Approximation Algorithms

Notes

Winter 2021/22

Martin Hoefer

Organizational

Email: mhoefer@cs.uni-frankfurt.de
Office: 115, R.M.S. 11-15, physical office hours postponed until further notice

Lectures:

- Tue/Thu every week, mostly writing on the board, recorded as videos
- Course is **similar (but not identical)** to the recent ones by Annamaria Kovacs.
- All relevant text is in this document – only figures and drawings are missing (I make them up on the fly in the lecture)
- This document is continously **expanded, updated and corrected**.
- Will provide additional material on the website.

Exercises/Exams:

- Weekly exercise sheets, published on Tuesday of week i, due Tuesday of week $i + 1$,
- Turn it in **before 10:15h** via moodle, use **PDF format**.
- Great to discuss solutions, but write them up in **your own words**
- Solutions returned and discussed in the next exercise session
- If you score $x\%$ of total number of exercise points, then
  If $50 \leq x < 75$, one grading step bonus for exam (e.g., 2.0 to 1.7, or 3.7 to 3.3)
  If $75 \leq x$, two grading steps bonus for exam (e.g., 2.0 to 1.3, or 3.3 to 2.7)
- Exams in February/March 2022 (oral or written depending on number of people)

Contents

Chapter 1

Introduction and Basic Ideas

1.1 Warm-Up Example

As an elementary problem to outline the basic ideas and concepts, consider the MAKESPAN SCHEDULING problem on identical machines:

- m identical machines, n tasks
- Task $i \in [n]$ has processing time $p_i > 0$
- Assign each task i (completely) to some machine $j \in [m]$
- Load of machine j is $\ell_j = \sum_{i \text{ on } j} p_i$
- Maximum load $\max_{j \in [m]} \ell_j$ is called the **makespan** of the assignment

Goal: Assign tasks to machines to minimize the makespan

[Pic: Example Scheduling instance, assignment, makespan]

MAKESPAN SCHEDULING is an *offline* problem, i.e., all tasks, machines and processing times are known and available upfront. Note that MAKESPAN SCHEDULING is NP-hard (why?), so it is very unlikely that the problem can always be solved optimally in polynomial time. Still, are there **good algorithms**, even though they might not always solve the problem optimally? How can we evaluate and compare the performance of such suboptimal algorithms?

Consider Algorithm **ListScheduling**:
Choose any permutation π of tasks. For $i = 1, \ldots, n$: assign task $\pi(i)$ to any machine that has curerntly smallest load ℓ_j.

Clearly, ListScheduling does not always comput an assignment with smallest makespan. Still, it is not overly bad! In fact, we will show the following theorem.

Theorem 1. *For any instance I of* MAKESPAN SCHEDULING *let $OPT(I)$ be the smallest makespan of any assignment, and let $LIST(I)$ be the makespan of an assignment computed by ListScheduling. It holds that*

$$LIST(I) \leq \left(2 - \frac{1}{m}\right) \cdot OPT(I) \ .$$

More formally, using our terminology defined below, we will say that ListScheduling *obtains a* $(2-1/m)$*-approximation*, or that *the approximation ratio of ListScheduling is at most* $2-1/m$. Intuitively, this means that, on any instance I, the makespan of the assignment computed by ListScheduling is *at most twice* the optimal makespan that can be achieved for that instance.

Proof of Theorem 1. We prove the theorem by showing **lower bounds on the optimum** that are related to the output of the algorithm.

- Consider a task k that upon assigment to machine j gives the makespan load $LIST(I)$.
- k gets assigned to a machine with currently smallest load
- Hence, $\ell_j \geq LIST(I) - p_k$ for all $j \in [m]$
- Clearly, $OPT(I) \geq LIST(I) - p_k$, because up to that time all machines are busy.
- Also: $OPT(I) \geq p_k$, because we must assign task k completely to some machine

[Pic: Schedule, p_k, lower bounds]

The two lower bounds on $OPT(I)$ relate it to $LIST(I)$. Assembling these bounds shows that a ratio of 2

$$LIST(I) \leq OPT(I) + p_k \leq 2 \cdot OPT(I) \ . \tag{1.1}$$

Looking a bit closer,

- $OPT(I)$ must be at least the average load of all machines.
- At time $LIST(I) - p_k$, all machines are busy, so the total load must be at least

$$\sum_{i \in [m]} p_i \geq m(LIST(I) - p_k) + p_k \ .$$

- This implies

$$OPT(I) \quad \geq \frac{1}{m} \sum_{i \in [m]} p_i \geq \frac{1}{m} \cdot ((m \cdot (LIST(I) - p_k) + p_k) = \quad LIST(I) - \frac{m-1}{m} \cdot p_k$$

Reasoning as in (1.1) above shows

$$LIST(I) \leq OPT(I) + \frac{m-1}{m} \cdot p_k \leq \left(1 + \frac{m-1}{m}\right) \cdot OPT(I) = \left(2 - \frac{1}{m}\right) \cdot OPT(I) \ .$$

□

Is this the best guarantee we can have? Is the algorithm maybe even better? Is there an instance, where the algorithm returns an assignment with almost twice the optimal makespan?

Theorem 2. *There is an instance I and a permutation of tasks such that*

$$LIST(I) = \left(2 - \frac{1}{m}\right) \cdot OPT(I) \ .$$

Consider the following instance. There is one task with $p_1 = m$ and $m(m-1)$ tasks with $p_i = 1$. Suppose ListScheduling first assigns all small tasks. The large task 1 in the end yields a makespan of $(m-1) + m$. Instead, there is an assignment with makespan of m.

1.2 Optimization and Approximation

The notion of approximation algorithms applies for **optimization problems**, in which we evaluate a solution with an objective function, which is supposed to be either **minimized** or **maximized**. Some examples of such problems are

Vertex Cover problem:
- Undirected graph $G = (V, E)$
- Subset of vertices $C \subseteq V$ is a *vertex cover* if for every edge $e \in E$ there is at least one incident vertex in C
- **Goal:** Find a vertex cover C with minimal size $|C|$.

[Pic: Example Vertex Cover]

Clique problem:
- Undirected graph $G = (V, E)$
- Subset of vertices $C \subseteq V$ is a *clique* if for every pair $u, v \in C$ the edge exists $\{u, v\} \in E$
- **Goal:** Find a clique C with maximal size $|C|$.

[Pic: Example Clique]

Independent Set problem:
- Undirected graph $G = (V, E)$
- Subset of vertices $S \subseteq V$ is an *independent set* if for every pair $u, v \in S$ the edge does not exist $\{u, v\} \notin E$
- **Goal:** Find an independent set S with maximal size $|S|$.

