on the number of vertices and edges in the output instances. Similarly, for a problem defined on formulas, we could say that the problem admits a kernel with $\mathcal{O}(k)$ variables.

It is important to mention here that the early definitions of kernelization required that $k' \leq k$. On an intuitive level this makes sense, as the parameter k measures the complexity of the problem — thus the larger the k , the harder the problem. This requirement was subsequently relaxed, notably in the context of lower bounds. An advantage of the more liberal notion of kernelization is that it is robust with respect to polynomial transformations of the kernel. However, it limits the connection with practical preprocessing. All the kernels mentioned in this chapter respect the fact that the output parameter is at most the input parameter, that is, $k' \leq k$.

While usually in Computer Science we measure the efficiency of an algorithm by estimating its running time, the central measure of the efficiency of a kernelization algorithm is a bound on its output size. Although the actual running time of a kernelization algorithm is of-

ten very important for practical applications, in theory a kernelization algorithm is only required to run in polynomial time.

If we have a kernelization algorithm for a problem for which there is some algorithm (with any running time) to decide whether (I, k) is a yes-instance, then clearly the problem is FPT, as the size of the reduced instance I is simply a function of k (and independent of the input size n). However, a surprising result is that the converse is also true.

Lemma 2.2. *If a parameterized problem Q is FPT then it admits a kernelization algorithm.*

Proof. Since Q is FPT, there is an algorithm \mathcal{A} deciding if $(I, k) \in Q$ in time $f(k) \cdot |I|^c$ for some computable function f and a constant c . We obtain a kernelization algorithm for Q as follows. Given an input (I, k) , the kernelization algorithm runs \mathcal{A} on (I, k) , for at most $|I|^{c+1}$ steps. If it terminates with an answer, use that answer to return either that (I, k) is a yes-instance or that it is a no-instance. If \mathcal{A} does not terminate within $|I|^{c+1}$ steps, then return (I, k) itself as the output of the kernelization algorithm. Observe that since \mathcal{A} did not terminate in $|I|^{c+1}$ steps, we have that $f(k) \cdot |I|^c > |I|^{c+1}$, and thus $|I| < f(k)$. Consequently, we have $|I| + k \leq f(k) + k$, and we obtain a kernel of size at most $f(k) + k$; note that this upper bound is computable as $f(k)$ is a computable function. \square

Lemma 2.2 implies that a decidable problem admits a kernel if and only if it is fixed-parameter tractable. Thus, in a sense, kernelization can be another way of defining fixed-parameter tractability.

However, kernels obtained by this theoretical result are usually of exponential (or even worse) size, while problem-specific data reduction rules often achieve quadratic ($g(k) = \mathcal{O}(k^2)$) or even linear-size ($g(k) = \mathcal{O}(k)$) kernels. So a natural question for any concrete FPT problem is whether it admits a problem kernel that is bounded by a polynomial function of the parameter ($g(k) = k^{\mathcal{O}(1)}$). In this chapter we give polynomial kernels for several problems using some elementary methods. In Chapter 9 we give more advanced methods for obtaining kernels.

2.2 Some simple kernels

In this section we give kernelization algorithms for VERTEX COVER and FEEDBACK ARC SET IN TOURNAMENTS (FAST) based on a few natural reduction rules.

2.2.1 VERTEX COVER

Let G be a graph and $S \subseteq V(G)$. The set S is called a *vertex cover* if for every edge of G at least one of its endpoints is in S . In other words, the graph $G - S$ contains no edges and thus $V(G) \setminus S$ is an *independent set*. In the VERTEX COVER problem, we are given a graph G and a positive integer k as input, and the objective is to check whether there exists a vertex cover of size at most k .

The first reduction rule is based on the following simple observation. For a given instance (G, k) of VERTEX COVER, if the graph G has an isolated vertex, then this vertex does not cover any edge and thus its removal does not change the solution. This shows that the following rule is safe.

Reduction VC.1. If G contains an isolated vertex v , delete v from G . The new instance is $(G - v, k)$.

The second rule is based on the following natural observation:

If G contains a vertex v of degree more than k , then v should be in every vertex cover of size at most k .

Indeed, this is because if v is not in a vertex cover, then we need at least $k + 1$ vertices to cover edges incident to v . Thus our second rule is the following.

Reduction VC.2. If there is a vertex v of degree at least $k + 1$, then delete v (and its incident edges) from G and decrement the parameter k by 1. The new instance is $(G - v, k - 1)$.

Observe that exhaustive application of reductions VC.1 and VC.2 completely removes the vertices of degree 0 and degree at least $k + 1$. The next step is the following observation.

If a graph has maximum degree d , then a set of k vertices can cover at most kd edges.

This leads us to the following lemma.

Lemma 2.3. *If (G, k) is a yes-instance and none of the reduction rules VC.1, VC.2 is applicable to G , then $|V(G)| \leq k^2 + k$ and $|E(G)| \leq k^2$.*

Proof. Because we cannot apply Reductions VC.1 anymore on G , G has no isolated vertices. Thus for every vertex cover S of G , every vertex of $G - S$ should be adjacent to some vertex from S . Since we cannot apply Reductions VC.2 every vertex of G has degree at most k . It follows that

$|V(G - S)| \leq k|S|$ and hence $|V(G)| \leq (k + 1)|S|$. Since (G, k) is a yes-instance, there is a vertex cover S of size at most k , so $|V(G)| \leq (k + 1)k$. Also every edge of G is covered by some vertex from a vertex cover and every vertex can cover at most k edges. Hence if G has more than k^2 edges, this is again a no-instance. \square

Lemma 2.3 allows us to claim the final reduction rule that explicitly bounds the size of the kernel.

Reduction VC.3. Let (G, k) be an input instance such that Reductions VC.1 and VC.2 are not applicable to (G, k) . If $k < 0$ and G has more than $k^2 + k$ vertices, or more than k^2 edges, then conclude that we are dealing with a no-instance.

Finally, we remark that all reduction rules are trivially applicable in linear time. Thus, we obtain the following theorem.

Theorem 2.4. VERTEX COVER admits a kernel with $\mathcal{O}(k^2)$ vertices and $\mathcal{O}(k^2)$ edges.

2.2.2 FEEDBACK ARC SET IN TOURNAMENTS

In this section we discuss a kernel for the FEEDBACK ARC SET IN TOURNA-
MENTS problem. A *tournament* is a directed graph T such that for every pair of vertices $u, v \in V(T)$, exactly one of (u, v) or (v, u) is a directed edge (also often called an *arc*) of T . A set of edges A of a directed graph G is called a *feedback arc set* if every directed cycle of G contains an edge from A . In other words, the removal of A from G turns it into a directed acyclic graph. Very often, acyclic tournaments are called *transitive* (note that then $E(G)$ is a transitive relation). In the FEEDBACK ARC SET IN TOURNA-
MENTS problem we are given a tournament T and a nonnegative integer k . The objective is to decide whether T has a feedback arc set of size at most k .

For tournaments, the deletion of edges results in directed graphs which are not tournaments anymore. Because of that, it is much more convenient to use the characterization of a feedback arc set in terms of “reversing edges”. We start with the following well-known result about *topological orderings* of directed acyclic graphs.

Lemma 2.5. A directed graph G is acyclic if and only if it is possible to order its vertices in such a way such that for every directed edge (u, v) , we have $u < v$.

We leave the proof of Lemma 2.5 as an exercise; see Exercise 2.1. Given a directed graph G and a subset $F \subseteq E(G)$ of edges, we define $G \circledast F$ to be the directed graph obtained from G by reversing all the edges of F . That is, if $\text{rev}(F) = \{(u, v) : (v, u) \in F\}$, then for $G \circledast F$ the vertex set is $V(G)$

and the edge set $E(G \circledast F) = (E(G) \cup \text{rev}(F)) \setminus F$. Lemma 2.5 implies the following.

Observation 2.6. Let G be a directed graph and let F be a subset of edges of G . If $G \circledast F$ is a directed acyclic graph then F is a feedback arc set of G .

The following lemma shows that, in some sense, the opposite direction of the statement in Observation 2.6 is also true. However, the minimality condition in Lemma 2.7 is essential, see Exercise 2.2.

Lemma 2.7. *Let G be a directed graph and F be a subset of $E(G)$. Then F is an inclusion-wise minimal feedback arc set of G if and only if F is an inclusion-wise minimal set of edges such that $G \circledast F$ is an acyclic directed graph.*

Proof. We first prove the forward direction of the lemma. Let F be an inclusion-wise minimal feedback arc set of G . Assume to the contrary that $G \circledast F$ has a directed cycle C . Then C cannot contain only edges of $E(G) \setminus F$, as that would contradict the fact that F is a feedback arc set. Let f_1, f_2, \dots, f_ℓ be the edges of $C \cap \text{rev}(F)$ in the order of their appearance on the cycle C , and let $e_i \in F$ be the edge f_i reversed. Since F is inclusion-wise minimal, for every e_i , there exists a directed cycle C_i in G such that $F \cap C_i = \{e_i\}$. Now consider the following closed walk W in G : we follow the cycle C , but whenever we are to traverse an edge $f_i \in \text{rev}(F)$ (which is not present in G), we instead traverse the path $C_i - e_i$. By definition, W is a closed walk in G and, furthermore, note that W does not contain any edge of F . This contradicts the fact that F is a feedback arc set of G .

The minimality follows from Observation 2.6. That is, every set of edges F such that $G \circledast F$ is acyclic is also a feedback arc set of G , and thus, if F is not a minimal set such that $G \circledast F$ is acyclic, then it will contradict the fact that F is a minimal feedback arc set.

For the other direction, let F be an inclusion-wise minimal set of edges such that $G \circledast F$ is an acyclic directed graph. By Observation 2.6, F is a feedback arc set of G . Moreover, F is an inclusion-wise minimal feedback arc set, because if a proper subset F' of F is an inclusion-wise minimal feedback arc set of G , then by the already proved implication of the lemma, $G \circledast F'$ is an acyclic directed graph, a contradiction with the minimality of F . \square

We are ready to give a kernel for FEEDBACK ARC SET IN TOURNAMENTS.

Theorem 2.8. FEEDBACK ARC SET IN TOURNAMENTS *admits a kernel with at most $k^2 + 2k$ vertices.*

Proof. Lemma 2.7 implies that a tournament T has a feedback arc set of size at most k if and only if it can be turned into an acyclic tournament by reversing directions of at most k edges. We will use this characterization for the kernel.

In what follows by a *triangle* we mean a directed cycle of length three. We give two simple reduction rules.

Reduction FAST.1. If an edge e is contained in at least $k + 1$ triangles, then reverse e and reduce k by 1.

Reduction FAST.2. If a vertex v is not contained in any triangle, then delete v from T .

The rules follow similar guidelines as in the case of VERTEX COVER. In Reduction **FAST.1**, we greedily take into a solution an edge that participates in $k + 1$ otherwise disjoint forbidden structures (here, triangles). In Reduction **FAST.2**, we discard vertices that do not participate in any forbidden structure, and should be irrelevant to the problem.

However, a formal proof of the safeness of Reduction **FAST.2** is not immediate: we need to verify that deleting v and its incident edges does not make a yes-instance out of a no-instance.

Note that after applying any of the two rules, the resulting graph is again a tournament. The first rule is safe because if we do not reverse e , we have to reverse at least one edge from each of $k + 1$ triangles containing e . Thus e belongs to every feedback arc set of size at most k .

Let us now prove the safeness of the second rule. Let $X = N^+(v)$ be the set of heads of directed edges with tail v and let $Y = N^-(v)$ be the set of tails of directed edges with head v . Because T is a tournament, X and Y is a partition of $V(T) \setminus \{v\}$. Since v is not a part of any triangle in T , we have that there is no edge from X to Y (with head in Y and tail in X). Consequently, for any feedback arc set A_1 of tournament $T[X]$ and any feedback arc set A_2 of tournament $T[Y]$, the set $A_1 \cup A_2$ is a feedback arc set of T . As the reverse implication is trivial (for any feedback arc set A in T , $A \cap E(T[X])$ is a feedback arc set of $T[X]$, and $A \cap E(T[Y])$ is a feedback arc set of $T[Y]$), we have that (T, k) is a yes-instance if and only if $(T - v, k)$ is.

Finally, we show that every reduced yes-instance T , an instance on which none of the presented reduction rules are applicable, has at most $k(k + 2)$ vertices. Let A be a feedback arc set of a reduced instance T of size at most k . For every edge $e \in A$, aside from the two endpoints of e , there are at most k vertices that are in triangles containing e — otherwise we would be able to apply Reduction **FAST.1**. Since every triangle in T contains an edge of A and every vertex of T is in a triangle, we have that T has at most $k(k + 2)$ vertices.

Thus, given (T, k) we apply our reduction rules exhaustively and obtain an equivalent instance (T', k') . If T' has more than $k'^2 + k'$ vertices, then the algorithm returns that (T, k) is a no-instance, otherwise we get the desired kernel. This completes the proof of the theorem. \square

2.2.3 EDGE CLIQUE COVER

Not all FPT problems admit polynomial kernels. In the EDGE CLIQUE COVER problem, we are given a graph G and a nonnegative integer k , and the goal is to decide whether the edges of G can be covered by at most k cliques. In this section we give an exponential kernel for EDGE CLIQUE COVER. In Theorem 14.20 of Section *14.3.3, we remark that this simple kernel is essentially optimal.

Let us recall the reader that we use $N(v) = \{u : uv \in E(G)\}$ to denote the neighborhood of vertex v in G , and $N[v] = N(v) \cup \{v\}$ to denote the closed neighborhood of v . We apply the following data reduction rules in the given order (i.e., we always use the lowest-numbered rule that modifies the instance).

Reduction ECC.1. Remove isolated vertices.

Reduction ECC.2. If there is an isolated edge uv (a connected component that is just an edge), delete it and decrease k by 1. The new instance is $(G - \{u, v\}, k - 1)$.

Reduction ECC.3. If there is an edge uv whose endpoints have exactly the same closed neighborhood, that is, $N[u] = N[v]$, then delete v . The new instance is $(G - v, k)$.

The crux of the presented kernel for EDGE CLIQUE COVER is an observation that two true twins (vertices u and v with $N[u] = N[v]$) can be treated in exactly the same way in some optimum solution, and hence we can reduce them. Meanwhile, the vertices that are contained in exactly the same set of cliques in a feasible solution *have to* be true twins. This observation bounds the size of the kernel.

The safeness of the first two reductions is trivial, while the safeness of Reduction ECC.3 follows from the observation that a solution in $G - v$ can be extended to a solution in G by adding v to all the cliques containing u (see Exercise 2.3).

Theorem 2.9. EDGE CLIQUE COVER admits a kernel with at most 2^k vertices.

Proof. We start with the following claim.

Claim. If (G, k) is a reduced yes-instance, on which none of the presented reduction rules can be applied, then $|V(G)| \leq 2^k$.

Proof. Let C_1, \dots, C_k be an edge clique cover of G . We claim that G has at most 2^k vertices. Targeting a contradiction, let us assume that G has more

than 2^k vertices. We assign to each vertex $v \in V(G)$ a binary vector b_v of length k , where bit i , $1 \leq i \leq k$, is set to 1 if and only if v is contained in clique C_i . Since there are only 2^k possible vectors, there must be $u \neq v \in V(G)$ with $b_u = b_v$. If b_u and b_v are zero vectors, the first rule applies; otherwise, u and v are contained in the same cliques. This means that u and v are adjacent and have the same neighborhood; thus either Reduction [ECC.2](#) or Reduction [ECC.3](#) applies. Hence, if G has more than 2^k vertices, at least one of the reduction rules can be applied to it, which is a contradiction to the initial assumption that G is reduced. This completes the proof of the claim. \square

The kernelization algorithm works as follows. Given an instance (G, k) , it applies Reductions [ECC.1](#) [ECC.2](#) and [ECC.3](#) exhaustively. If the resulting graph has more than 2^k vertices the kernelization algorithm outputs that the input instance is a no-instance, else it outputs the reduced instance. \square

2.3 Crown decomposition

Crown decomposition is a general kernelization technique that can be used to obtain kernels for many problems. The technique is based on the classical matching theorems of Kőnig and Hall.

Recall that for disjoint vertex subsets U, W of a graph G , a matching M is called a *matching of U into W* if every edge of M connects a vertex of U and a vertex of W and, moreover, every vertex of U is an endpoint of some edge of M . In this situation, we also say that M *saturates U* .

Definition 2.10 (Crown decomposition). A *crown decomposition* of a graph G is a partitioning of $V(G)$ into three parts C, H and R , such that

1. C is nonempty.
2. C is an independent set.
3. There are no edges between vertices of C and R . That is, H separates C and R .
4. Let E' be the set of edges between vertices of C and H . Then E' contains a matching of size $|H|$. In other words, G contains a matching of H into C .

The set C can be seen as a crown put on head H of the remaining part R , see Fig. [2.1](#). Note that the fact that E' contains a matching of size $|H|$ implies that there is a matching of H into C . This is a matching in the subgraph G' , with the vertex set $C \cup H$ and the edge set E' , saturating all the vertices of H .

For finding a crown decomposition in polynomial time, we use the following well known structural and algorithmic results. The first is a mini-max theorem due to Kőnig.

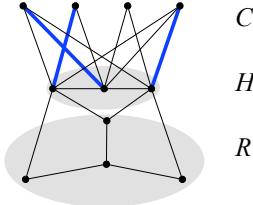


Fig. 2.1: Example of a crown decomposition. Set C is an independent set, H separates C and R , and there is a matching of H into C

Theorem 2.11 (König's theorem, [303]). *In every undirected bipartite graph the size of a maximum matching is equal to the size of a minimum vertex cover.*

Let us recall that a matching *saturates* a set of vertices S when every vertex in S is incident to an edge in the matching. The second classic result states that in bipartite graphs, a trivial necessary condition for the existence of a matching is also sufficient.

Theorem 2.12 (Hall's theorem, [256]). *Let G be an undirected bipartite graph with bipartition (V_1, V_2) . The graph G has a matching saturating V_1 if and only if for all $X \subseteq V_1$, we have $|N(X)| \geq |X|$.*

The following theorem is due to Hopcroft and Karp [268]. The proof of the (nonstandard) second claim of the theorem is deferred to Exercise 2.21.

Theorem 2.13 (Hopcroft-Karp algorithm, [268]). *Let G be an undirected bipartite graph with bipartition V_1 and V_2 , on n vertices and m edges. Then we can find a maximum matching as well as a minimum vertex cover of G in time $\mathcal{O}(m\sqrt{n})$. Furthermore, in time $\mathcal{O}(m\sqrt{n})$ either we can find a matching saturating V_1 or an inclusion-wise minimal set $X \subseteq V_1$ such that $|N(X)| < |X|$.*

The following lemma is the basis for kernelization algorithms using crown decomposition.

Lemma 2.14 (Crown lemma). *Let G be a graph without isolated vertices and with at least $3k + 1$ vertices. There is a polynomial-time algorithm that either*

- finds a matching of size $k + 1$ in G ; or
- finds a crown decomposition of G .

Proof. We first find an inclusion-maximal matching M in G . This can be done by a greedy algorithm. If the size of M is $k + 1$, then we are done.

Hence, we assume that $|M| \leq k$, and let V_M be the endpoints of M . We have $|V_M| \leq 2k$. Because M is a maximal matching, the remaining set of vertices $I = V(G) \setminus V_M$ is an independent set.

Consider the bipartite graph G_{I,V_M} formed by edges of G between V_M and I . We compute a minimum-sized vertex cover X and a maximum sized matching M' of the bipartite graph G_{I,V_M} in polynomial time using Theorem 2.13. We can assume that $|M'| \leq k$, for otherwise we are done. Since $|X| = |M'|$ by Kőnig's theorem (Theorem 2.11), we infer that $|X| \leq k$.

If no vertex of X is in V_M , then $X \subseteq I$. We claim that $X = I$. For a contradiction assume that there is a vertex $w \in I \setminus X$. Because G has no isolated vertices there is an edge, say wz , incident to w in G_{I,V_M} . Since G_{I,V_M} is bipartite, we have that $z \in V_M$. However, X is a vertex cover of G_{I,V_M} such that $X \cap V_M = \emptyset$, which implies that $w \in X$. This is contrary to our assumption that $w \notin X$, thus proving that $X = I$. But then $|I| \leq |X| \leq k$, and G has at most

$$|I| + |V_M| \leq k + 2k = 3k$$

vertices, which is a contradiction.

Hence, $X \cap V_M \neq \emptyset$. We obtain a crown decomposition (C, H, R) as follows. Since $|X| = |M'|$, every edge of the matching M' has exactly one endpoint in X . Let M^* denote the subset of M' such that every edge from M^* has exactly one endpoint in $X \cap V_M$ and let V_{M^*} denote the set of endpoints of edges in M^* . We define head $H = X \cap V_M = X \cap V_{M^*}$, crown $C = V_{M^*} \cap I$, and the remaining part $R = V(G) \setminus (C \cup H) = V(G) \setminus V_{M^*}$. In other words, H is the set of endpoints of edges of M^* that are present in V_M and C is the set of endpoints of edges of M^* that are present in I . Obviously, C is an independent set and by construction, M^* is a matching of H into C . Furthermore, since X is a vertex cover of G_{I,V_M} , every vertex of C can be adjacent only to vertices of H and thus H separates C and R . This completes the proof. \square

The crown lemma gives a relatively strong structural property of graphs with a small vertex cover (equivalently, a small maximum matching). If in a studied problem the parameter upper bounds the size of a vertex cover (maximum matching), then there is a big chance that the structural insight given by the crown lemma would help in developing a small kernel — quite often with number of vertices bounded linearly in the parameter.

We demonstrate the application of crown decompositions on kernelization for VERTEX COVER and MAXIMUM SATISFIABILITY.

2.3.1 VERTEX COVER

Consider a VERTEX COVER instance (G, k) . By an exhaustive application of Reduction VC.1, we may assume that G has no isolated vertices. If $|V(G)| > 3k$, we may apply the crown lemma to the graph G and integer k , obtaining either a matching of size $k + 1$, or a crown decomposition $V(G) = C \cup H \cup R$. In the first case, the algorithm concludes that (G, k) is a no-instance.

In the latter case, let M be a matching of H into C . Observe that the matching M witnesses that, for every vertex cover X of the graph G , X contains at least $|M| = |H|$ vertices of $H \cup C$ to cover the edges of M . On the other hand, the set H covers all edges of G that are incident to $H \cup C$. Consequently, there exists a minimum vertex cover of G that contains H , and we may reduce (G, k) to $(G - H, k - |H|)$. Note that in the instance $(G - H, k - |H|)$, the vertices of C are isolated and will be reduced by Reduction VC.1.

As the crown lemma promises us that $H \neq \emptyset$, we can always reduce the graph as long as $|V(G)| > 3k$. Thus, we obtain the following.

Theorem 2.15. VERTEX COVER admits a kernel with at most $3k$ vertices.

2.3.2 MAXIMUM SATISFIABILITY

For a second application of the crown decomposition, we look at the following parameterized version of MAXIMUM SATISFIABILITY. Given a CNF formula F , and a nonnegative integer k , decide whether F has a truth assignment satisfying at least k clauses.

Theorem 2.16. MAXIMUM SATISFIABILITY admits a kernel with at most k variables and $2k$ clauses.

Proof. Let φ be a CNF formula with n variables and m clauses. Let ψ be an arbitrary assignment to the variables and let $\neg\psi$ be the assignment obtained by complementing the assignment of ψ . That is, if ψ assigns $\delta \in \{\top, \perp\}$ to some variable x then $\neg\psi$ assigns $\neg\delta$ to x . Observe that either ψ or $\neg\psi$ satisfies at least $m/2$ clauses, since every clause is satisfied by ψ or $\neg\psi$ (or by both). This means that, if $m \geq 2k$, then (φ, k) is a yes-instance. In what follows we give a kernel with $n < k$ variables.

Let G_φ be the *variable-clause* incidence graph of φ . That is, G_φ is a bipartite graph with bipartition (X, Y) , where X is the set of the variables of φ and Y is the set of clauses of φ . In G_φ there is an edge between a variable $x \in X$ and a clause $c \in Y$ if and only if either x , or its negation, is in c . If there is a matching of X into Y in G_φ , then there is a truth assignment satisfying at least $|X|$ clauses: we can set each variable in X in such a way that the clause matched to it becomes satisfied. Thus at least $|X|$ clauses are satisfied. Hence, in this case, if $k \leq |X|$, then (φ, k) is a yes-instance. Otherwise,

$k > |X| = n$, and we get the desired kernel. We now show that, if φ has at least $n \geq k$ variables, then we can, in polynomial time, either reduce φ to an equivalent smaller instance, or find an assignment to the variables satisfying at least k clauses (and conclude that we are dealing with a yes-instance).

Suppose φ has at least k variables. Using Hall's theorem and a polynomial-time algorithm computing a maximum-size matching (Theorems 2.12 and 2.13), we can in polynomial time find either a matching of X into Y or an inclusion-wise minimal set $C \subseteq X$ such that $|N(C)| < |C|$. As discussed in the previous paragraph, if we found a matching, then the instance is a yes-instance and we are done. So suppose we found a set C as described. Let H be $N(C)$ and $R = V(G_\varphi) \setminus (C \cup H)$. Clearly, $N(C) \subseteq H$, there are no edges between vertices of C and R and $G[C]$ is an independent set. Select an arbitrary $x \in C$. We have that there is a matching of $C \setminus \{x\}$ into H since $|N(C')| \geq |C'|$ for every $C' \subseteq C \setminus \{x\}$. Since $|C| > |H|$, we have that the matching from $C \setminus \{x\}$ to H is in fact a matching of H into C . Hence (C, H, R) is a crown decomposition of G_φ .

We prove that all clauses in H are satisfied in every assignment satisfying the maximum number of clauses. Indeed, consider any assignment ψ that does not satisfy all clauses in H . Fix any variable $x \in C$. For every variable y in $C \setminus \{x\}$ set the value of y so that the clause in H matched to y is satisfied. Let ψ' be the new assignment obtained from ψ in this manner. Since $N(C) \subseteq H$ and ψ' satisfies all clauses in H , more clauses are satisfied by ψ' than by ψ . Hence ψ cannot be an assignment satisfying the maximum number of clauses.

The argument above shows that (φ, k) is a yes-instance to MAXIMUM SATISFIABILITY if and only if $(\varphi \setminus H, k - |H|)$ is. This gives rise to the following simple reduction.

Reduction MSat.1. Let (φ, k) and H be as above. Then remove H from φ and decrease k by $|H|$. That is, $(\varphi \setminus H, k - |H|)$ is the new instance.

Repeated applications of Reduction MSat.1 and the arguments described above give the desired kernel. This completes the proof of the theorem. \square

2.4 Expansion lemma

In the previous subsection, we described crown decomposition techniques based on the classical Hall's theorem. In this section, we introduce a powerful variation of Hall's theorem, which is called the expansion lemma. This lemma captures a certain property of neighborhood sets in graphs and can be used to obtain polynomial kernels for many graph problems. We apply this result to get an $\mathcal{O}(k^2)$ kernel for FEEDBACK VERTEX SET in Chapter 9.

A q -star, $q \geq 1$, is a graph with $q + 1$ vertices, one vertex of degree q , called the *center*, and all other vertices of degree 1 adjacent to the center. Let G be

a bipartite graph with vertex bipartition (A, B) . For a positive integer q , a set of edges $M \subseteq E(G)$ is called by a *q -expansion of A into B* if

- every vertex of A is incident to exactly q edges of M ;
- M saturates exactly $q|A|$ vertices in B .

Let us emphasize that a q -expansion saturates all vertices of A . Also, for every $u, v \in A$, $u \neq v$, the set of vertices E_u adjacent to u by edges of M does not intersect the set of vertices E_v adjacent to v via edges of M , see Fig. 2.2. Thus every vertex $v \in A$ could be thought of as the center of a star with its q leaves in B , with all these $|A|$ stars being vertex-disjoint. Furthermore, a collection of these stars is also a family of q edge-disjoint matchings, each saturating A .

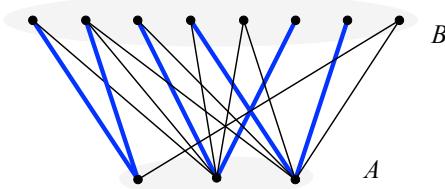


Fig. 2.2: Set A has a 2-expansion into B

Let us recall that, by Hall's theorem (Theorem 2.12), a bipartite graph with bipartition (A, B) has a matching of A into B if and only if $|N(X)| \geq |X|$ for all $X \subseteq A$. The following lemma is an extension of this result.

Lemma 2.17. *Let G be a bipartite graph with bipartition (A, B) . Then there is a q -expansion from A into B if and only if $|N(X)| \geq q|X|$ for every $X \subseteq A$. Furthermore, if there is no q -expansion from A into B , then a set $X \subseteq A$ with $|N(X)| < q|X|$ can be found in polynomial time.*

Proof. If A has a q -expansion into B , then trivially $|N(X)| \geq q|X|$ for every $X \subseteq A$.

For the opposite direction, we construct a new bipartite graph G' with bipartition (A', B) from G by adding $(q - 1)$ copies of all the vertices in A . For every vertex $v \in A$ all copies of v have the same neighborhood in B as v . We would like to prove that there is a matching M from A' into B in G' . If we prove this, then by identifying the endpoints of M corresponding to the copies of vertices from A , we obtain a q -expansion in G . It suffices to check that the assumptions of Hall's theorem are satisfied in G' . Assume otherwise, that there is a set $X \subseteq A'$ such that $|N_{G'}(X)| < |X|$. Without loss of generality, we can assume that if X contains some copy of a vertex v , then it contains all the copies of v , since including all the remaining copies increases $|X|$ but

does not change $|N_{G'}(X)|$. Hence, the set X in A' naturally corresponds to the set X_A of size $|X|/q$ in A , the set of vertices whose copies are in X . But then $|N_G(X_A)| = |N_{G'}(X)| < |X| = q|X_A|$, which is a contradiction. Hence A' has a matching into B and thus A has a q -expansion into B .