[Pic: Example Independent Set]

Max-Matching problem:
- Undirected graph $G = (V, E)$
- Subset of edges $M \subseteq E$ is a *matching* if for every pair of edges $e, e' \in M$ the edges have no common endvertex $e \cap e' = \emptyset$
- **Goal:** Find a matching M with maximal size $|M|$.

[Pic: Example Matching]

Note that we are discussing **optimization problems** – and not **decision problems** (where we just decide yes or no). Formally, decision problems are the basis for the complexity classes P and NP. It is, however, easy to see that optimization and decision problems are closely related. An optimization problem can be turned into a decision problem by asking whether there is a solution with value at most or at least a given value k. For example, in the decision problem of Clique, we are given a graph G and a value k, and the problem is to decide if there is a clique of size at least k (rather than *finding a largest* clique).

Formally, we will consider problems from a class NPO of optimization problems.

Definition 1. *A problem is in the class* NPO *if the following conditions hold. Given any input I as an instance and L as a solution, the following can be computed/decided in time polynomial in $|I|$:*

1. *Does I represent an instance of the problem?*
2. *If yes, does L represent a feasible solution for the input instance I?*
3. *If yes, compute the objective function value of the solution L for the instance I.*

Note that it is *not necessary* to have a polynomial-time algorithm to *compute a solution*! Instead, for problems in NPO there is a **non-deterministic polynomial-time** optimization algorithm: There is a non-deterministic Turing machine that "guesses an optimal solution", then verifies in polynomial time that it is a solution and computes its' objective function value.

Suppose we have an algorithm that always computes an optimal solution for, say, Clique. Clearly, such an algorithm can be used to *decide the decision version* of Clique. However, the decision version of Clique is NP-complete! As such, we cannot expect the algorithm to simultaneously

1. compute an optimal solution,
2. run in polynomial time, and
3. do this for every instance of the problem.

Instead, we want a *good (suboptimal) algorithm* for the problem that *always runs in polynomial time*, i.e., we here relax condition (1). Note that there is also lots of research on relaxing (2) (e.g., exact exponential-time algorithms), or (3) (e.g., algorithms for special cases of problems).

Our approach is to measure the **worst-case ratio** of the solution quality obtained by an algorithm compared to the optimal solution.

Definition 2. *Consider a minimization problem, and let I be any instance. We denote by $OPT(I)$ the objective function value of an optimal solution.*

- *For any $\alpha \geq 1$, a solution is α-**approximate** if it has a value of at most $\alpha \cdot OPT(I)$.*
- *Consider an algorithm ALG, and let $ALG(I)$ be the objective function value of the computed solution. For any $\alpha \geq 1$, ALG **obtains an α-approximation** if for every instance I of the problem*

$$ALG(I) \leq \alpha \cdot OPT(I) \qquad or \qquad \frac{ALG(I)}{OPT(I)} \leq \alpha.$$

- *Similarly, for a maximization problem a solution is α-approximate if it has a value of at least $OPT(I)/\alpha$, and ALG obtains an α-approximation if for every instance I of the problem*

$$ALG(I) \geq OPT(I)/\alpha \qquad or \qquad \frac{OPT(I)}{ALG(I)} \leq \alpha.$$

*We say the **approximation ratio** of the algorithm is **at most** α.*

Note that bounds on the approximation ratio of an algorithm shall hold for *every instance* I of a problem. Hence, to bound the ratio from above (i.e., show that the algorithm is "good"), we must show that the computed solution yields the guarantee *in every single instance of that problem* (c.f. Theorem 1). To bound the ratio from below (i.e., show that the algorithm is "bad"), it suffices to *construct a single instance* I, on which the ratio $\frac{ALG(I)}{OPT(I)}$ (or $\frac{OPT(I)}{ALG(I)}$ for maximization) attains the bound (c.f. Theorem 2).

1.3 PTAS and FPTAS

Can we improve the ratio of $2 - 1/m$ for Makespan Scheduling? We focus on instances with **a constant number m of machines**. Imagine m as a small value, e.g., $m = 5$.

- In ListScheduling, the task k achieving the makespan upon assignment might be a large one – we assume above that it could be as large as $OPT(I)$).
- Suppose in our instance I all tasks have time $p_i \leq OPT(I)/2$.
- Repeating the bound in (1.1) above, the approximation ratio of ListScheduling decreases to at most 1.5:

$$LIST(I) \leq OPT(I) + p_k \leq 1.5 \cdot OPT(I).$$

- Ok, great, but what if there are tasks with $p_i > OPT(I)$?
- There can be at most 2 such tasks per machine, i.e., at most $2m$ in total.
- Consider the $2m$ biggest tasks. Call them **big tasks** (no matter how big they actually are), and the others small.
- Note: Every small task i must have $p_i \leq OPT(I)/2$.

Consider the following algorithm: Find an **optimal assignment of the $2m$ big tasks**, i.e., one that minimizes the makespan of these tasks. Then use ListScheduling to complete the assignment and add remaining small tasks.

Running time:

- Each big task could go to each machine $\Rightarrow m^{2m}$ possible assignments of big tasks.
- Need time $O(2mn)$ to find the big tasks, $O(m^{2m})$ for finding their optimal assignment, and $O(n \log m)$ to complete the assignment with ListScheduling.
- Since m is a constant, overall running time is $O(mn + m^{2m} + n \log m) = O(n)$.

Analysis of approximation ratio:

- Case 1: Machine j with *only big tasks* has ℓ_j equal to makespan.

$\Rightarrow$ Schedule was optimal for big tasks and has same makespan even after adding all tasks

$\Rightarrow$ Schedule is optimal for all tasks, approximation ratio is 1!

- Case 2: Only machines j with *small task* have ℓ_j equal to makespan.

$\Rightarrow$ Use analysis of ListScheduling above. Since $p_k < OPT(I)/2$, ratio at most 1.5.

[Pic: Approximation Ratio, big task attains makespan]

How far can we push this idea? **Arbitrarily far!**

- Suppose we take $3m$ big tasks. Then the small tasks must have size $p_i \leq OPT(I)/3$.