For the algorithmic claim, note that, if there is no q -expansion from A into B , then we can use Theorem 2.13 to find a set $X \subseteq A'$ such that $|N_{G'}(X)| < |X|$, and the corresponding set X_A satisfies $|N_G(X_A)| < q|X_A|$. \square

Finally, we are ready to prove a lemma analogous to Lemma 2.14.

Lemma 2.18. (Expansion lemma) *Let $q \geq 1$ be a positive integer and G be a bipartite graph with vertex bipartition (A, B) such that*

- (i) $|B| \geq q|A|$, and
- (ii) there are no isolated vertices in B .

Then there exist nonempty vertex sets $X \subseteq A$ and $Y \subseteq B$ such that

- there is a q -expansion of X into Y , and
- no vertex in Y has a neighbor outside X , that is, $N(Y) \subseteq X$.

Furthermore, the sets X and Y can be found in time polynomial in the size of G .

Note that the sets X , Y and $V(G) \setminus (X \cup Y)$ form a crown decomposition of G with a stronger property — every vertex of X is not only matched into Y , but there is a q -expansion of X into Y . We proceed with the proof of expansion lemma.

Proof. We proceed recursively, at every step decreasing the cardinality of A . When $|A| = 1$, the claim holds trivially by taking $X = A$ and $Y = B$.

We apply Lemma 2.17 to G . If A has a q -expansion into B , then we are done as we may again take $X = A$ and $Y = B$. Otherwise, we can in polynomial time find a (nonempty) set $Z \subseteq A$ such that $|N(Z)| < q|Z|$. We construct the graph G' by removing Z and $N(Z)$ from G . We claim that G' satisfies the assumptions of the lemma. Indeed, because we removed less than q times more vertices from B than from A , we have that (i) holds for G' . Moreover, every vertex from $B \setminus N(Z)$ has no neighbor in Z , and thus (ii) also holds for G' . Note that $Z \neq A$, because otherwise $N(A) = B$ (there are no isolated vertices in B) and $|B| \geq q|A|$. Hence, we recurse on the graph G' with bipartition $(A \setminus Z, B \setminus N(Z))$, obtaining nonempty sets $X \subseteq A \setminus Z$ and $Y \subseteq B \setminus N(Z)$ such that there is a q -expansion of X into Y and such that $N_{G'}(Y) \subseteq X$. Because $Y \subseteq B \setminus N(Z)$, we have that no vertex in Y has a neighbor in Z . Hence, $N_{G'}(Y) = N_G(Y) \subseteq X$ and the pair (X, Y) satisfies all the required properties. \square

The expansion lemma is useful when the matching saturating the head part H in the crown lemma turns out to be not sufficient for a reduction, and we would like to have a few vertices of the crown C matched to a single vertex of the head H . For example, this is the case for the FEEDBACK VERTEX SET kernel presented in Section 9.1, where we need the case $q = 2$.

2.5 Kernels based on linear programming

In this section we design a $2k$ -vertex kernel for VERTEX COVER exploiting the solution to a linear programming formulation of VERTEX COVER.

Many combinatorial problems can be expressed in the language of INTEGER LINEAR PROGRAMMING (ILP). In an INTEGER LINEAR PROGRAMMING instance, we are given a set of integer-valued variables, a set of linear inequalities (called *constraints*) and a linear *cost function*. The goal is to find an (integer) evaluation of the variables that satisfies all constraints, and minimizes or maximizes the value of the cost function.

Let us give an example on how to encode a VERTEX COVER instance (G, k) as an INTEGER LINEAR PROGRAMMING instance. We introduce $n = |V(G)|$ variables, one variable x_v for each vertex $v \in V(G)$. Setting variable x_v to 1 means that v is in the vertex cover, while setting x_v to 0 means that v is not in the vertex cover. To ensure that every edge is covered, we can introduce constraints $x_u + x_v \geq 1$ for every edge $uv \in E(G)$. The size of the vertex cover is given by $\sum_{v \in V(G)} x_v$. In the end, we obtain the following ILP formulation:

$$\begin{aligned} & \text{minimize} && \sum_{v \in V(G)} x_v \\ & \text{subject to} && x_u + x_v \geq 1 \quad \text{for every } uv \in E(G), \\ & && 0 \leq x_v \leq 1 \quad \text{for every } v \in V(G), \\ & && x_v \in \mathbb{Z} \quad \text{for every } v \in V(G). \end{aligned} \tag{2.1}$$

Clearly, the optimal value of (2.1) is at most k if and only if G has a vertex cover of size at most k .

As we have just seen, INTEGER LINEAR PROGRAMMING is at least as hard as VERTEX COVER, so we do not expect it to be polynomial-time solvable. In fact, it is relatively easy to express many NP-hard problems in the language of INTEGER LINEAR PROGRAMMING. In Section 6.2 we discuss FPT algorithms for INTEGER LINEAR PROGRAMMING and their application in proving fixed-parameter tractability of other problems.

Here, we proceed in a different way: we relax the integrality requirement of INTEGER LINEAR PROGRAMMING, which is the main source of the hardness of this problem, to obtain LINEAR PROGRAMMING. That is, in LINEAR

PROGRAMMING the instance looks exactly the same as in INTEGER LINEAR PROGRAMMING, but the variables are allowed to take arbitrary real values, instead of just integers.

In the case of VERTEX COVER, we relax (2.1) by dropping the constraint $x_v \in \mathbb{Z}$ for every $v \in V(G)$. In other words, we obtain the following LINEAR PROGRAMMING instance. For a graph G , we call this relaxation LPVC(G).

$$\begin{aligned} & \text{minimize} && \sum_{v \in V(G)} x_v \\ & \text{subject to} && x_u + x_v \geq 1 \quad \text{for every } uv \in E(G), \\ & && 0 \leq x_v \leq 1 \quad \text{for every } v \in V(G). \end{aligned} \tag{2.2}$$

Note that constraints $x_v \leq 1$ can be omitted because every optimal solution of LPVC(G) satisfies these constraints.

Observe that in LPVC(G), a variable x_v can take fractional values in the interval $[0, 1]$, which corresponds to taking “part of the vertex v ” into a vertex cover. Consider an example of G being a triangle. A minimum vertex cover of a triangle is of size 2, whereas in LPVC(G) we can take $x_v = \frac{1}{2}$ for every $v \in V(G)$, obtaining a feasible solution of cost $\frac{3}{2}$. Thus, LPVC(G) does not express exactly the VERTEX COVER problem on graph G , but its optimum solution can still be useful to learn something about minimum vertex covers in G .

The main source of utility of LINEAR PROGRAMMING comes from the fact that LINEAR PROGRAMMING can be solved in polynomial time, even in some general cases where there are exponentially many constraints, accessed through an oracle. For this reason, LINEAR PROGRAMMING has found abundant applications in approximation algorithms (for more on this topic, we refer to the book of Vazirani [27]). In this section, we use LP to design a small kernel for VERTEX COVER. In Section 3.4, we will use LPVC(G) to obtain an FPT branching algorithm for VERTEX COVER.

Let us now have a closer look at the relaxation LPVC(G). Fix an optimal solution $(x_v)_{v \in V(G)}$ of LPVC(G). In this solution the variables corresponding to vertices of G take values in the interval $[0, 1]$. We partition $V(G)$ according to these values into three sets as follows.

- $V_0 = \{v \in V(G) : x_v < \frac{1}{2}\}$,
- $V_{\frac{1}{2}} = \{v \in V(G) : x_v = \frac{1}{2}\}$,
- $V_1 = \{v \in V(G) : x_v > \frac{1}{2}\}$.

Theorem 2.19 (Nemhauser-Trotter theorem). *There is a minimum vertex cover S of G such that*

$$V_1 \subseteq S \subseteq V_1 \cup V_{\frac{1}{2}}.$$

Proof. Let $S^* \subseteq V(G)$ be a minimum vertex cover of G . Define

$$S = (S^* \setminus V_0) \cup V_1.$$

By the constraints of (2.2), every vertex of V_0 can have a neighbor only in V_1 and thus S is also a vertex cover of G . Moreover, $V_1 \subseteq S \subseteq V_1 \cup V_{\frac{1}{2}}$. It suffices to show that S is a *minimum* vertex cover. Assume the contrary, i.e., $|S| > |S^*|$. Since $|S| = |S^*| - |V_0 \cap S^*| + |V_1 \setminus S^*|$ we infer that

$$|V_0 \cap S^*| < |V_1 \setminus S^*|. \quad (2.3)$$

Let us define

$$\varepsilon = \min\{|x_v - \frac{1}{2}| : v \in V_0 \cup V_1\}.$$

We decrease the fractional values of vertices from $V_1 \setminus S^*$ by ε and increase the values of vertices from $V_0 \cap S^*$ by ε . In other words, we define a vector $(y_v)_{v \in V(G)}$ as

$$y_v = \begin{cases} x_v - \varepsilon & \text{if } v \in V_1 \setminus S^*, \\ x_v + \varepsilon & \text{if } v \in V_0 \cap S^*, \\ x_v & \text{otherwise.} \end{cases}$$

Note that $\varepsilon > 0$, because otherwise $V_0 = V_1 = \emptyset$, a contradiction with (2.3). This, together with (2.3), implies that

$$\sum_{v \in V(G)} y_v < \sum_{v \in V(G)} x_v. \quad (2.4)$$

Now we show that $(y_v)_{v \in V(G)}$ is a feasible solution, i.e., it satisfies the constraints of LPVC(G). Since $(x_v)_{v \in V(G)}$ is a feasible solution, by the definition of ε we get $0 \leq y_v \leq 1$ for every $v \in V(G)$. Consider an arbitrary edge $uv \in E(G)$. If none of the endpoints of uv belong to $V_1 \setminus S^*$, then both $y_u \geq x_u$ and $y_v \geq x_v$, so $y_u + y_v \geq x_u + x_v \geq 1$. Otherwise, by symmetry we can assume that $u \in V_1 \setminus S^*$, and hence $y_u = x_u - \varepsilon$. Because S^* is a vertex cover, we have that $v \in S^*$. If $v \in V_0 \cap S^*$, then

$$y_u + y_v = x_u - \varepsilon + x_v + \varepsilon = x_u + x_v \geq 1.$$

Otherwise, $v \in (V_{\frac{1}{2}} \cup V_1) \cap S^*$. Then $y_v \geq x_v \geq \frac{1}{2}$. Note also that $x_u - \varepsilon \geq \frac{1}{2}$ by the definition of ε . It follows that

$$y_u + y_v = x_u - \varepsilon + y_v \geq \frac{1}{2} + \frac{1}{2} = 1.$$

Thus $(y_v)_{v \in V(G)}$ is a feasible solution of LPVC(G) and hence (2.4) contradicts the optimality of $(x_v)_{v \in V(G)}$. \square

Theorem 2.19 allows us to use the following reduction rule.

Reduction VC.4. Let $(x_v)_{v \in V(G)}$ be an optimum solution to LPVC(G) in a VERTEX COVER instance (G, k) and let V_0 , V_1 and $V_{\frac{1}{2}}$ be defined as above. If $\sum_{v \in V(G)} x_v > k$, then conclude that we are dealing with a no-instance. Otherwise, greedily take into the vertex cover the vertices of V_1 . That is, delete all vertices of $V_0 \cup V_1$, and decrease k by $|V_1|$.

Let us now formally verify the safeness of Reduction [VC.4](#).

Lemma 2.20. *Reduction [VC.4](#) is safe.*

Proof. Clearly, if (G, k) is a yes-instance, then an optimum solution to $\text{LPVC}(G)$ is of cost at most k . This proves the correctness of the step if we conclude that (G, k) is a no-instance.

Let $G' = G - (V_0 \cup V_1) = G[V_{\frac{1}{2}}]$ and $k' = k - |V_1|$. We claim that (G, k) is a yes-instance of VERTEX COVER if and only if (G', k') is. By Theorem [2.19](#), we know that G has a vertex cover S of size at most k such that $V_1 \subseteq S \subseteq V_1 \cup V_{\frac{1}{2}}$. Then $S' = S \cap V_{\frac{1}{2}}$ is a vertex cover in G' and the size of S' is at most $k - |V_1| = k'$.

For the opposite direction, let S' be a vertex cover in G' . For every solution of $\text{LPVC}(G)$, every edge with an endpoint from V_0 should have an endpoint in V_1 . Hence, $S = S' \cup V_1$ is a vertex cover in G and the size of this vertex cover is at most $k' + |V_1| = k$. \square

Reduction [VC.4](#) leads to the following kernel for VERTEX COVER.

Theorem 2.21. *VERTEX COVER admits a kernel with at most $2k$ vertices.*

Proof. Let (G, k) be an instance of VERTEX COVER. We solve $\text{LPVC}(G)$ in polynomial time, and apply Reduction [VC.4](#) to the obtained solution $(x_v)_{v \in V(G)}$, either concluding that we are dealing with a no-instance or obtaining an instance (G', k') . Lemma [2.20](#) guarantees the safeness of the reduction. For the size bound, observe that

$$|V(G')| = |V_{\frac{1}{2}}| = \sum_{v \in V_{\frac{1}{2}}} 2x_v \leq 2 \sum_{v \in V(G)} x_v \leq 2k.$$

\square

While it is possible to solve linear programs in polynomial time, usually such solutions are less efficient than combinatorial algorithms. The specific structure of the LP-relaxation of the vertex cover problem [\(2.2\)](#) allows us to solve it by reducing to the problem of finding a maximum-size matching in a bipartite graph.

Lemma 2.22. *For a graph G with n vertices and m edges, the optimal (fractional) solution to the linear program $\text{LPVC}(G)$ can be found in time $\mathcal{O}(m\sqrt{n})$.*

Proof. We reduce the problem of solving $\text{LPVC}(G)$ to a problem of finding a minimum-size vertex cover in the following bipartite graph H . Its vertex set consists of two copies V_1 and V_2 of the vertex set of G . Thus, every vertex $v \in V(G)$ has two copies $v_1 \in V_1$ and $v_2 \in V_2$ in H . For every edge $uv \in E(H)$, we have edges u_1v_2 and v_1u_2 in H .

Using the Hopcroft-Karp algorithm (Theorem 2.13), we can find a minimum vertex cover S of H in time $\mathcal{O}(m\sqrt{n})$. We define a vector $(x_v)_{v \in V(G)}$ as follows: if both vertices v_1 and v_2 are in S , then $x_v = 1$. If exactly one of the vertices v_1 and v_2 is in S , we put $x_v = \frac{1}{2}$. We put $x_v = 0$ if none of the vertices v_1 and v_2 are in S . Thus

$$\sum_{v \in V(G)} x_v = \frac{|S|}{2}.$$

Since S is a vertex cover in H , we have that for every edge $uv \in E(G)$ at least two vertices from $\{u_1, u_2, v_1, v_2\}$ should be in S . Thus $x_u + x_v \geq 1$ and vector $(x_v)_{v \in V(G)}$ satisfies the constraints of LPVC(G).