Algorithm 1: BruteForceList

Input: n tasks with processing times p_i, m machines, parameter $\varepsilon > 0$

Determine the set T of the $\min(n, m \cdot \lceil 1/\varepsilon \rceil)$ tasks with largest processing time.
Find an optimal assignment for the tasks in T.
Run ListScheduling to complete the assignment for the remaining tasks.
return assignment of tasks to machines

- For the ratio, we obtain a factor of 1 in case 1. In case 2, we get a ratio of 4/3:

$$ALG(I) \leq OPT(I) + p_k \leq 4/3 \cdot OPT(I)$$

- Ok, lets use $B \cdot m$ tasks for a big $B > 0$. Then small tasks have size $p_i \leq OPT(I)/B$.
- The ratio becomes at most (1 in case 1 and) $1 + 1/B$ (in case 2).
- The bigger B the better the approximation ratio. Arbitrarily good ratios are possible!

Hey, wait a minute, but **what about the running time**!?

- Optimal assignment of big tasks needs time $O(m^{Bm})$.
- This must be polynomial in n! So B better be a constant, independent of n.
- More precisely, $B = c \cdot \log_m n$ for constant c would still be fine, since then $O(m^{Bm}) = O(n^{cm})$ and c and m are constants.

We formally introduce the notion of a **polynomial-time approximation scheme (PTAS)**.

Definition 3. *A PTAS for an optimization problem is a family A_ε of algorithms such that, for every $\epsilon > 0$, the approximation ratio of algorithm A_ε is $(1 + \varepsilon)$, and the running time of A_ε is polynomial in the input size* (but not necessarily in $1/\varepsilon$).

Consider **Algorithm BruteForceList** (Algorithm 1) for Makespan Scheduling with constantly many machines. The algorithm is parametrized by $\varepsilon > 0$. Observe that with $B = \lceil 1/\varepsilon \rceil \leq 1/\varepsilon + 1$, the algorithm exactly implements our idea above. Hence, the approximation ratio becomes $1 + 1/B \leq 1 + \varepsilon$ and the running time is bounded by $O(nm/\varepsilon + m^{m/\varepsilon + m})$.

Theorem 3. *Algorithm BrueForceList is a PTAS for* Makespan Scheduling *with constantly many machines.*

Some remarks:

- Running time is huge. Even with small values, say $m = 10$ and $\varepsilon = 0.01$, it becomes astronomical and totally impractical! So what's the point?
- PTAS'es often serve as an orientation for the complexity of a problem – is there something fundamental in the structure of the problem that excludes arbitrarily good approximation?
- Here our running time is exponential in $1/\varepsilon$ (ok) and m (less ok). It is *not a PTAS* for plain Makespan Scheduling, without the assumption of a constant number m of machines.

Can we further push the limits? The ratio is already best possible, but the running time might be improved. In a **fully** polynomial-time approximation scheme **(FPTAS)** we have an even better dependence on $1/\varepsilon$.

Definition 4. *An FPTAS for an optimization problem is a family A_ε of algorithms such that, for every $\epsilon > 0$, the approximation ratio of algorithm A_ε is $(1+\varepsilon)$, and the running time of A_ε is polynomial in the input size* and in $1/\varepsilon$.

Definition 5. PTAS *is the class of problems in* NPO *that admit a PTAS, and* FPTAS $\subseteq$ PTAS *the class of problems that admit an FPTAS.*

What about Makespan Scheduling?

- Above we saw Makespan Scheduling has a PTAS when we have constantly many machines. It is not an FPTAS.
- In fact, there exists also an FPTAS when we have constantly many machines.
- There exists a PTAS for plain Makespan Scheduling with arbitrarily many machines, but no FPTAS!

Do all problems admit an FPTAS? Consider Clique.

- Suppose Clique has an FPTAS. For every given $\varepsilon > 0$ and every graph with $n = |V|$ nodes, it computes a $(1+\varepsilon)$-approximation in time polynomial in n and $1/\varepsilon$.
- Let's set $\varepsilon = 1/n$, then the running time is polynomial in n and $1/\varepsilon = n$.
- Let $OPT(I)$ be the size of the largest clique in the graph. The FPTAS with $\varepsilon = 1/n$ obtains a clique of size

$$\begin{aligned} FPTAS(I) &\geq \frac{OPT(I)}{1+1/n} \\ &= \frac{n}{n+1} \cdot OPT(I) \\ &= OPT(I) - \frac{OPT(I)}{n+1} \\ &> OPT(I) - 1 \end{aligned}$$

- Clique sizes are integers, so $FPTAS(I) = OPT(I)$.
- The FPTAS can be forced to compute an optimal solution in polynomial time!

[Pic: Granularity, approximation bound]

The problem is the **granularity of the optimal solution**. There are only $n+1$ integer values for the size of the clique. At some point, the approximation guarantee forces the solution value to be so close to the optimum that there is only one possible value left...

We observe this condition more generally.

Theorem 4. *Consider an* NP*-hard optimization problem, such that for every instance I*

1. *the objective function attains only non-negative integers, and*
2. *there is a polynomial q such that every solution has value at most $q(|I|)$.*

Then there is no FPTAS for this problem unless P = NP.

Algorithm 2: Match-And-Cover

```
1 M ← ∅
2 Let V[M] be the vertices incident to edges in M
3 while there is e ∈ E such that e ∩ V[M] = ∅ do M ← M ∪ {e}
4 return V[M]
```

The proof follows by repeating our arguments for CLIQUE using $\varepsilon = 1/q(|I|)$.

MAKESPAN SCHEDULING with constant number of machines admits an FTPAS. Why is it not covered by the previous theorem?

- Suppose all processing times are integers. Then the input size is in $O\left(\sum_{i\in[n]} \log p_i\right)$.
- The makespan lies in the interval $\max_{i\in[n]} p_i, \ldots, \sum_{i\in[n]} p_i$. This can be **exponential in the input size**. Theorem 4 does not apply!

1.4 The Class APX

Theorem 4 shows that many problems do not allow an FPTAS (unless $\mathsf{P} = \mathsf{NP}$). For example, VERTEX COVER, CLIQUE, INDEPENDENT SET are all problems of this type. For some problems, we can at least show a constant-factor approximation, for which the approximation ratio does not depend on the input size. This is the best condition if no (F)PTAS can be achieved.

For example, consider **Algorithm Match-And-Cover** (Algorithm 2) for VERTEX COVER.

Theorem 5. *Match-And-Cover obtains a 2-approximation for* VERTEX COVER *in polynomial time.*

Proof. The algorithm first computes a **non-extendable matching** M.

- No two matching edges $e, e' \in M$ have a common endvertex.
- True initially, maintained inductively throughout the while-loop.
- Consider an optimal vertex cover C^*. Every matching edge $e \in M$ must be covered by a **separate vertex**. Hence, $|C^*| \geq |M|$.
- The cover $C = V[M]$ returned by the algorithm has size $|C| = 2 \cdot |M|$. Thus,

$$|C| = 2 \cdot |M| \leq 2 \cdot |C^*|.$$

But why is C a **feasible vertex cover**?