To show that $(x_v)_{v \in V(G)}$ is an optimal solution of LPVC(G), we argue as follows. Let $(y_v)_{v \in V(G)}$ be an optimal solution of LPVC(G). For every vertex v_i , $i \in \{1, 2\}$, of H , we assign the weight $\mathbf{w}(v_i) = y_v$. This weight assignment is a fractional vertex cover of H , i.e., for every edge $v_1 u_2 \in E(H)$, $\mathbf{w}(v_1) + \mathbf{w}(u_2) \geq 1$. We have that

$$\sum_{v \in V(G)} y_v = \frac{1}{2} \sum_{v \in V(G)} (\mathbf{w}(v_1) + \mathbf{w}(v_2)).$$

On the other hand, the value $\sum_{v \in V(H)} \mathbf{w}(v)$ of any fractional solution of LPVC(H) is at least the size of a maximum matching M in H . A reader familiar with linear programming can see that this follows from weak duality; we also ask you to verify this fact in Exercise 2.24.

By König's theorem (Theorem 2.11), $|M| = |S|$. Hence

$$\sum_{v \in V(G)} y_v = \frac{1}{2} \sum_{v \in V(G)} (\mathbf{w}(v_1) + \mathbf{w}(v_2)) = \frac{1}{2} \sum_{v \in V(H)} \mathbf{w}(v) \geq \frac{|S|}{2} = \sum_{v \in V(G)} x_v.$$

Thus $(x_v)_{v \in V(G)}$ is an optimal solution of LPVC(G). \square

We immediately obtain the following.

Corollary 2.23. *For a graph G with n vertices and m edges, the kernel of Theorem 2.21 can be found in time $\mathcal{O}(m\sqrt{n})$.*

The following proposition is another interesting consequence of the proof of Lemma 2.22.

Proposition 2.24. *Let G be a graph on n vertices and m edges. Then LPVC(G) has a half-integral optimal solution, i.e., all variables have values in the set $\{0, \frac{1}{2}, 1\}$. Furthermore, we can find a half-integral optimal solution in time $\mathcal{O}(m\sqrt{n})$.*

In short, we have proved properties of $\text{LPVC}(G)$. There exists a half-integral optimal solution $(x_v)_{v \in V(G)}$ to $\text{LPVC}(G)$, and it can be found efficiently. We can look at this solution as a partition of $V(G)$ into parts V_0 , $V_{\frac{1}{2}}$, and V_1 with the following message: greedily take V_1 into a solution, do not take any vertex of V_0 into a solution, and in $V_{\frac{1}{2}}$, we do not know what to do and that is the hard part of the problem. However, as an optimum solution pays $\frac{1}{2}$ for every vertex of $V_{\frac{1}{2}}$, the hard part — the kernel of the problem — cannot have more than $2k$ vertices.

2.6 Sunflower lemma

In this section we introduce a classical result of Erdős and Rado and show some of its applications in kernelization. In the literature it is known as the sunflower lemma or as the Erdős-Rado lemma. We first define the terminology used in the statement of the lemma. A *sunflower* with k *petals* and a *core* Y is a collection of sets S_1, \dots, S_k such that $S_i \cap S_j = Y$ for all $i \neq j$; the sets $S_i \setminus Y$ are petals and we require *none of them to be empty*. Note that a family of pairwise disjoint sets is a sunflower (with an empty core).

Theorem 2.25 (Sunflower lemma). *Let \mathcal{A} be a family of sets (without duplicates) over a universe U , such that each set in \mathcal{A} has cardinality exactly d . If $|\mathcal{A}| > d!(k-1)^d$, then \mathcal{A} contains a sunflower with k petals and such a sunflower can be computed in time polynomial in $|\mathcal{A}|$, $|U|$, and k .*

Proof. We prove the theorem by induction on d . For $d = 1$, i.e., for a family of singletons, the statement trivially holds. Let $d \geq 2$ and let \mathcal{A} be a family of sets of cardinality at most d over a universe U such that $|\mathcal{A}| > d!(k-1)^d$.

Let $\mathcal{G} = \{S_1, \dots, S_\ell\} \subseteq \mathcal{A}$ be an inclusion-wise maximal family of pairwise disjoint sets in \mathcal{A} . If $\ell \geq k$ then \mathcal{G} is a sunflower with at least k petals. Thus we assume that $\ell < k$. Let $S = \bigcup_{i=1}^{\ell} S_i$. Then $|S| \leq d(k-1)$. Because \mathcal{G} is maximal, every set $A \in \mathcal{A}$ intersects at least one set from \mathcal{G} , i.e., $A \cap S \neq \emptyset$. Therefore, there is an element $u \in U$ contained in at least

$$\frac{|\mathcal{A}|}{|S|} > \frac{d!(k-1)^d}{d(k-1)} = (d-1)!(k-1)^{d-1}$$

sets from \mathcal{A} . We take all sets of \mathcal{A} containing such an element u , and construct a family \mathcal{A}' of sets of cardinality $d-1$ by removing from each set the element u . Because $|\mathcal{A}'| > (d-1)!(k-1)^{d-1}$, by the induction hypothesis, \mathcal{A}' contains a sunflower $\{S'_1, \dots, S'_k\}$ with k petals. Then $\{S'_1 \cup \{u\}, \dots, S'_k \cup \{u\}\}$ is a sunflower in \mathcal{A} with k petals.

The proof can be easily transformed into a polynomial-time algorithm, as follows. Greedily select a maximal set of pairwise disjoint sets. If the size

of this set is at least k , then return this set. Otherwise, find an element u contained in the maximum number of sets in \mathcal{A} , and call the algorithm recursively on sets of cardinality $d - 1$, obtained from deleting u from the sets containing u . \square

2.6.1 d -HITTING SET

As an application of the sunflower lemma, we give a kernel for d -HITTING SET. In this problem, we are given a family \mathcal{A} of sets over a universe U , where each set in the family has cardinality at most d , and a positive integer k . The objective is to decide whether there is a subset $H \subseteq U$ of size at most k such that H contains at least one element from each set in \mathcal{A} .

Theorem 2.26. *d -HITTING SET admits a kernel with at most $d!k^d$ sets and at most $d!k^d \cdot d^2$ elements.*

Proof. The crucial observation is that if \mathcal{A} contains a sunflower

$$S = \{S_1, \dots, S_{k+1}\}$$

of cardinality $k + 1$, then every hitting set H of \mathcal{A} of cardinality at most k intersects the core Y of the sunflower S . Indeed, if H does not intersect Y , it should intersect each of the $k + 1$ disjoint petals $S_i \setminus Y$. This leads to the following reduction rule.

Reduction HS.1. Let (U, \mathcal{A}, k) be an instance of d -HITTING SET and assume that \mathcal{A} contains a sunflower $S = \{S_1, \dots, S_{k+1}\}$ of cardinality $k + 1$ with core Y . Then return (U', \mathcal{A}', k) , where $\mathcal{A}' = (\mathcal{A} \setminus S) \cup \{Y\}$ is obtained from \mathcal{A} by deleting all sets $\{S_1, \dots, S_{k+1}\}$ and by adding a new set Y and $U' = \bigcup_{X \in \mathcal{A}'} X$.

Note that when deleting sets we do not delete the elements contained in these sets but only those which do not belong to any set. Then the instances (U, \mathcal{A}, k) and (U', \mathcal{A}', k) are equivalent, i.e. (U, \mathcal{A}) contains a hitting set of size k if and only if (U', \mathcal{A}') does.

The kernelization algorithm is as follows. If for some $d' \in \{1, \dots, d\}$ the number of sets in \mathcal{A} of size exactly d' is more than $d'!k^{d'}$, then the kernelization algorithm applies the sunflower lemma to find a sunflower of size $k + 1$, and applies Reduction HS.1 on this sunflower. It applies this procedure exhaustively, and obtains a new family of sets \mathcal{A}' of size at most $d!k^d \cdot d$. If $\emptyset \in \mathcal{A}'$ (that is, at some point a sunflower with an empty core has been discovered), then the algorithm concludes that there is no hitting set of size at most k and returns that the given instance is a no-instance. Otherwise, every set contains at most d elements, and thus the number of elements in the kernel is at most $d!k^d \cdot d^2$. \square

Exercises

2.1 (E). Prove Lemma 2.5. A digraph is acyclic if and only if it is possible to order its vertices in such a way such that for every arc (u, v) , we have $u < v$.

2.2 (E). Give an example of a feedback arc set F in a tournament G , such that $G \circledast F$ is not acyclic.

2.3 (E). Show that Reductions [ECC.1], [ECC.2], and [ECC.3] are safe.

2.4 (E). In the POINT LINE COVER problem, we are given a set of n points in the plane and an integer k , and the goal is to check if there exists a set of k lines on the plane that contain all the input points. Show a kernel for this problem with $\mathcal{O}(k^2)$ points.

2.5. A graph G is a *cluster graph* if every connected component of G is a clique. In the CLUSTER EDITING problem, we are given as input a graph G and an integer k , and the objective is to check whether one can edit (add or delete) at most k edges in G to obtain a cluster graph. That is, we look for a set $F \subseteq \binom{V(G)}{2}$ of size at most k , such that the graph $(V(G), (E(G) \setminus F) \cup (F \setminus E(G)))$ is a cluster graph.

1. Show that a graph G is a cluster graph if and only if it does not have an induced path on three vertices (sequence of three vertices u, v, w such that uv and vw are edges and $uw \notin E(G)$).
2. Show a kernel for CLUSTER EDITING with $\mathcal{O}(k^2)$ vertices.

2.6. In the SET SPLITTING problem, we are given a family of sets \mathcal{F} over a universe U and a positive integer k , and the goal is to test whether there exists a coloring of U with two colors such that at least k sets in \mathcal{F} are nonmonochromatic (that is, they contain vertices of both colors). Show that the problem admits a kernel with at most $2k$ sets and $\mathcal{O}(k^2)$ universe size.

2.7. In the MINIMUM MAXIMAL MATCHING problem, we are given an undirected graph G and a positive integer k , and the objective is to decide whether there exists a maximal matching in G on at most k edges. Obtain a polynomial kernel for the problem (parameterized by k).

2.8. In the MIN-ONES-2-SAT problem, we are given a 2-CNF formula ϕ and an integer k , and the objective is to decide whether there exists a satisfying assignment for ϕ with at most k variables set to true. Show that MIN-ONES-2-SAT admits a polynomial kernel.

2.9. In the *d*-BOUNDED-DEGREE DELETION problem, we are given an undirected graph G and a positive integer k , and the task is to find at most k vertices whose removal decreases the maximum vertex degree of the graph to at most d . Obtain a kernel of size polynomial in k and d for the problem. (Observe that VERTEX COVER is the case of $d = 0$.)

2.10. Show a kernel with $\mathcal{O}(k^2)$ vertices for the following problem: given a graph G and an integer k , check if G contains a subgraph with exactly k edges, whose vertices are all of odd degree in the subgraph.

2.11. A set of vertices D in an undirected graph G is called a *dominating set* if $N[D] = V(G)$. In the DOMINATING SET problem, we are given an undirected graph G and a positive integer k , and the objective is to test whether there exists a dominating set of size at most k . Show that DOMINATING SET admits a polynomial kernel on graphs where the length of the shortest cycle is at least 5. (What would you do with vertices with degree more than k ? Note that unlike for the VERTEX COVER problem, you cannot delete a vertex once you pick it in the solution.)

2.12. Show that FEEDBACK VERTEX SET admits a kernel with $\mathcal{O}(k)$ vertices on undirected regular graphs.

2.13. We say that an n -vertex digraph is *well-spread* if every vertex has indegree at least \sqrt{n} . Show that DIRECTED FEEDBACK ARC SET, restricted to well-spread digraphs, is FPT by obtaining a polynomial kernel for this problem. Does the problem remain FPT if we replace the lower bound on indegree by any monotonically increasing function of n (like $\log n$ or $\log \log \log n$)? Does the assertion hold if we replace indegree with outdegree? What about DIRECTED FEEDBACK VERTEX SET?

2.14. In the CONNECTED VERTEX COVER problem, we are given an undirected graph G and a positive integer k , and the objective is to decide whether there exists a vertex cover C of G such that $|C| \leq k$ and $G[C]$ is connected.

1. Explain where the kernelization procedure described in Theorem 2.4 for VERTEX COVER breaks down for the CONNECTED VERTEX COVER problem.
2. Show that the problem admits a kernel with at most $2^k + \mathcal{O}(k^2)$ vertices.
3. Show that if the input graph G does not contain a cycle of length 4 as a subgraph, then the problem admits a kernel with at most $\mathcal{O}(k^2)$ vertices.

2.15 (2). Extend the argument of the previous exercise to show that, for every fixed $d \geq 2$, CONNECTED VERTEX COVER admits a kernel of size $\mathcal{O}(k^d)$ if restricted to graphs that do not contain the biclique $K_{d,d}$ as a subgraph.

2.16 (2). A graph G is chordal if it contains no induced cycles of length more than 3, that is, every cycle of length at least 4 has a chord. In the CHORDAL COMPLETION problem, we are given an undirected graph G and a positive integer k , and the objective is to decide whether we can add at most k edges to G so that it becomes a chordal graph. Obtain a polynomial kernel for CHORDAL COMPLETION (parameterized by k).

2.17 (2). In the EDGE DISJOINT CYCLE PACKING problem, we are given an undirected graph G and a positive integer k , and the objective is to test whether G has k pairwise edge disjoint cycles. Obtain a polynomial kernel for EDGE DISJOINT CYCLE PACKING (parameterized by k).

2.18 (2). A *bisection* of a graph G with an even number of vertices is a partition of $V(G)$ into V_1 and V_2 such that $|V_1| = |V_2|$. The size of (V_1, V_2) is the number of edges with one endpoint in V_1 and the other in V_2 . In the MAXIMUM BISECTION problem, we are given an undirected graph G with an even number of vertices and a positive integer k , and the objective is to test whether there exists a bisection of size at least k .

1. Show that every graph with m edges has a bisection of size at least $\lceil \frac{m}{2} \rceil$. Use this to show that MAXIMUM BISECTION admits a kernel with $2k$ edges.
2. Consider the following “above guarantee” variant of MAXIMUM BISECTION, where we are given an undirected graph G and a positive integer k , but the objective is to test whether there exists a bisection of size at least $\lceil \frac{m}{2} \rceil + k$. Show that the problem admits a kernel with $\mathcal{O}(k^2)$ vertices and $\mathcal{O}(k^3)$ edges.

2.19 (2). Byteland, a country of area exactly n square miles, has been divided by the government into n regions, each of area exactly one square mile. Meanwhile, the army of Byteland divided its area into n military zones, each of area again exactly one square mile. Show that one can build n airports in Byteland, such that each region and each military zone contains one airport.

2.20. A magician and his assistant are performing the following magic trick. A volunteer from the audience picks five cards from a standard deck of 52 cards and then passes the deck to the assistant. The assistant shows to the magician, one by one in some order, four cards from the chosen set of five cards. Then, the magician guesses the remaining fifth card. Show that this magic trick can be performed without any help of magic.

2.21. Prove the second claim of Theorem 2.13.

2.22. In the DUAL-COLORING problem, we are given an undirected graph G on n vertices and a positive integer k , and the objective is to test whether there exists a proper coloring of G with at most $n - k$ colors. Obtain a kernel with $\mathcal{O}(k)$ vertices for this problem using crown decomposition.

2.23 (✉). In the MAX-INTERNAL SPANNING TREE problem, we are given an undirected graph G and a positive integer k , and the objective is to test whether there exists a spanning tree with at least k internal vertices. Obtain a kernel with $\mathcal{O}(k)$ vertices for MAX-INTERNAL SPANNING TREE.

2.24 (✉). Let G be an undirected graph, let $(x_v)_{v \in V(G)}$ be any feasible solution to LPVC(G), and let M be a matching in G . Prove that $|M| \leq \sum_{v \in V(G)} x_v$.

2.25 (✉). Let G be a graph and let $(x_v)_{v \in V(G)}$ be an optimum solution to LPVC(G) (not necessarily a half-integral one). Define a vector $(y_v)_{v \in V(G)}$ as follows:

$$y_v = \begin{cases} 0 & \text{if } x_v < \frac{1}{2} \\ \frac{1}{2} & \text{if } x_v = \frac{1}{2} \\ 1 & \text{if } x_v > \frac{1}{2}. \end{cases}$$

Show that $(y_v)_{v \in V(G)}$ is also an optimum solution to LPVC(G).

2.26 (✉). In the MIN-ONES-2-SAT, we are given a CNF formula, where every clause has exactly two literals, and an integer k , and the goal is to check if there exists a satisfying assignment of the input formula with at most k variables set to true. Show a kernel for this problem with at most $2k$ variables.

2.27 (✉). Consider a restriction of d -HITTING SET, called Ed -HITTING SET, where we require every set in the input family \mathcal{A} to be of size *exactly* d . Show that this problem is not easier than the original d -HITTING SET problem, by showing how to transform a d -HITTING SET instance into an equivalent Ed -HITTING SET instance without changing the number of sets.

2.28. Show a kernel with at most $f(d)k^d$ sets (for some computable function f) for the Ed -HITTING SET problem, defined in the previous exercise.

2.29. In the d -SET PACKING problem, we are given a family \mathcal{A} of sets over a universe U , where each set in the family has cardinality at most d , and a positive integer k . The objective is to decide whether there are sets $S_1, \dots, S_k \in \mathcal{A}$ that are pairwise disjoint. Use the sunflower lemma to obtain a kernel for d -SET PACKING with $f(d)k^d$ sets, for some computable function d .

2.30. Consider a restriction of d -SET PACKING, called Ed -SET PACKING, where we require every set in the input family \mathcal{A} to be of size *exactly* d . Show that this problem is not easier than the original d -SET PACKING problem, by showing how to transform a d -SET PACKING instance into an equivalent Ed -SET PACKING instance without changing the number of sets.

2.31. A *split graph* is a graph in which the vertices can be partitioned into a clique and an independent set. In the VERTEX DISJOINT PATHS problem, we are given an undirected graph G and k pairs of vertices (s_i, t_i) , $i \in \{1, \dots, k\}$, and the objective is to decide whether there exists paths P_i joining s_i to t_i such that these paths are pairwise vertex disjoint. Show that VERTEX DISJOINT PATHS admits a polynomial kernel on split graphs (when parameterized by k).

2.32. Consider now the VERTEX DISJOINT PATHS problem, defined in the previous exercise, restricted, for a fixed integer $d \geq 3$, to a class of graphs that does not contain a d -vertex path as an induced subgraph. Show that in this class the VERTEX DISJOINT PATHS problem admits a kernel with $\mathcal{O}(k^{d-1})$ vertices and edges.

2.33. In the CLUSTER VERTEX DELETION problem, we are given as input a graph G and a positive integer k , and the objective is to check whether there exists a set $S \subseteq V(G)$ of size at most k such that $G - S$ is a cluster graph. Show a kernel for CLUSTER VERTEX DELETION with $\mathcal{O}(k^3)$ vertices.

2.34. An undirected graph G is called *perfect* if for every induced subgraph H of G , the size of the largest clique in H is same as the chromatic number of H . In the ODD CYCLE TRANSVERSAL problem, we are given an undirected graph G and a positive integer k , and the objective is to find at most k vertices whose removal makes the resulting graph bipartite. Obtain a kernel with $\mathcal{O}(k^2)$ vertices for ODD CYCLE TRANSVERSAL on perfect graphs.

2.35. In the SPLIT VERTEX DELETION problem, we are given an undirected graph G and a positive integer k and the objective is to test whether there exists a set $S \subseteq V(G)$ of size at most k such that $G - S$ is a split graph (see Exercise 2.31 for the definition).

1. Show that a graph is split if and only if it has no induced subgraph isomorphic to one of the following three graphs: a cycle on four or five vertices, or a pair of disjoint edges.
2. Give a kernel with $\mathcal{O}(k^5)$ vertices for SPLIT VERTEX DELETION.

2.36 (2). In the SPLIT EDGE DELETION problem, we are given an undirected graph G and a positive integer k , and the objective is to test whether G can be transformed into a split graph by deleting at most k edges. Obtain a polynomial kernel for this problem (parameterized by k).

2.37 (2). In the RAMSEY problem, we are given as input a graph G and an integer k , and the objective is to test whether there exists in G an independent set or a clique of size at least k . Show that RAMSEY is FPT.

2.38 (2). A directed graph D is called *oriented* if there is no directed cycle of length at most 2. Show that the problem of testing whether an oriented digraph contains an induced directed acyclic subgraph on at least k vertices is FPT.

Hints

2.4 Consider the following reduction rule: if there exists a line that contains more than k input points, delete the points on this line and decrease k by 1.

2.5 Consider the following natural reduction rules:

1. delete a vertex that is not a part of any P_3 (induced path on three vertices);
2. if an edge uv is contained in at least $k + 1$ different P_3 s, then delete uv ;
3. if a non-edge uv is contained in at least $k + 1$ different P_3 s, then add uv .

Show that, after exhaustive application of these rules, a yes-instance has $\mathcal{O}(k^2)$ vertices.

2.6 First, observe that one can discard any set in \mathcal{F} that is of size at most 1. Second, observe that if every set in \mathcal{F} is of size at least 2, then a random coloring of U has at least

$|\mathcal{F}|/2$ nonmonochromatic sets on average, and an instance with $|\mathcal{F}| \geq 2k$ is a yes-instance. Moreover, observe that if we are dealing with a yes-instance and $F \in \mathcal{F}$ is of size at least $2k$, then we can always tweak the solution coloring to color F nonmonochromatically: fix two differently colored vertices for $k - 1$ nonmonochromatic sets in the solution, and color some two uncolored vertices of F with different colors. Use this observation to design a reduction rule that handles large sets in \mathcal{F} .

2.7 Observe that the endpoints of the matching in question form a vertex cover of the input graph. In particular, every vertex of degree larger than $2k$ needs to be an endpoint of a solution matching. Let X be the set of these large-degree vertices. Argue, similarly as in the case of $\mathcal{O}(k^2)$ kernel for VERTEX COVER, that in a yes-instance, $G \setminus X$ has only few edges. Design a reduction rule to reduce the number of isolated vertices of $G \setminus X$.

2.8 Proceed similarly as in the $\mathcal{O}(k^2)$ kernel for VERTEX COVER.

2.9 Proceed similarly as in the case of VERTEX COVER. Argue that the vertices of degree larger than $d + k$ need to be included in the solution. Moreover, observe that you may delete isolated vertices, as well as edges connecting two vertices of degree at most d . Argue that, if no rule is applicable, then a yes-instance is of size bounded polynomially in $d + k$.

2.10 The important observation is that a matching of size k is a good subgraph. Hence, we may restrict ourselves to the case where we are additionally given a vertex cover X of the input graph of size at most $2k$. Moreover, assume that X is inclusion-wise minimal. To conclude, prove that, if a vertex $v \in X$ has at least k neighbors in $V(G) \setminus X$, then (G, k) is a yes-instance.

2.11 The main observation is that, since there is no 3-cycle nor 4-cycle in the graph, if $x, y \in N(v)$, then only v can dominate both x and y at once. In particular, every vertex of degree larger than k needs to be included in the solution.

However, you cannot easily delete such a vertex. Instead, mark it as “obligatory” and mark its neighbors as “dominated”. Note now that you can delete a “dominated” vertex, as long as it has no unmarked neighbor and its deletion does not drop the degree of an “obligatory” vertex to k .

Prove that, in a yes-instance, if no rule is applicable, then the size is bounded polynomially in k . To this end, show that

1. any vertex can dominate at most k unmarked vertices, and, consequently, there are at most k^2 unmarked vertices;
2. there are at most k “obligatory” vertices;
3. every remaining “dominated” vertex can be charged to one unmarked or obligatory vertex in a manner that each unmarked or obligatory vertex is charged at most $k + 1$ times.

2.12 Let (G, k) be a FEEDBACK VERTEX SET instance and assume G is d -regular. If $d \leq 2$, then solve (G, k) in polynomial time. Otherwise, observe that G has $dn/2$ edges and, if (G, k) is a yes-instance and X is a feedback vertex set of G of size at most k , then at most dk edges of G are incident to X and $G - X$ contains less than $n - k$ edges (since it is a forest). Consequently, $dn/2 \leq dk + n - k$, which gives $n = \mathcal{O}(k)$ for $d \geq 3$.

2.13 Show, using greedy arguments, that if every vertex in a digraph G has indegree at least d , then G contains d pairwise edge-disjoint cycles.

For the vertex-deletion variant, design a simple reduction that boosts up the indegree of every vertex without actually changing anything in the solution space.

2.14 Let X be the set of vertices of G of degree larger than k . Clearly, any connected vertex cover of G of size at most k needs to contain X . Moreover, as in the case of VERTEX COVER, in a yes-instance there are only $\mathcal{O}(k^2)$ edges in $G - X$. However, we cannot easily discard the isolated vertices of $G - X$, as they may be used to make the solution connected.

To obtain an exponential kernel, note that in a yes-instance, $|X| \leq k$, and if we have two vertices u, v that are isolated in $G - X$, and $N_G(u) = N_G(v)$ (note that $N_G(u) \subseteq X$ for every u that is isolated in $G - X$), then we need only one of the vertices u, v in a CONNECTED VERTEX COVER solution. Hence, in a kernel, we need to keep:

1. $G[X]$, and all edges and non-isolated vertices of $G - X$;
2. for every $x \in X$, some $k + 1$ neighbors of x ;
3. for every $Y \subseteq X$, one vertex u that is isolated in $G - X$ and $N_G(u) = Y$ (if there exists any such vertex).

For the last part of the exercise, note that in the presence of this assumption, no two vertices of X share more than one neighbor and, consequently, there are only $\mathcal{O}(|X|^2)$ sets $Y \subseteq X$ for which there exist $u \notin X$ with $N_G(u) = Y$.

2.15 We repeat the argument of the previous exercise, and bound the number of sets $Y \subseteq X$ for which we need to keep a vertex $u \in V(G) \setminus X$ with $N_G(u) = Y$. First, there are $\mathcal{O}(d|X|^{d-1})$ sets Y of size smaller than d . Second, charge every set Y of size at least d to one of its subset of size d . Since G does not contain $K_{d,d}$ as a subgraph, every subset X of size d is charged less than d times. Consequently, there are at most $(d-1)\binom{|X|}{d}$ vertices $u \in V(G) \setminus X$ such that $N_G(u) \subseteq X$ and $|N_G(u)| \geq d$.

2.16 The main observation is as follows: an induced cycle of length ℓ needs exactly $\ell - 3$ edges to become chordal. In particular, if a graph contains an induced cycle of length larger than $k + 3$, then the input instance is a no-instance, as we need more than k edges to triangulate the cycle in question.

First, prove the safeness of the following two reduction rules:

1. Delete any vertex that is not contained in any induced cycle in G .
2. A vertex x is a *friend* of a non-edge uv , if u, x, v are three consecutive vertices of some induced cycle in G . If $uv \notin E(G)$ has more than $2k$ friends, then add the edge uv and decrease k by one.

Second, consider the following procedure. Initiate A to be the vertex set of any inclusion-wise maximal family of pairwise vertex-disjoint induced cycles of length at most 4 in G . Then, as long as there exists an induced cycle of length at most 4 in G that contains two consecutive vertices in $V(G) \setminus A$, move these two vertices to A . Show, using a charging argument, that, in a yes-instance, the size of A remains $\mathcal{O}(k)$. Conclude that the size of a reduced yes-instance is bounded polynomially in k .

2.17 Design reduction rules that remove vertices of degree at most 2 (you may obtain a multigraph in the process). Prove that every n -vertex multigraph of minimum degree at least 3 has a cycle of length $\mathcal{O}(\log n)$. Use this to show a greedy argument that an n -vertex multigraph of minimum degree 3 has $\Omega(n^\varepsilon)$ pairwise edge-disjoint cycles for some $\varepsilon > 0$.

2.18 Consider the following argument. Let $|V(G)| = 2n$ and pair the vertices of G arbitrarily: $V(G) = \{x_1, y_1, x_2, y_2, \dots, x_n, y_n\}$. Consider the bisection (V_1, V_2) where, in each pair (x_i, y_i) , one vertex goes to V_1 and the other goes to V_2 , where the decision is made uniformly at random and independently of other pairs. Prove that, in expectation, the obtained bisection is of size at least $(m + \ell)/2$, where ℓ is the number of pairs (x_i, y_i) where $x_i y_i \in E(G)$.

Use the arguments in the previous paragraph to show not only the first point of the exercise, but also that the input instance is a yes-instance if it admits a matching of size $2k$. If this is not the case, then let X be the set of endpoints of a maximal matching in G ; note that $|X| \leq 4k$.

First, using a variation of the argument of the first paragraph, prove that, if there exists $x \in X$ that has at least $2k$ neighbors and at least $2k$ non-neighbors outside X , then the input instance is a yes-instance. Second, show that in the absence of such a vertex, all but

$\mathcal{O}(k^2)$ vertices of $V(G) \setminus X$ have exactly the same neighborhood in X , and design a way to reduce them.

2.19 Construct the following bipartite graph: on one side there are regions of Byteland, on the second side there are military zones, and a region R is adjacent to a zone Z if $R \cap Z \neq \emptyset$. Show that this graph satisfies the condition of Hall's theorem and, consequently, contains a perfect matching.

2.20 Consider the following bipartite graph: on one side there are all $\binom{52}{5}$ sets of five cards (possibly chosen by the volunteer), and on the other side there are all $52 \cdot 51 \cdot 50 \cdot 49$ tuples of pairwise different four cards (possibly shown by the assistant). A set S is adjacent to a tuple T if all cards of T belong to S . Using Hall's theorem, show that this graph admits a matching saturating the side with all sets of five cards. This matching induces a strategy for the assistant and the magician.