- When the while-loop terminates, no edge can be added to M without intersecting it.
- Hence, every edge $e' \in E \setminus M$ shares at least one endvertex with some edge $e \in M$
- This endvertex is in $C \Rightarrow C$ covers all edges.

[Pic: Matching edges, separate vertices in C^*] □

Can we approximate VERTEX COVER in polynomial time to within a ratio less than 2? Some lower bounds (without proofs):

- $10\sqrt{5} - 21 \approx 1.36$ under standard conjectures ($\mathsf{P} \neq \mathsf{NP}$)
- $2 - \varepsilon$, for every constant ε, under stronger conjectures ("unique games conjecture")
- Improved results exist for special graph classes, e.g., bipartite graphs.

Definition 6. APX *is the class of problems in* NPO *with the following property. There is an efficient algorithm ALG and some constant $c \geq 1$ independent of the input size, such that ALG obtains a c-approximation.*

Note that there are even problems that do not admit any constant-factor approximation, e.g., the Clique problem.

Theorem 6. *For every constant $\delta > 0$, there is no efficient algorithm for the* Clique *problem with approximation ratio $n^{1-\delta}$, unless* $\mathsf{P} = \mathsf{NP}$.

We might return to give a proof of the theorem later in the lecture. Let us quickly note that there is a trivial algorithm to compute a **n^1-approximate solution**: Pick a single vertex. It is a clique of size $|C| = 1$. The optimum has size $|C^*| \leq n$. Hence: $|C| \geq |C^*|/n$.

Definition 7. *$f(n)$-*APX *is the class of problems in* NPO *such that there is an efficient algorithm that obtains an $O(f(n))$-approximation on instances with input size n.*

Similarly, we can define log-APX, poly-log-APX, or poly-APX as the classes of problems that allow efficient algorithms with approximation ratios that are logarithmic in n, or poly-logarithmic in n, or polynomial in n, respectively. Finally, we can define PO as the problems from NPO that can be solved optimally in polynomial time.

Observe

$$\mathsf{PO} \subset \mathsf{FPTAS} \subset \mathsf{PTAS} \subset \mathsf{APX} \subset \text{log-}\mathsf{APX} \subset \text{poly-log-}\mathsf{APX} \subset \text{poly-}\mathsf{APX} \subset \mathsf{NPO}$$

[Pic: Classes and examples]

Chapter 2

Greedy Algorithms

2.1 Priority Algorithms

Greedy algorithms make **iterative, non-reversible decisions** often based on **partial input** to advance towards a feasible (hopefully near-optimal) solution.

This is a very imprecise description.

We focus on **priority algorithms**, a subclass of greedy algorithms with a more precise and rigorous definition:

- Instance consists of data elements (tasks with processing times, release dates or deadlines; edges with weights; vertices with weights, neighborhoods; etc.)
- There is an initial ordering of **all possible data elements that can occur** in any possible instance.
- A priority algorithm sequentially picks a remaining element with highest priority, makes an irreversible decision about this element
- Priorities of remaining elements can change after a decision is made (if this happens, we call the algorithm *adaptive*).

As an example, let us consider the Interval Scheduling problem:

- n tasks, each with a release date $r_i > 0$ and a deadline $d_i > r_i > 0$.
- Subset of intervals S is **non-overlapping** if for every pair $i, j \in S$, we have $r_i \geq d_j$ or $d_i \leq r_j$.
- **Goal:** Maximize the number of non-overlapping intervals

Consider **Algorithm GreedyIntervals** (Algorithm 3). It is a (non-adaptive) priority algorithm – intervals are considered iteratively in non-decreasing order of deadlines. The algorithm computes an optimal solution in polynomial time.

[Pic: Example Interval Scheduling, Greedy Algorithm]

Theorem 7. *GreedyIntervals solves the* Interval Scheduling *problem optimally in polynomial time.*

Proof. By induction on the number of tasks in the instance:

Algorithm 3: GreedyIntervals for Interval Scheduling

```
1 Sort tasks in non-decreasing order of deadlines
2 S ← ∅
3 for task i in non-decreasing order of deadlines do
4     if S ∪ {i} is non-overlapping then S ← S ∪ {i}
5 return S
```

- Base: $n = 1$: Single task, Greedy picks it, optimal.
- Hypothesis: Greedy is optimal when applied to any subset of ℓ tasks, for any $\ell \leq n-1$.
- Step: Consider optimal solution S^* for n tasks. Let task 1 be the one with earliest deadline.

Case $1 \in S^*$:

- Both GreedyIntervals and S^* choose task 1.
- Remove 1 and tasks overlapping with it $\rightarrow$ remaining set T of $|T| < n$ tasks.
- By hypothesis: Greedy computes optimal solution S^*_{-1} for T.
- $S^* \setminus \{1\}$ is a feasible solution for T, but S^*_1 is optimal. Hence $|S^*_{-1}| \geq |S^* \setminus \{1\}|$.
- Thus: $|S^*_{-1} \cup \{1\}| \geq |S^*|$. Greedy also computes an optimal solution for n tasks.

Case $1 \notin S^*$:

- Delete any tasks from S^* that overlap with task 1, then add 1 to S^*
- We delete at most one task. Hence, there is optimal solution S' with $1 \in S'$.
- Apply Case $1 \in S^*$.

□

[Pic: Intervals, task 1, S^* is composition of task 1 and optimal solution for remaining tasks T]

As another example, consider the Min-Spanning Tree problem:

- $G = (V, E)$ is an undirected graph, each edge $e \in E$ has **edge weight** $w(e) \in \mathbb{R}$
- Subset $T \subseteq E$ of edges is
- ...**cycle-free**: Induced subgraph $G[T]$ contains no cycle.
- ...**tree**: $G[T]$ is connected and cycle-free
- ...**spanning** tree: $G[T]$ contains all vertices from V.
- G not connected: Spanning *forest*, i.e., spanning tree for every connected component.
- We define $w(S) = \sum_{e \in S} w(e)$ as the sum of edge weights, for any edge set $S \subseteq E$

Goal: A spanning tree (or forest) T^* that minimizes the sum of edge weights $w(T^*)$

Recall **Kruskal's algorithm** (Algorithm 4), which is another (non-adaptive) priority algorithm (also note the striking similarity to GreedyIntervals)! The algorithm computes an optimal solution in polynomial time.