We now show a relatively simple explicit strategy, so that you can impress your friends and perform this trick at some party. In every set of five cards, there are two cards of the same color, say a and b . Moreover, as there are 13 cards of the same color, the cards a and b differ by at most 6, that is, $a + i = b$ or $b + i = a$ for some $1 \leq i \leq 6$, assuming some cyclic order on the cards of the same color. Without loss of generality, assume $a + i = b$. The assistant first shows the card a to the magician. Then, using the remaining three cards, and some fixed total order on the whole deck of cards, the assistant shows the integer i (there are $3! = 6$ permutations of remaining three cards). Consequently, the magician knows the card b by knowing its color (the same as the first card show by the assistant) and the value of the card a and the number i .

2.21 Let M be a maximum matching, which you can find using the Hopcroft-Karp algorithm (the first part of Theorem 2.13). If M saturates V_1 , then we are done. Otherwise, pick any $v \in V_1 \setminus V(M)$ (i.e., a vertex $v \in V_1$ that is not an endpoint of an edge of M) and consider all vertices of G that are reachable from v using alternating paths. (A path P is *alternating* if every second edge of P belongs to M .) Show that all vertices from V_1 that are reachable from v using alternating paths form an inclusion-wise minimal set X with $|N(X)| < |X|$.

2.22 Apply the crown lemma to \bar{G} , the edge complement of G (\bar{G} has vertex set $V(G)$ and $uv \in E(\bar{G})$ if and only if $uv \notin E(G)$) and the parameter $k - 1$. If it returns a matching M_0 of size k , then note that one can color the endpoints of each edge of M_0 with the same color, obtaining a coloring of G with $n - k$ colors. Otherwise, design a way to greedily color the head and the crown of the obtained crown decomposition.

2.23 Your main tool is the following variation of the crown lemma: if $V(G)$ is sufficiently large, then you can find either a matching of size $k + 1$, or a crown decomposition $V(G) = C \cup H \cup R$, such that $G[H \cup C]$ admits a spanning tree where all vertices of H and $|H| - 1$ vertices of C are of degree at least two. Prove it, and use it for the problem in question.

2.24 Observe that for every $uv \in M$ we have $x_u + x_v \geq 1$ and, moreover, all these inequalities for all edges of M contain different variables. In other words,

$$\sum_{v \in V(G)} \mathbf{w}(x_v) \geq \sum_{v \in V(M)} \mathbf{w}(x_v) = \sum_{vu \in M} (\mathbf{w}(x_v) + \mathbf{w}(x_u)) \geq \sum_{vu \in M} 1 = |M|.$$

2.25 Let $V_\delta = \{v \in V(G) : 0 < x_v < \frac{1}{2}\}$ and $V_{1-\delta} = \{v \in V(G) : \frac{1}{2} < x_v < 1\}$. For sufficiently small $\varepsilon > 0$, consider two operations: first, an operation of adding ε to all variables x_v for $v \in V_\delta$ and subtracting ε from x_v for $v \in V_{1-\delta}$, and second, an operation of adding ε to all variables x_v for $v \in V_{1-\delta}$ and subtracting ε from x_v for $v \in V_\delta$. Show that both these operations lead to feasible solutions to LPVC(G), as long as ε is small enough. Conclude that $|V_\delta| = |V_{1-\delta}|$, and that both operations lead to other *optimal* solutions

to $\text{LPVC}(G)$. Finally, observe that, by repeatedly applying the second operation, one can empty sets V_δ and $V_{1-\delta}$, and reach the vector $(y_v)_{v \in V(G)}$.

2.26 First, design natural reduction rules that enforce the following: for every variable x in the input formula φ , there exists a truth assignment satisfying φ that sets x to false, and a truth assignment satisfying φ that sets x to true. In other words, whenever a value of some variable is fixed in any satisfying assignment, fix this value and propagate it in the formula.

Then, consider the following *closure* of the formula φ : for every two-literal clause C that is satisfied in every truth assignment satisfying φ , add C to φ . Note that testing whether C is such a clause can be done in polynomial time: force two literals in C to be false and check if φ remains satisfiable. Moreover, observe that the sets of satisfying assignments for φ and φ' are equal.

Let φ' be the closure of φ . Consider the following auxiliary graph H : $V(H)$ is the set of variables of φ' , and $xy \in E(H)$ iff the clause $x \vee y$ belongs to φ' . Clearly, if we take any truth assignment ψ satisfying φ , then $\psi^{-1}(\top)$ is a vertex cover of H . A somewhat surprising fact is that a partial converse is true: for every inclusion-wise minimal vertex cover X of H , the assignment ψ defined as $\psi(x) = \top$ if and only if $x \in X$ satisfies φ' (equivalently, φ). Note that such a claim would solve the exercise: we can apply the LP-based kernelization algorithm to VERTEX COVER instance (H, k) , and translate the reductions it makes back to the formula φ .

Below we prove the aforementioned claim in full detail. We encourage you to try to prove it on your own before reading.

Let X be a minimal vertex cover of H , and let ψ be defined as above. Take any clause C in φ' and consider three cases. If $C = x \vee y$, then $xy \in E(H)$, and, consequently, either x or y belongs to X . It follows from the definition of ψ that $\psi(x) = \top$ or $\psi(y) = \top$, and ψ satisfies C .

In the second case, $C = x \vee \neg y$. For a contradiction, assume that ψ does not satisfy C and, consequently, $x \notin X$ and $y \in X$. Since X is a minimal vertex cover, there exists $z \in N_H(y)$ such that $z \notin X$ and the clause $C' = y \vee z$ belongs to φ' . If $z = x$, then any satisfying assignment to φ' sets y to true, a contradiction to our first preprocessing step. Otherwise, the presence of C and C' implies that in any assignment ψ' satisfying φ' we have $\psi'(x) = \top$ or $\psi'(z) = \top$. Thus, $x \vee z$ is a clause of φ' , and $xz \in E(H)$. However, neither x nor z belongs to X , a contradiction.

In the last case, $C = \neg x \vee \neg y$ and, again, we assume that ψ does not satisfy C , that is, $x, y \in X$. Since X is a minimal vertex cover, there exist $s \in N_H(x)$, $t \in N_H(y)$ such that $s, t \notin X$. It follows from the definition of H that the clauses $C_x = x \vee s$ and $C_y = y \vee t$ are present in φ' . If $s = t$, then the clauses C , C_x and C_y imply that t is set to true in any truth assignment satisfying φ' , a contradiction to our first preprocessing step. If $s \neq t$, then observe that the clauses C , C_x and C_y imply that either s or t is set to true in any truth assignment satisfying φ' and, consequently, $s \vee t$ is a clause of φ' and $st \in E(H)$. However, $s, t \notin X$, a contradiction.

2.27 If $\emptyset \in \mathcal{A}$, then conclude that we are dealing with a no-instance. Otherwise, for every set $X \in \mathcal{A}$ of size $|X| < d$, create $d - |X|$ new elements and add them to X .

2.28 There are two ways different ways to solve this exercise. First, you can treat the input instance as a d -HITTING SET instance, proceed as in Section 2.6.1 and at the end apply the solution of Exercise 2.27 to the obtained kernel, in order to get an Ed -HITTING SET instance.

In a second approach, try to find a sunflower with $k + 2$ sets, instead of $k + 1$ as in Section 2.6.1. If a sunflower is found, then discard one of the sets: the remaining $k + 1$ sets still ensure that the core needs to be hit in any solution of size at most k .

2.29 Show that in a $(dk + 2)$ -sunflower, one set can be discarded without changing the answer to the problem.

2.30 Proceed as in Exercise 2.27 and pad every set $X \in \mathcal{A}$ with $d - |X|$ newly created elements.

2.31 Show that, in a split graph, every path can be shortened to a path on at most four vertices. Thus, for every $1 \leq i \leq k$, we have a family \mathcal{F}_i of vertex sets of possible paths between s_i and t_i , and this family is of size $\mathcal{O}(n^4)$. Interpret the problem as a d -SET PACKING instance for some constant d and family $\mathcal{F} = \bigcup_{i=1}^k \mathcal{F}_i$. Run the kernelization algorithm from the previous exercise, and discard all vertices that are not contained in any set in the obtained kernel.

2.32 Show that, in a graph excluding a d -vertex path as an induced subgraph, every path in a solution can be shortened to a path on at most $d - 1$ vertices. Proceed then as in Exercise 2.31.

2.33 Let \mathcal{A} be a family of all vertex sets of a P_3 (induced path on three vertices) in G . In this manner, CLUSTER VERTEX DELETION becomes a 3-HITTING SET problem on family \mathcal{A} , as we need to hit all induced P_3 s in G . Reduce \mathcal{A} , but not exactly as in the d -HITTING SET case: repeatedly find a $k + 2$ sunflower and delete one of its elements from \mathcal{A} . Show that this reduction is safe for CLUSTER VERTEX DELETION. Moreover, show that, if \mathcal{A}' is the family after the reduction is exhaustively applied, then $(G[\bigcup \mathcal{A}'], k)$ is the desired kernel.

2.34 Use the following observation: a perfect graph is bipartite if and only if it does not contain a triangle. Thus, the problem reduces to hitting all triangles in the input graph, which is a 3-HITTING SET instance.

2.35 Proceed as in the case of CLUSTER VERTEX DELETION: interpret a SPLIT VERTEX DELETION instance as a 5-HITTING SET instance.

2.36 Let $\{C_4, C_5, 2K_2\}$ be the set of *forbidden induced subgraphs* for split graphs. That is, a graph is a split graph if it contains none of these three graphs as an induced subgraph.

You may need (some of) the following reduction rules. (Note that the safeness of some of them is not so easy.)

1. The “standard” sunflower-like: if more than k forbidden induced subgraphs share a single edge (and otherwise are pairwise edge-disjoint), delete the edge in question.
2. The irrelevant vertex rule: if a vertex is not part of any forbidden induced subgraph, then delete the vertex in question. (Safeness is not obvious here!)
3. If two adjacent edges uv and uw are contained in more than k induced C_4 s, then delete uv and uw , and replace them with edges va and wb , where a and b are new degree-1 vertices.
4. If two adjacent edges uv and uw are contained in more than k pairwise edge-disjoint induced C_5 s, then delete uv and uw , and decrease k by 2.
5. If the edges v_1v_2 , v_2v_3 and v_3v_4 are contained in more than k induced C_5 s, delete v_2v_3 and decrease k by 1.

2.37 By induction, show that every graph on at least 4^k vertices has either a clique or an independent set on k vertices. Observe that this implies a kernel of exponential size for the problem.

2.38 Show that every tournament on n vertices has a transitive subtournament on $\mathcal{O}(\log n)$ vertices. Then, use this fact to show that every oriented directed graph on n vertices has an induced directed acyclic subgraph on $\log n$ vertices. Finally, obtain an exponential kernel for the considered problem.

Bibliographic notes

The history of preprocessing, such as that of applying reduction rules to simplify truth functions, can be traced back to the origins of computer science—the 1950 work of Quine [391]. A modern example showing the striking power of efficient preprocessing is the commercial integer linear program solver CPLEX. The name “lost continent of polynomial-time preprocessing” is due to Mike Fellows [175].

Lemma 2.2 on equivalence of kernelization and fixed-parameter tractability is due to Cai, Chen, Downey, and Fellows [66]. The reduction rules for VERTEX COVER discussed in this chapter are attributed to Buss in [62] and are often referred to as Buss kernelization in the literature. A more refined set of reduction rules for VERTEX COVER was introduced in [24]. The kernelization algorithm for FAST in this chapter follows the lines provided by Dom, Guo, Hüffner, Niedermeier and Truß [140]. The improved kernel with $(2 + \varepsilon)k$ vertices, for $\varepsilon > 0$, is obtained by Bessy, Fomin, Gaspers, Paul, Perez, Saurabh, and Thomassé in [32]. The exponential kernel for EDGE CLIQUE COVER is taken from [234], see also Gyárfás [253]. Cygan, Kratsch, Pilipczuk, Pilipczuk and Wahlström [114] showed that EDGE CLIQUE COVER admits no kernel of polynomial size unless $\text{NP} \subseteq \text{coNP/poly}$ (see Exercise 15.4 point 16). Actually, as we will see in Chapter 14 (Exercise 14.10), one can prove a stronger result: no subexponential kernel exists for this problem unless $P = \text{NP}$.

König’s theorem (found also independently in a more general setting of weighted graphs by Egervary) and Hall’s theorem [256, 303] are classic results from graph theory, see also the book of Lovasz and Plummer [338] for a general overview of matching theory. The crown rule was introduced by Chor, Fellows and Juedes in [91], see also [174]. Implementation issues of kernelization algorithms for vertex cover are discussed in [4]. The kernel for MAXIMUM SATISFIABILITY (Theorem 2.16) is taken from [323]. Abu-Khzam used crown decomposition to obtain a kernel for d -HITTING SET with at most $(2d - 1)k^{d-1} + k$ elements [3] and for d -SET PACKING with $\mathcal{O}(k^{d-1})$ elements [2]. Crown Decomposition and its variations were used to obtain kernels for different problems by Wang, Ning, Feng and Chen [431], Prieto and Sloper [388, 389], Fellows, Heggernes, Rosamond, Sloper and Telle [178], Moser [369], Chlebík and Chlebíkova [93]. The expansion lemma, in a slightly different form, appears in the PhD thesis of Prieto [387], see also Thomasse [420, Theorem 2.3] and Halmos and Vaughan [257].

The Nemhauser-Trotter theorem is a classical result from combinatorial optimization [375]. Our proof of this theorem mimics the proof of Khuller from [289]. The application of the Nemhauser-Trotter theorem in kernelization was observed by Chen, Kanj and Jia [81]. The sunflower lemma is due to Erdős and Rado [167]. Our kernelization for d -HITTING SET follows the lines of [189].

Exercise 2.11 is from [392], Exercise 2.15 is from [121, 383], Exercise 2.16 is from [281], Exercise 2.18 is from [251] and Exercise 2.23 is from [190]. An improved kernel for the above guarantee variant of MAXIMUM BISECTION, discussed in Exercise 2.18, is obtained by Mnich and Zenklusen [364]. A polynomial kernel for SPLIT EDGE DELETION (Exercise 2.36) was first shown by Guo [240]. An improved kernel, as well as a smaller kernel for SPLIT VERTEX DELETION, was shown by Ghosh, Kolay, Kumar, Misra, Panolan, Rai, and Ramanujan [228]. It is worth noting that the very simple kernel of Exercise 2.4 is probably optimal by the result of Kratsch, Philip, and Ray [308].

CS 602

Assignment - 1

Saksham Rathi

3rd year, B.Tech CSE

22B1003

CS 602 - Applied Algorithms: Assignment 1

Total Marks - 50

Instructions. Please try to be brief, clear, and technically precise. Use pseudo-codes to describe the algorithms. To solve the problems, one may assume that the instances are in general position, unless stated otherwise. Novelty in the answer carries marks.