[Pic: Spanning Tree, Example Kruskal Algo]

Theorem 8. *Kruskal's algorithm solves the* Min-Spanning Tree *problem optimally in polynomial time.*

Algorithm 4: Kruskal Algorithm for Min-Spanning Tree

```
1 Sort edges in non-decreasing order of edge weight
2 T ← ∅
3 for edge e in non-decreasing order of weight do
4     if T ∪ {e} is cycle-free then T ← T ∪ {e}
5 return T
```

Proof. By induction on the number of *vertices* in the graph:

- Base: $|V| = 2$: At most one edge, algo picks it, optimal.
- Hypothesis: Algo is optimal when applied to any graph with $n - 1$ vertices.
- Step: Consider optimal solution T^* for G with n vertices. Let e_1 be the edge with smallest weight.

Step 1: Show that we can assume $e_1 \in T^*$

- Suppose $e_1 \notin T^*$. Then $T^* \cup \{e_1\}$ contains exactly one cycle C (why?)
- Adding e_1 and removing any edge e' from C results in another spanning tree T'
- By definition $w(e_1) \leq w(e')$, so

$$w(T') = w(T^*) + w(e_1) - w(e') \leq w(T^*)$$

- T^* is optimal, so T' must be another optimal solution.
- Hence, we can assume $e_1 \in T^*$.

[Pic: Add e_1 to T^*, cycle, remove costlier edge]

Step 2: Use hypothesis to finish proof

- Kruskal pics e_1. No problem, there is optimal T^* that does so, too.
- Contract $e_1 = \{u, v\}$: Merge vertices u and v into new vertex x. All edges of u and v now incident to x (except e_1, which gets removed).
- Two edges between x and some vertex $v' \rightarrow$ keep only the one with smaller weight
- Let the resulting graph be G'. Note: G' has $n - 1$ vertices.
- T' optimal solution for $G' \Rightarrow T' \cup \{e_1\}$ optimal solution for G (why?)
- By hypothesis, Kruskal computes optimal T' in G', and $T' \cup \{e_1\}$ in G.

[Pic: Contraction, optimal solution in contracted graph] □

Our proof does *not require* that weights $w(e)$ are *positive* – the algorithm works even for **negative weights**! It also solves the Max-Spanning Tree problem: Simply multiply all weights by -1 and solve the Min-Spanning Tree problem.

2.2 Matroids

We study greedy algorithms for subset selection problems. These selection problems generalize Max-Spanning Tree.

We first define the notion of a downward-closed set system.

Algorithm 5: Greedy for Weighted Set Systems

```
1 Sort elements of R in non-increasing order of element weight w(r)
2 Y ← ∅
3 for element e in non-increasing order of weight do
4     if (Y ∪ {r}) ∈ I then add Y ← Y ∪ {r}
5 return Y
```

- There is a ground set R of **elements** (e.g., edges in a graph)
- **Set system** $(\mathcal{I}, R)$: $\mathcal{I}$ is set of subsets of R, i.e., $\mathcal{I} \subseteq 2^R$, a subset of the power set of R. $\mathcal{I}$ can contain *some* of the subsets of R, but maybe not *all* like the power set.
- **Downward-Closed**: If $X \in \mathcal{I}$ and $Y \subset X$, then $Y \in \mathcal{I}$.

[Pic: Lattice of subsets, if a set is in $\mathcal{I}$, then all subsets are in there as well]

Examples for a given graph $G = (V, E)$:

- Let $(\mathcal{T}, E)$ be such that $\mathcal{T}$ contains all *cycle-free edge sets*. If X is a cycle-free edge set, then $Y \subset X$ is also cycle-free $\Rightarrow (\mathcal{T}, E)$ is downward-closed.
- Let $(\mathcal{M}, V)$ be such that $\mathcal{M}$ contains all *node set of independent sets*. If X is an independent set, then $Y \subset X$ is also an independent set $\Rightarrow (\mathcal{M}, V)$ is downward-closed.
- Let $(\mathcal{C}, V)$ be such that $\mathcal{C}$ contains all *vertex covers*. If X is a vertex cover, then $Y \subset X$ might not always be a vertex cover. This system is not downward-closed.

[Pic: Examples]

Suppose the set system is **weighted**: Every element $r \in R$ has a weight $w(r) \geq 0$.
Goal: Find a set $Y^* \in \mathcal{I}$ that maximizes the sum of weights $w(Y^*)$ of elements in Y^*.

We focus on **downward-closed set systems**.

- For $(\mathcal{T}, E)$, we want to find a max-weight cycle-free subset of edges
  $\rightarrow$ Max-Spanning Tree, can be solved in polynomial time.
- For $(\mathcal{M}, V)$, we want to find a max-weight independent node set
  $\rightarrow$ Generalizes Independent Set, (very) hard to approximate.

What is different in these set systems? Consider the **Greedy algorithm** (Algorithm 5).

- Run the algorithm for $(\mathcal{M}, V)$. Initially, $S = \emptyset$
- We always extend Y by a vertex v of highest weight when $Y \cup \{v\}$ contains no neigbors
- At some point, we might obtain a suboptimal, yet non-extendable set of nodes
- In contrast, for $(\mathcal{T}, E)$ this Kruskal's algorithm – here non-extendable = optimal.

[Pic: Greedy independent set, non-extendable and suboptimal]

Definition 8. *We define three properties of a downward-closed set system* $(\mathcal{I}, R)$*:*

(a) ***Greedy Optimality****: For every assignment of element weights* $w(r) \geq 0$*, Algorithm 5 computes an optimal set* Y^**.*

(b) ***Extendability***: *For every $X, Y \in \mathcal{I}$ with $|Y| < |X|$, there is some element $r \in X \setminus Y$ such that $(Y \cup \{r\}) \in \mathcal{I}$.*
(c) ***Maximal Cardinality***: *For every set $Z \subseteq R$, all sets $X \subseteq Z$ that are non-extendable w.r.t. Z have the same cardinality.*

The properties seem quite different. In particular, greedy optimality seems stronger than extendability, which in turns seems stronger than maximal cardinality. It turns out, though, that they are completely equivalent!

Theorem 9. *Greedy optimality, extendability and maximal cardinality are equivalent.*

A downward-closed set system with these properties is called a **matroid**. As a classic example, given a subset R of vectors in Euclidean space $\mathbb{R}^d$, the subsets of linearly independent vectors form a matroid. This gives an intuition for the nomenclature.

Definition 9. *A downward-closed set system $(\mathcal{I}, R)$ with the properties is called a **matroid**. The sets in $\mathcal{I}$ are called **independent sets**. The sets of $\mathcal{I}$ with largest cardinality are called **bases**, and their cardinality is the **rank** of the matroid.*

Before proving Theorem 9, we observe that $(\mathcal{T}, E)$ is a matroid. $(\mathcal{T}, E)$ is downward-closed, so by Theorem 9, we need to show only one of the three properties. For completeness, we show all of them:

- Greedy for $(\mathcal{T}, E)$ is Kruskal's algorithm, computes an optimal cycle-free subset of edges, for every set of weights $w(e) \geq 0$ (as proved above).
- Extendability: Consider two cycle-free subsets of edges E_1 and E_2, where $|E_1| < |E_2|$.
- In (V, E_1) we have $n - |E_1|$ components, in (V, E_2) only $n - |E_2| < n - |E_1|$
  $\Rightarrow$ There are nodes u and v connected in E_2, which are not connected in E_1
  $\Rightarrow$ At least one $e \in E_2$ on u-v-path that connects two components in (V, E_1)
  $\Rightarrow$ e can be added to E_1 without closing a cycle.
- Maximal cardinality: Maximal subsets are spanning forests of $G = (V, E)$. Every spanning forest has $n - d$ edges, with d the number of connected components in G. This is true also in every subgraph $G[Z]$ induced by edge set $Z \subseteq E$.

[Pic: Extend E_1 by an edge of E_2]

Proof of Theorem 9. We prove that (a) $\Rightarrow$ (b) $\Rightarrow$ (c) $\Rightarrow$ (a)

(a) $\Rightarrow$ (b): Greedy optimality $\Rightarrow$ Extendability.

- Suppose Greedy computes optimal solution in $(\mathcal{I}, R)$ for all non-negative weights.
- Now suppose this $(\mathcal{I}, R)$ does not satisfy extendability:
  There are $X, X' \in \mathcal{I}$ with $|X| < |X'|$ such that $X \cup \{r\} \notin \mathcal{I}$ for every $r \in X' \setminus X$.
- We show: Then there exist element weights $\hat{w}(r)$ such that Greedy is suboptimal
  $\Rightarrow$ Contradiction.
- How should the weights $\hat{w}$ look like? We use $\varepsilon < 1/|X|$ and set

$$\hat{w}(r) = \begin{cases} 1 + \varepsilon & \text{if } e \in X \\ 1 & \text{if } e \in X' \setminus X \\ 0 & \text{otherwise.} \end{cases}$$

Now run Greedy on $(\mathcal{I}, R)$ with $\hat{w}$:

- It first adds all elements from X. This is ok – every subset of X is in $\mathcal{I}$, so Greedy can always extend it by another element from X.
- No element from $X' \setminus X$ can be added, all of them get discarded.
- Greedy might add more elements from $R \setminus X'$ of value 0.
- Terminates with a set of value $|X|(1+\varepsilon) < |X| + 1 \leq |X'|$, since $\varepsilon < 1/|X|$
- Greedy solution is suboptimal.

[Pic: X, X', weights]

(b) ⇒ (c): Extendability ⇒ Maximal cardinality

- Suppose in $(\mathcal{I}, R)$ we have extendability. Consider any $X, Y \in \mathcal{I}$ with $|Y| < |X|$.
- Then the smaller set Y is extendable (can be extended at least one element from X).
- Thus, all non-extendable sets $X, Y \in \mathcal{I}$ must have $|X| = |Y|$.

(c) ⇒ (a): Maximal cardinality ⇒ Greedy optimality

- Suppose we have the maximal cardinality property.
- For non-negative weights, there is some optimal $Y^* \in \mathcal{I}$ that is non-extendable.
- Greedy computes a set Y that is also non-extendable (→ Exercise)
- Suppose there are weights $\hat{w}(r)$ such that Y is suboptimal.
- Number the elements $Y = \{r_1, \ldots, r_d\}$ and $Y^* = \{r_1^*, \ldots, r_m^*\}$ in non-increasing order of their weights, respectively. Note that Y and Y^* can overlap.
- Since in sum $\hat{w}(Y) < \hat{w}(Y^*)$, there must exist an index j such that $\hat{w}(r_j) < \hat{w}(r_j^*)$.
- Let k be the smallest such index, $Y_{k-1} = \{r_1, \ldots, r_{k-1}\}$ and $Z = \{r_1, \ldots, r_{k-1}, r_1^*, \ldots, r_k^*\}$.
- At some point, Greedy constructs S_{k-1} and then chooses r_k with weight strictly smaller than every r_i^*, for $i = 1, \ldots, k$. Hence, none of the r_i^* can be used to extend S_{k-1}.
- Y_{k-1} is non-extendable w.r.t. elements from Z.
- Let $Y_k^* = \{r_1^*, \ldots, r_k^*\} \subseteq Z$, which is a subset of Y^* and hence in $\mathcal{I}$. There is a set $Y' \in \mathcal{I}$ with $Y_k^* \subseteq Y' \subseteq Z$ that is non-extendable w.r.t. Z.
- We see that $|Y'| \geq k > |Y_{k-1}| \Rightarrow$ Contradiction.

[Pic: Elements in Y, Y^* in non-increasing order of weight, index k, subsets Y_{k-1}, Y_k^*, Z] □

Graphic Matroid: The set system $(\mathcal{T}, E)$ of cycle-free edge sets of a graph

Matrix Matroid:

- $R =$ set of vectors in Euclidean space $\mathbb{R}^d$
- $\mathcal{I} = \{X \subseteq R \mid X$ contains only linearly independent vectors$\}$
- Downward-closed: Any subset of linearly independent vectors remains independent.
- Extendability: For any two sets of linearly independent vectors X, Y with $|Y| < |X|$, there is a vector $r \in X$ that can be added to Y such that $Y \cup \{r\}$ remains linearly independent.

Uniform Matroid:

- For some integer $k \geq 0$, all subsets of up to k elements: $\mathcal{I} = \{X \subseteq R \mid |X| \leq k\}$
- Downward-closed: Clear.
- Extendability: Clear :)

As a slightly more involved example, consider the MATROID SCHEDULING problem:

- n tasks $(t_i)_{i=1,\dots,n}$, each task t_i has a deadline $d_i \geq 0$ and a penalty cost $c_i \geq 0$
- Single machine, all tasks available at time 0,
- Every task takes 1 unit of processing time to complete
- If task t_i is not completed before time d_i, it is *late*
- For every late task, we have to pay penalty c_i

Goal: Schedule tasks to minimize sum of penalties of late tasks

We want to show that this problem can be formulated using a matroid.