- Given a point set P , we would like to perform a k -median clustering of it, where we are allowed to ignore m of the points. These m points are *outliers* which we would like to ignore since they represent irrelevant data. Unfortunately, we do not know the m outliers in advance. It is natural to conjecture that one can perform a local search for the optimal solution. Here one maintains a set of k centers and a set of m outliers. At every point in time the algorithm moves one of the centers or the outliers if it improves the solution.

Show that local search does not work for this problem; namely, the approximation factor is not a constant. [15 Marks]

- Show that Sauer's lemma is tight. Specifically, provide a finite range space that has the number of ranges as claimed by the lemma. [15 Marks]
- Given a set of n unit disks \mathcal{S} in the plane, we are interested in the problem of computing the minimum length TSP that visits all the disks. Here, the tour has to intersect each disk somewhere.

- Let P be the set of centers of the disks of \mathcal{S} . Show how to get a constant factor approximation for the case that $\text{diam}(P) \leq 1$. [10 Marks]
- Extend the above algorithm to the general case. [10 Marks]

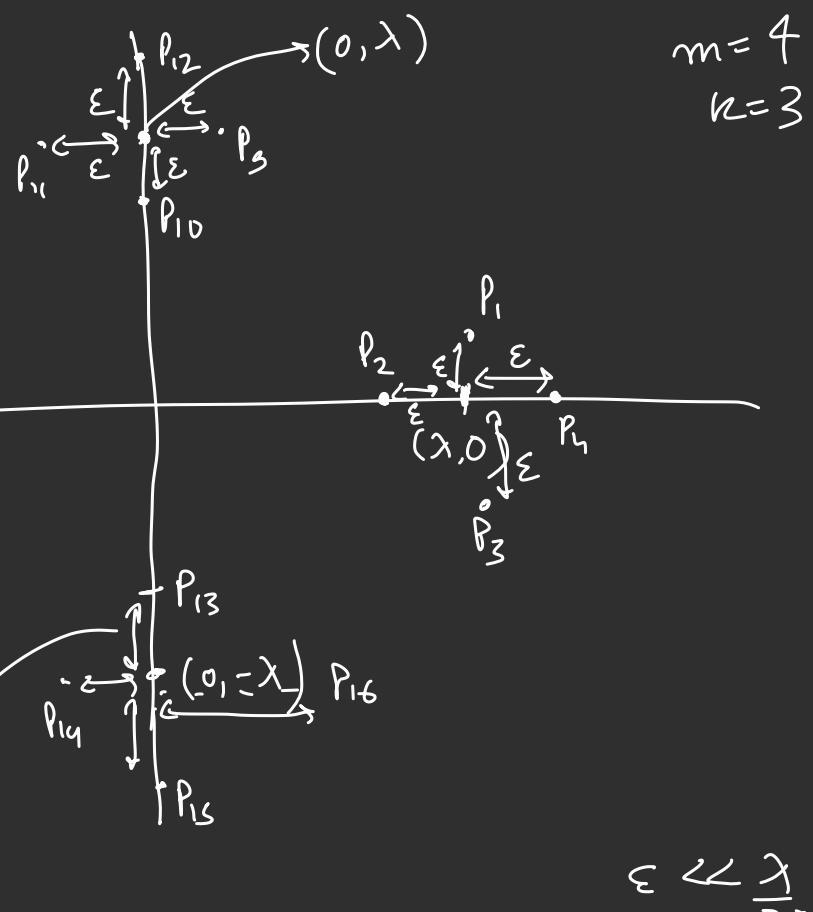
1. k-medians clustering with m-outliers

Now, to show that the approximation factor for local search is not constant, we will have to create a suitable example.

One possible example design is $(k+1)$ sets of points, where in each set, the points are closely to each other, and each set has size = m . The local search will not be able to get out of such a set. And then, we can have the intra-set distances different for different sets.

For example :

Each set has 4
points equidistant from
some centers



$$\underbrace{\varepsilon}_{\text{is very small}} \ll \frac{\lambda}{20}$$

Clearly, local search algorithm might get stuck in the cluster $(P_{13}, P_{14}, P_{15}, P_{16})$ and mark some

other cluster points (e.g. P_1, P_2, P_3, P_4) as outliers.

But this solution will be far from the optimal solution which involves $(P_{13}, P_{14}, P_{15}, P_{16})$ as outliers and the other 3 as $k=3$ clusters.

Sauer's Lemma: If (X, \mathcal{R}) is a range space of VC-dimension $\leq S$ with $|X| = n$, then $|\mathcal{R}| \leq G_S(n)$

$$\text{where } G_S(n) = \sum_{i=0}^S \binom{n}{i}$$

Proof of Tightness: We basically need to construct an example where the equality holds.

Consider $Y = \mathbb{R} \rightarrow$ set of all real numbers
(basically the real line)

and $\mathcal{R} =$ set of all intervals.

Consider $X \subseteq Y$

where $X =$ set of n points $= \{x_1, x_2, \dots, x_n\}$

We already know that the VC-dimension of the range space (real line, intervals) is of size two. [No subset of size = 3 can be shattered].

We need to find $|\mathcal{R}|$. To realize empty set (\emptyset), there must be an interval which does not intersect with any of these n points. Then, to get singleton points, there should be intervals which intersect with only one of the points. (n in number). Then to get two point intersection,

we can choose intervals of the form $[x_i, x_j] \quad x_i \leq x_j$
 $1 \leq i < j \leq n$ (Number of intervals of this form = $\frac{n(n-1)}{2}$)

$$|\mathcal{R}| = 1 + n + \frac{n(n-1)}{2}$$

$$G_S(n) = \sum_{i=0}^S \binom{n}{i} = \binom{n}{0} + \binom{n}{1} + \binom{n}{2} = 1 + n + \frac{n(n-1)}{2} = |\mathcal{R}|$$

$(S=2)$

Hence, we have shown that Sauer's Lemma is tight.

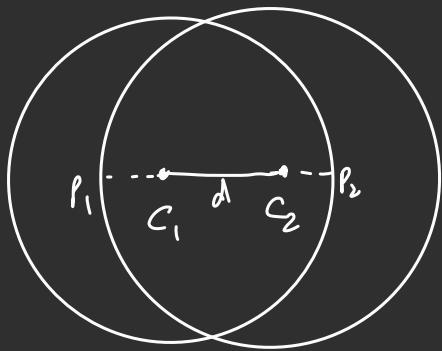
3. (1) We have n unit circles

Consider a pair of circles :

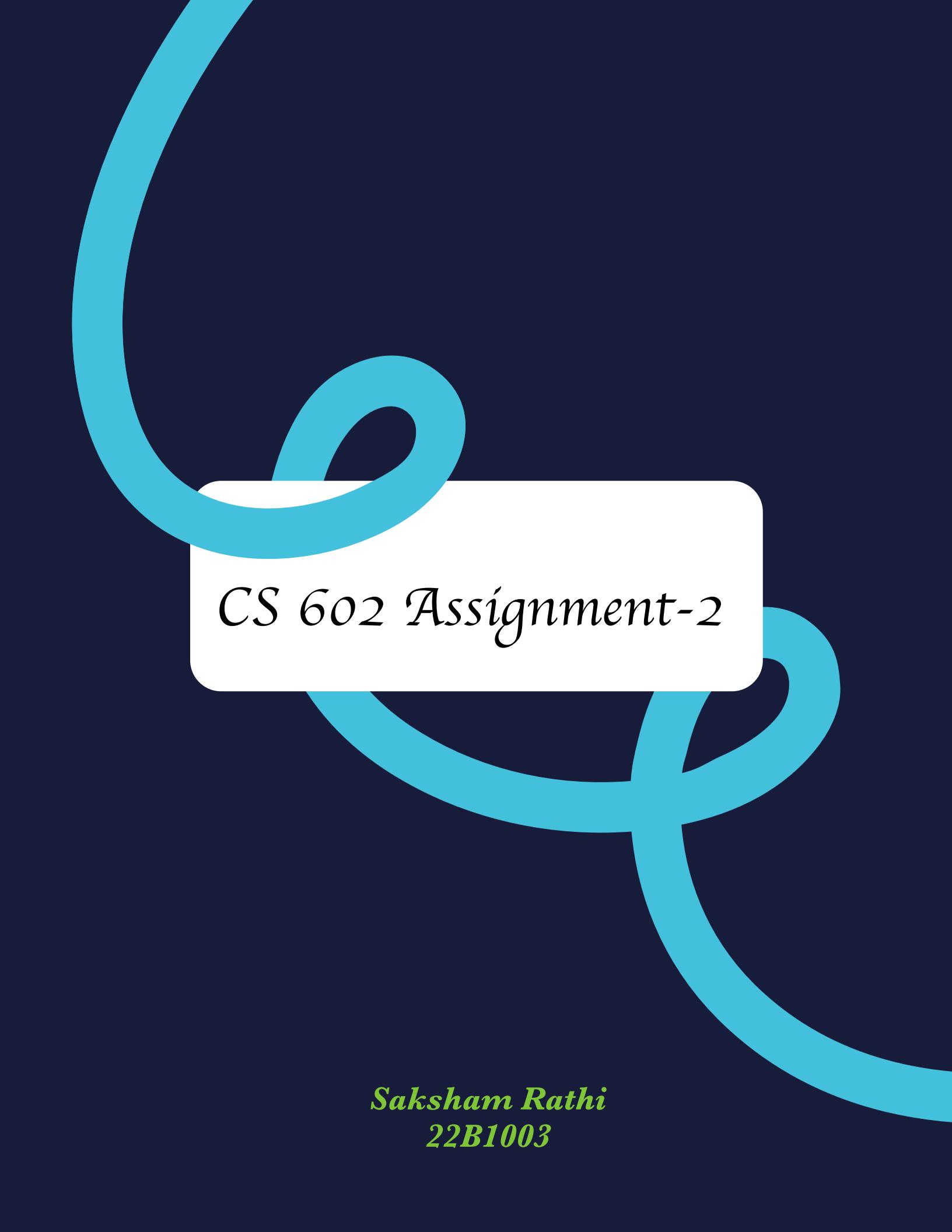


$$d \leq 1$$

(diam ≤ 1)
given



Clearly the distance
between the circles
in the optimal tons
 $\geq P_1P_2$ (they are



CS 602 Assignment-2

Saksham Rathi
22B1003

CS 602 - Applied Algorithms: Assignment 2

Total Marks - 60

Instructions. Please try to be brief, clear, and technically precise. Use pseudo-codes to describe the algorithms. To solve the problems, one may assume that the instances are in general position, unless stated otherwise. Novelty in the answer carries marks.

Question 1[15 Marks] Let $(\mathcal{X}, \mathcal{R})$ and $(\mathcal{X}, \mathcal{R}')$ be two set systems with bounded VC dimension.

Show that $\text{VC dim}(\mathcal{R} \cup \mathcal{R}') \leq \text{VC dim}(\mathcal{R}) + \text{VC dim}(\mathcal{R}') + 1$.

Hint: Use Sauer's lemma.

Question 2[15 Marks] Recall the definition of a metric space. We now introduce a metric called the **shortest-path metric** of a graph. Given a simple, undirected graph G with vertex set V , the distance between any two vertices $u, v \in V$ is defined as the length of the shortest path connecting u and v in G , where the path length is measured by the number of edges it contains. We assume that G is connected. As a simple example, consider the complete graph K_n , which gives the n -point **equilateral space**, where the distance between every pair of vertices is 1.

Show that any tree and its corresponding shortest-path metric on the vertices can be *isometrically embedded* into l_1 (also known as Manhattan distance).

Hint: Begin by considering the embedding of trees with unit-length edges.

Question 3[15 Marks] Consider the star graph below and its corresponding shortest-path metric on the vertices (as defined in question 2).

1. Prove that this graph cannot be isometrically embedded into l_2 .
2. Show that minimum distortion for embedding this graph into l_2 is $\frac{2}{\sqrt{3}}$.

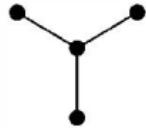


Figure 1: Star graph

Question 4[15 Marks] Let \mathcal{U} be a set of size n , and let $\mathcal{X} \subseteq \mathcal{U}$ be a subset of size k . Let $\xi : \mathcal{U} \rightarrow [k]$ be a coloring of the elements of \mathcal{U} , chosen uniformly at random (i.e. each element of \mathcal{U} is colored with one of the k colors uniformly and independently at random). Then the probability that the elements of X are colored with pairwise distinct colors is at least e^{-k} .

$$S = (X, R) \quad S' = (X, R') \quad S'' = (X, R \cup R')$$

$$\dim_{VC}(S) = s \quad \dim_{VC}(S') = s'$$

Let X'' be the largest set that can be shattered by S'' i.e.

$$\dim_{VC}(S'') = X'' \quad S'' = \dim_{VC}(S'')$$

Clearly, $|X''| > \max(s, s')$ as any set of size $\max(s, s')$ can be shattered

by either the ranges R or R'

W.L.O.G. $s > s'$

Claim: S'' cannot shatter a subset of size greater than $s + s' + 1$

Proof: Say S'' can shatter a subset of size $s + s' + 2$

$$\text{since } 2^{X''} \leq |R \cup R'| \text{ if } X'' \text{ is to be shattered by } S''.$$

$$|R \cup R'| \leq |R| + |R'| \leq \sum_{i=0}^s \binom{n}{i} + \sum_{i=0}^{s'} \binom{n}{i}$$

(from union bound and Sauer's lemma)

$$\text{if } s + s' + 2 > n \text{ then } s'' \leq n < s + s' + 2$$

$$\Rightarrow s'' \leq s + s' + 1 \Rightarrow \text{Contradiction}$$

Case where $s + s' + 2 \leq n$: (X', R) and (X', R') → consider these range spaces

where $X' \subseteq X$ $|X'| = s + s' + 2$ and $\dim_{VC}(X', R') \leq s'$ (because this is a subset)

Clearly $\dim_{VC}(X, R) \leq s$ and $\dim_{VC}(X', R') \leq s'$

$$|R \cup R'| \leq |R| + |R'| \leq \sum_{i=0}^s \binom{|X'|}{i} + \sum_{i=0}^{s'} \binom{|X'|}{i}$$

$$|R \cup R'| \leq \sum_{i=0}^s \binom{s+s'+3}{i} + \sum_{i=0}^{s'} \binom{s+s'+2}{i} = \sum_{i=0}^s \binom{s+s'+2}{i} + \sum_{i=0}^{s'} \binom{s+s'+2}{s+s'+2-i}$$

$$= \sum_{i=0}^s \binom{s+s'+2}{i} + \sum_{i=0}^{s+s'+2} \binom{s+s'+2}{i} - \binom{s+s'+2}{s+1} = 2^{\frac{s+s'+2}{2}} - \binom{s+s'+2}{s+1}$$

$$|R \cup R'| \leq 2^{s+s'+2}$$

⇒ A contradiction

⇒ S'' cannot shatter of size $s + s' + 2$

⇒ $\dim_{VC}(X, R \cup R') \leq \dim_{VC}(X, R) + \dim_{VC}(X, R') + 1$

2) firstly we prove that a line graph can be embedded exactly into l_1 over \mathbb{R} . $G = (V, E)$ with vertices $V = \{1 \dots n\}$
 $edges E = \{(1,2), (2,3), \dots, (n-1, n)\}$
embedding can be defined recursively : $f(1) = 0$ $f(1:i) = f(i-1) + d_G(i-1, i)$

Now, consider a tree $T = (V, E)$ on n vertices. This can be embedded exactly into l_1 over \mathbb{R}^{n-1} which can be seen by induction. As the base case, when $|V| = 2$, T is a line graph, which we have seen can be embedded into l_1 over \mathbb{R} ,

Tree $T_k = (V_k, E_k)$ with k vertices. Since it is a tree, we must have leaf v' and the corresponding edge $(u; v')$. We can remove these to obtain the tree $T_{k-1} = (V_{k-1}, E_{k-1})$ and inductively embed this into l_1 . $f: V_{k-1} \rightarrow \mathbb{R}^{n-2}$ (assumption of induction)

$g: V_k \rightarrow \mathbb{R}^{n-1} \rightarrow$ we need to construct this.