- Elements are tasks, i.e., $R = \{t_1, \dots, t_n\}$
- Minimize penalties of late tasks = maximize penalties of early tasks!
- For a given $X \subseteq R$, in a *feasible* schedule every task of X is finished before its deadline, i.e., no task is getting late.
- X called *feasible* if there is a feasible schedule.
- Set system $\mathcal{I} = \{X \subseteq R \mid X \text{ feasible}\}$
- Weight of task t_i is c_i.

How can we implement the standard greedy algorithm? We use a simple structural lemma.

Lemma 1. *X is feasible $\Leftrightarrow$ Schedule of X in non-decreasing order of d_i is feasible.*

Proof. It never makes sense to leave the machine idle, so we can process tasks back-to-back. Suppose there is a feasible schedule that orders tasks of X differently.

- There are tasks violating the order: t_i processed before t_j, but $d_j < d_i$.
- Both tasks finished before both deadlines!
- Switch positions of t_i and t_j, both tasks finished before both deadlines.
- Repeat argument until all tasks are in non-decreasing order of deadline.

[Pic: Task order, deadlines, switching keeps schedule feasible] □

Corollary 1. *Algorithm 5 can be implemented in polynomial time.*

The main step is to check $(Y \cup \{t_j\}) \in \mathcal{I}$ in polynomial time: Sort tasks in $Y \cup \{t_j\}$ according to deadlines and check that no task is late.

Lemma 2. *$(\mathcal{I}, R)$ is a matroid, so Algorithm 5 computes an optimal solution for* MATROID SCHEDULING.

Proof. Downward-closed: If there is a feasible schedule for $X \subseteq R$, then the same schedule is also feasible for any $Y \subseteq X$.

Extendability: Suppose $X, Y \in \mathcal{I}$ with $|Y| < |X|$. Consider feasible schedules for X and Y in non-decreasing order of deadline, respectively. Consider task $t_i \in X$ with largest deadline.

Case 1: $t_i \notin Y$.

- t_i finished at time $|X|$, so $d_i \geq |X|$. Extend schedule for Y and add t_i as the last task
- Then t_i finished at time $|Y| + 1 \leq |X| \leq d_i$.

Algorithm 6: Greedy for Weighted MAX-MATCHING

```
1 Sort edges of E in non-increasing order of edge weight w(e)
2 M ← ∅
3 for edge e in non-increasing order of weight do
4     if (M ∪ {e}) is matching then add M ← M ∪ {e}
5 return M
```

- Rest of schedule feasible for $Y \Rightarrow$ extended schedule feasible for $Y \cup \{t_i\}$
- Hence, $t_i \in X \setminus Y$ such that $(Y \cup \{t_i\}) \in \mathcal{I}$.

Case 2: $t_i \in Y$.

- Change schedule for Y and switch t_i to last position. Even if one more task would be added before t_i, then t_i still finished before deadline $|Y| + 1 \leq |X| \leq d_i$.
- Drop t_i from consideration in X and Y, repeat argument with $t_i' \in X$ with largest (remaining) deadline.

After at most $|Y|$ repetitions, a task t_j with largest remaining deadline in X is found with $t_j \notin Y$. Place t_j into schedule for Y before the tasks from $X \cap Y$ that were moved to the back in the repetitions. This keeps schedule the same for earlier tasks. Tasks moved to the back still finish before their deadline. Hence, $t_i \in X \setminus Y$ such that $(Y \cup \{t_i\}) \in \mathcal{I}$. □

[Repeated process if t_i with largest deadline from X is in Y]

2.3 k-Matroids

Our initial example in this section is a weighted version of MAX-MATCHING:

- Undirected Graph $G = (V, E)$ with weight $w(e) \geq 0$, for every edge $e \in E$
- Matching $M \subseteq E$: No "neighboring edges", i.e., every pair $e, e' \in M$ has $e \cap e' = \emptyset$.
- Let $\mathcal{M} = \{M \subseteq E \mid M \text{ is a matching in } G\}$
- Is $(\mathcal{M}, E)$ a matroid? Can we find good matchings greedily with Algorithm 6?

Clearly, $(\mathcal{M}, E)$ is a downward-closed set system.

But it is not a matroid – consider a path (e_1, e_2, e_3) of 3 edges. There are two non-extendable matchings, $\{e_1, e_3\}$ and $\{e_2\}$. Clearly, for $w(e_1) = w(e_3) = 1$ and $w(e_2) = 1.1$ the greedy algorithm computes a suboptimal solution.

[Pic: Path, matchings, greedy solution]

Still, the greedy algorithm is not overly bad!

Theorem 10. *Algorithm 6 has an approximation ratio of at most 2.*

Proof. Consider an optimal matching M^*. Let M be a matching computed by Greedy.

- For a moment drop all edges $e \in M \cap M^*$, the ones where Greedy was "correct"
- Why didn't Greedy pick a remaining $e \in M^*$? Because it picked some edge e_1 earlier that overlaps with e.

- This means $w(e_1) \geq w(e)$. We say e_1 is a *witness* for e.
- Hence, every edge $e \in M^*$ has a witness that is at least as valuable.
- Now observe that every $e = \{u, v\} \in M$ can be the witness for *at most 2 edges* from M^* – one incident to u and one to v. Both of these are less valuable.

Overall,

$$w(M^*) = \sum_{e \in M \cap M^*} w(e) + \sum_{e \in M^* \setminus M} w(e) \leq \sum_{e \in M \cap M^*} w(e) + \sum_{e \in M \setminus M^*} 2w(e) \leq 2 \cdot w(M).$$

[Pic: M and M^*, overlapping edges, $e \in M$ witness for at most 2 edges from M^*] □

More fundamentally, consider any two matchings $M, M' \in \mathcal{M}$. Any edge $e \in M$ can forbid the extension of M by at most two edges from M'. Hence, if $|M'| > 2|M|$, there must be some edge $e \in M'$ that can be added to M, i.e., where $M \cup \{e\}$ is a matching. We say $(\mathcal{M}, E)$ satisfies 2-extendability.

As a consequence, the same argument shows that, for every subset Z of edges and any two matchings M, M' that are non-extendable w.r.t. Z, we must have $|M| \leq 2|M'|$, i.e., they differ by at most a factor of 2 in cardinality. We say $(\mathcal{M}, E)$ satisfies 2-maximal cardinality.