For every $v \in V_{k-1}$ and $v' \in V_k$ $g(v) = (f(v), 0)$
 $g(v') = (f(v'), d_{T_k}(v, v'))$

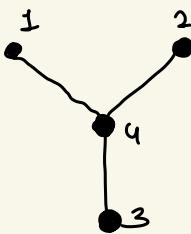
The l_1 distances between any two vertices in V_{k-1} have not changed in this new embedding (since we have simply added a 0 in the new coordinate of all such vertices). Finally, the l_1 distance between any vertex $v \in V_{k-1}$ and v' is

$$d(v, v') = d(v, v') + d(v, v')$$

since there is a unique path between any pair of vertices in a tree.
This is isometrically embedded by g .

(If the edge weights are 1, then we can replace d by 1 in the definition of g).

3) (1)



$$d(1,4) = d(2,4) = d(3,4) = 1$$

$$d(i,j) = 2 \text{ otherwise } (i \neq j)$$

(shortest path metric)

$(1, 2, 3, 4)$ map to $(f(1), f(2), f(3), f(4))$ on some dimension \mathbb{R}^m

Clearly,

$$\|f(1) - f(2)\|_2 \leq \|f(1) - f(4)\|_2 + \|f(2) - f(4)\|_2$$

↓

for isometric embedding these values are $(2, 1, 1)$

since, there is equality here, it means in the embedded space, they have to lie on a single straight line.

(triangle inequality is strict in \mathbb{R}^n except collinear triple).

By similar arguments, $\{1, 2, 4\}$, $\{1, 3, 4\}$, and $\{2, 3, 4\}$ are on a single line. Therefore, all four points must be on the same line which clearly leads to contradiction. Therefore this graph cannot be isometrically embedded into ℓ_2 .

(2) We need to show that minimum distortion for embedding this graph into ℓ_2 is $\frac{2}{\sqrt{3}}$. Let Δ denote the triangle formed by $a'b'c'$ where $a' = f(1)$, $b' = f(2)$ and $c' = f(3)$. Next, consider the quantity $\max(\|a' - s'\|, \|b' - s'\|, \|c' - s'\|)$,

which lower bounds the distortion of f . This quantity is maximized when $s = \|a' - s'\| = \|b' - s'\| = \|c' - s'\| \Rightarrow s'$ is center of the smallest enclosing circle of Δ . However, s is minimized when all the edges of Δ are of equal length and are of length $d_G(1,2)=2$.

Height of the equilateral triangle with side length 2 $\Rightarrow h = \sqrt{3}$

$$\text{radius of inscribing circle} \Rightarrow r = \left(\frac{2}{3}\right)h = \frac{2}{\sqrt{3}}$$

$\text{dist}(f) \geq r = \frac{2}{\sqrt{3}}$ - This argument is independent of the target dimension d .

48

$X \subseteq \tilde{U}$
 \tilde{U} set of size n
 subset of
 size k

The number of possible colours for the first element of X is k .
 second elements is $(k-1)$

⋮
 last element is 1

Total number of possibilities in which elements of X
 are coloured with pairwise distinct colors is $(k?)$

overall, number of possibilities = k^n (each element can
 take one of the
 k colors)

$$\text{Probability (required)} = \frac{k!}{k^n}$$

By Stirling approximation,

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(e^{\frac{1}{12n} - \frac{1}{360n^3}}\right) < n!$$

$$\left(\frac{k!}{k^n}\right) > e^{-k} \underbrace{e^{\frac{1}{12k} - \frac{1}{360k^3}}}_{>1} > e^{-k}$$

$$\Rightarrow \frac{k!}{k^n} = \text{prob} > e^{-k}$$

Hence, we have proved, the probability that the elements of X
 are coloured with pairwise distinct colors is at least e^{-k} .

(Derivation of Stirling's Approximation
 can be found here) \Rightarrow

CS602

Assignment-3

Saksham Rathi
22B1003

CS 602 - Applied Algorithms: Assignment 3

Total Marks - 60

Instructions. Please try to be brief, clear, and technically precise. Use pseudo-codes to describe the algorithms. To solve the problems, one may assume that the instances are in general position, unless stated otherwise. Novelty in the answer carries marks.

- **Question 1:** [20 Marks] *Ski rental problem*

At a ski resort, renting a ski costs 1 rupee per day, while buying skis costs B rupees. A skier arrives at the ski resort for a ski vacation and has to decide whether to rent or buy skis. However, an unknown factor is the number of remaining skiing days left before the snow melts. A randomized $\frac{e}{e-1}$ competitive algorithm exists.

1. Construct a simple deterministic 2- competitive algorithm for the problem.
2. Formulate the problem as a linear program.

Hint: define an indicator variable which is set to 1 if the skier buys the skis, and for each day $i, i \in [1, k]$ (which is unknown in advance), define another indicator variable which is set to 1 if the skier decides to rent skis on day j . The constraints guarantee that on each day, we either rent skis or buy them.

3. Formulate the dual program for this.
4. Construct a 2-competitive algorithm using primal-dual method.

Hint: Whenever we have a new ski day, the primal program is updated by adding a new constraint. The dual program is updated by adding a new dual variable. The online requirement is that previous decisions cannot be undone. In other words, the primal variables are monotonically non-decreasing over time.

- **Question 2:** [20 Marks] Suppose we have one machine and jobs released over time: job i is released at time r_i , has size w_i , benefit b_i , and deadline

d_i . Jobs are allowed to be preempted (i.e., interrupted and later resumed) and/or partially executed (as long as it is before the deadline). Denote by $p_i \leq w_i$ the total time job i was processed before its deadline. Our goal is to maximize $\sum_i \frac{p_i}{w_i} b_i$. Assume that r_i, w_i, d_i (but not b_i) are integers and the algorithm as well as OPT are allowed to preempt jobs only at integer times. Design a 2-competitive algorithm for the problem.

- **Question 3:** [20 Marks] *MARKING Problem* : Consider a randomized paging algorithm that processes a request sequence in phases. At the beginning of each phase, all pages in the memory system are unmarked. Whenever a page is requested, it is marked. On a fault, a page is chosen uniformly at random from among the unmarked pages in fast memory, and this page is evicted. A phase ends when all pages in fast memory are marked and a page fault occurs. Then, all marks are erased and a new phase is started.

The competitive ratio of a randomized online algorithm A is defined with respect to an adversary. The adversary generates a request sequence σ and it also has to serve σ . When constructing σ , the adversary always knows the description of A . We hereby define the notion of oblivious adversary.

Oblivious Adversary: The oblivious adversary has to generate a complete request sequence in advance, before any requests are served by the online algorithm. The adversary is charged the cost of the optimum online algorithm for that sequence.

Prove that the MARKING algorithm is $2H_k$ -competitive against any oblivious adversary, where $H_k = \sum_{i=1}^k \frac{1}{i}$ is the k-th Harmonic number.

Question 1 (1) A 2-Competitive algorithm ensures that the cost incurred by the algorithm is at most twice the cost of an optimal offline solution.

We can design a deterministic algorithm as follows :

The skier rents skis for B days (where B is the cost of buying the skis).

- The skier rents skis for B days, if the skier still needs skis, they buy them.
- On the B^{th} day, if the skier still needs skis, they buy them.
- After $(B)^{\text{th}}$ day, the cost is 0 for skier, since it has already bought the ski.

Proof that it is 2-competitive :

If the skier skies for fewer than B days, the cost is at most B

- If the skier skies for fewer than B days, the cost is at most B which is already optimal

- If the skier skies for $(B+x)$ days $x > 0$

$$\Rightarrow \text{optimal} = B$$

$$\Rightarrow \text{from deterministic algo} = (B-1) + B = 2B - 1$$

$$\text{ratio} = \frac{2B-1}{B} \leq 2$$

Hence, this is a deterministic 2-competitive algorithm

(2) x is a binary variable that is 1 if the skier buys the skis

y_i is a binary variable that is 1 if the skier rents skis on day i , where $i \in [1, k]$ ($k = \text{unknown}$)

$$\text{Cost } C = Bx + \sum_{i=1}^k y_i$$

Objective: minimize C

Constraints: On each day j :

$$y_i + x \geq 1 \quad \text{for } j = 1 \dots k$$

Either the skier rents skis or has already bought them.

x and y_j are binary

(3)

Primal

$$Ax \geq b$$

$$x \geq 0$$

$$\min(w^T x)$$

$$A = \begin{pmatrix} 1 & 1 & 0 & \dots & 0 \\ 1 & 0 & 1 & \dots & 0 \\ \vdots & 1 & 0 & 0 & \dots & 1 \end{pmatrix}_{k \times (k+1)}$$

$$x = \begin{pmatrix} x \\ y_1 \\ \vdots \\ y_k \end{pmatrix}_{(k+1) \times 1}$$

$$b = \begin{pmatrix} ; \\ ; \\ ; \end{pmatrix}_{k \times 1}$$

$$w = \begin{pmatrix} B \\ \vdots \end{pmatrix}_{(k+1) \times 1}$$

Dual

$$\max(y^T b)$$

$$A^T y \leq w$$

$$y \geq 0$$

$$y = \begin{pmatrix} \gamma_1 \\ \vdots \\ \gamma_k \end{pmatrix}$$

$$\max(\gamma_1 + \dots + \gamma_k)$$

$$\max\left(\sum_{i=1}^k \gamma_i\right)$$

$$\text{Constraints: } \gamma_i \geq 0 \quad \forall i = 1 \dots k$$

$$\gamma_1 + \gamma_2 + \dots + \gamma_k \leq B$$

$$\gamma_j \leq 1 \quad \forall j = 1 \dots k$$

Dual problem

(4)

We will start with an empty set of constraints in the primal.

For each new day j , we will update the primal program by adding the constraint $y_j + x \geq 1$ and the corresponding

dual variable γ_j .

We will increase γ_j in the dual as long as the constraints $\sum_{j=1}^k \gamma_j \leq B$ and $\gamma_j \leq 1$ are satisfied

Steps of the Algorithm :

On each new day, set $y_j = 1$ to rent skis and increase γ_j in the dual.

Continue until $\sum_{j=1}^k \gamma_j = B$ at which point we will set $x = 1$
(buy the skis)

We are also ensuring that the previous decisions are unchanged,
satisfying the online requirement.

* This algorithm aligns with the deterministic algorithm and is
therefore 2-competitive

Question-2

Job i released at time r_i , has size w_i , benefit b_i and deadline d_i

Here is a simple greedy 2-competitive algorithm \Rightarrow

For each available job i at time t (i.e. $r_i \leq t < d_i$)

→ Compute the benefit density $\frac{b_i}{w_i}$

Sort jobs by highest benefit density to lowest.

If there is a tie, prioritize the job with the nearest deadline d_i

At each integer time t , choose the highest priority job i according to the priority defined above and process it for unit amount of time.

Bump the job if a new job j with higher priority arrives at time $t+1$
(i.e. $r_j = t+1$)

If p_i reaches w_i (the job is fully processed) or the time reaches d_i (i.e. the deadline), remove the job from the queue

At any time $t \rightarrow$ our algorithm will pick the job with highest $\frac{b_i}{w_i}$

At some time t , OPT is running something and our algo isn't. Let's say this has value A. We are running best possible in that time. Let's say this has value B. In the worst case scenario, OPT will run the best case later on completely. But in that frame, what OPT is running cannot be better than what we are running.

Thus, we get B and OPT gets $A+B$, $A \leq B$

This is true for every time frame, summing over all time instants, we get that our algo is 2-competitive.

Question 3

We need to prove that the randomized marking algorithm (RMA) has competitive ratio $2H_k$ against any oblivious adversary. ($H_n = n^{\text{th}}$ harmonic number)

We divide the sequence of requests (σ) into phases. The i^{th} phase ends immediately before the $(k+l)^{\text{th}}$ distinct page is requested in the phase. If we denote the set of pages requested in phase i by P_i , then at the end of a phase i , the contents of RMA's cache is exactly P_i .

m_i = number of new requests in phase i . (not present in phase $i-1$)

Let us suppose that just before the j^{th} old page is requested, there have been l new pages requested so far in the phase. By induction, we can show that at this time there are exactly l pages in P_{i-1} that are not in the cache. These are distributed randomly (uniform) among the $k-(j+l+1)$ unmarked pages in P_{i-1} . Probability that it is not in the cache is exactly $\frac{l}{k-(j+l+1)}$ $\underbrace{l \leq m_i}_{\text{by definition}}$

$$P(\text{fault on the request to the } j^{\text{th}} \text{ old page}) \leq \frac{m_i}{k-(j+m_i-1)}$$

$$\mathbb{E}[\text{cost}_{\text{RMA}}(\sigma)] \leq m_i + \sum_{1 \leq j \leq k-m_i} \frac{m_i}{k-j-m_i+1} \leq m_i H_k$$

Summing up over all the phases, $\mathbb{E}[\text{cost}_{\text{RMA}}(\sigma)] \leq H_k \leq m_i$

We need to prove a lower bound for the optimal cost.

$$\underline{\text{Claim: }} \text{cost}_{\text{OPT}}(\sigma) \geq \sum_i \frac{m_i}{2}$$

Consider the $(i-1)^{\text{st}}$ and i^{th} phases. The number of distinct pages requested in both phases is $k+m_i$. OPT has only k pages in the cache at the beginning of the $(i-1)^{\text{st}}$ phase, it must incur at least m_i faults during the two phases. We can apply the same argument to every pair of adjacent phases, we have that $\text{cost}_{\text{OPT}}(\sigma) \geq \sum_i m_i$ and $\text{cost}_{\text{OPT}}(\sigma) \geq \sum_i m_{2+i}$. Therefore, OPT has cost at least the average of these i.e. $\text{cost}_{\text{OPT}}(\sigma) \geq \sum_i \frac{m_i}{2}$. Thus $\mathbb{E}[\text{cost}_{\text{RMA}}(\sigma)] \leq 2H_k \text{cost}_{\text{OPT}}(\sigma) \Rightarrow \text{Marking algorithm} = 2H_k \text{competitive}$