Definition 10. *Reconsider the properties of matroids. Consider a downward-closed set system $(\mathcal{I}, R)$. For any number $k \geq 1$, we define*

(a) ***Greedy k-Optimality****: For every assignment of element weights $w(r) \geq 0$, Algorithm 5 has an approximation ratio of at most k.*

(b) ***k-Extendability****: For every $X, Y \in \mathcal{I}$ with $k \cdot |Y| < |X|$, there is some element $r \in X \setminus Y$ such that $(Y \cup \{r\}) \in \mathcal{I}$.*

(c) ***k-Maximal Cardinality****: For every set $Z \subseteq R$, if two sets $X, X' \subseteq Z$ are non-extendable w.r.t. Z, then $|X| \leq k \cdot |X'|$.*

Theorem 11. *Greedy k-optimality, k-extendability and k-maximal cardinality are equivalent.*

Proof. Exercise. □

A downward-closed set system that satisfies these properties is called a **k-matroid**.
Note that $(\mathcal{M}, E)$ discussed above for matchings is a 2-matroid.

For our next example, we consider (yet) another definining property of k-matroids.

Definition 11. *For a downward-closed set system, we consider the following properties:*

- ***exchange****: For any two sets $Y \subset X$ and every $r \notin X$ the following holds: If $(Y \cup \{r\}) \in \mathcal{I}$, then there is $s \in X \setminus Y$ such that $((X \setminus \{s\}) \cup \{r\}) \in \mathcal{I}$.*
- ***k-exchange****: For any two sets $Y \subset X$ and every $r \notin X$ the following holds: If $(Y \cup \{r\}) \in \mathcal{I}$, then there is $S \subseteq X \setminus Y$ such that $|S| \leq k$ and $((X \setminus S) \cup \{r\}) \in \mathcal{I}$.*

[Pic: Exchange $r \notin X$ for some $s \in X \setminus Y$ when $Y \cup \{r\}$ is independent]

We show that k-exchange $\Rightarrow$ k-matroid. The other direction holds only for $k = 1$ (every matroid has the exchange property), but does not always hold for $k > 1$ (there are k-matroids without k-exchange property).

Theorem 12. *Consider a downward-closed set system $(\mathcal{I}, R)$.*
1. *$(\mathcal{I}, R)$ is a matroid. $\implies$ $(\mathcal{I}, R)$ has the exchange property.*
2. *$(\mathcal{I}, R)$ has k-exchange property. $\implies$ $(\mathcal{I}, R)$ is a k-matroid.*

Proof of 1.) Suppose $(\mathcal{I}, R)$ is matroid with the extendability property. Now consider $Y \subset X \in \mathcal{I}$ and $r \notin X$ with $(Y \cup \{r\}) \in \mathcal{I}$. We show that there is $s \in X \setminus Y$ such that $((X \setminus \{s\}) \cup \{r\}) \in \mathcal{I}$.
- Start with $Z = Y \cup \{r\}$. Note that $Z, X \in \mathcal{I}$.
- Case $|Z| = |X|$: $X = Y \cup \{s\}$ for some element s, so $((X \setminus \{s\}) \cup \{r\}) = Z \in \mathcal{I}$. $\surd$
- Case $|Z| < |X|$: Extend Z by some $r' \in X \setminus Z$ to $(Z \cup \{r'\}) \in \mathcal{I}$.
  Repeat until $|Z| = |X|$. Then $Z, X \in \mathcal{I}$ and differ by a single element.
  Previous case applies, shows exchange property.

□

Proof of 2.) Suppose $(\mathcal{I}, R)$ has the k-exchange property. We show k-extendability.
- Suppose $Y, X \in \mathcal{I}$ and $k|Y| < |X|$.
- To show: There is $r \in X \setminus Y$ such that $(Y \cup \{r\}) \in \mathcal{I}$.
- If $Y \subset X$, then we are done $\rightarrow$ $(Y \cup \{r\}) \subseteq X$ and all subsets of X are in $\mathcal{I}$.

So suppose $Y \setminus X \neq \emptyset$.
- Choose some $r \in Y \setminus X$ and let $Z = X \cap Y$.
- Note that $(Z \cup \{r\}) \in \mathcal{I}$ and $r \notin X$. We apply the k-exchange property:
- There is $S \subseteq (X \setminus Z)$ with $|S| \leq k$ such that removing S from and adding r to X gives a set X'. X' is in $\mathcal{I}$.
- Is $Y \subseteq X'$? If not, then $Y \setminus X' \neq \emptyset$. Repeat the previous step with Y, X'.

Consider this repeated process.
- Each repetition removes at most k elements from $X \setminus Y$ and adds one from $Y \setminus X$.
- After at most Y steps, we arrive at a set $X' \in \mathcal{I}$ with $Y \subseteq X'$.
- Recall that $|X| > k|Y|$: There must be $r \in X \setminus Y$ that survives in X'.
- $Y \cup \{r\} \subseteq X'$, so $(Y \cup \{r\}) \in \mathcal{I}$.

This proves k-extendability. □

As a second example for a k-matroid, we consider the MaxTSP problem. We are given a *complete directed* graph $\vec{K_n}$ with a distance $d(u, v) \geq 0$ for every $u, v \in V$, $u \neq v$. The **goal** is to find a tour that visits every vertex exactly once and *maximizes* the sum of distances.

[Pic: Example MaxTSP]

Consider a set system $(\mathcal{I}, R)$ as follows.
- $R = E = \{(u, v) \in V \times V \mid u \neq v\}$, i.e., all ordered pairs of different vertices (graph is complete and directed)
- $\mathcal{I} = \{X \subseteq R \mid X$ consists of node-disjoint directed paths, or is a tour$\}$

[Pic: Example independent sets]

Lemma 3. *$(\mathcal{I}, R)$ is a 3-matroid. Algorithm 5 has approximation ratio 3 for* MaxTSP.

Proof. Downward-closed: If X consists only of node-disjoint paths, removing edges splits them into more paths. If X is a tour, removing edges splits tour into node-disjoint paths.

We show that $(\mathcal{I}, R)$ has the 3-exchange property.

- Consider any set $X \in \mathcal{I}$ and any edge $e = (u, v) \notin X$.
- u and v have at most four incident edges in X
- If we remove all four edges, (u, v) becomes it's own path. Rest stays feasible.
  $\Rightarrow$ Adding e and removing four edges from X, we get another set from $\mathcal{I}$.
  $\Rightarrow$ $(\mathcal{I}, R)$ has the 4-exchange property.
- Note that we can keep an edge leaving v in X, then (u, v) becomes start of a path P.
- We remove all edges of u in X, so P is not a cycle
  $\Rightarrow$ we get a set from $\mathcal{I} \Rightarrow (\mathcal{I}, R)$ has the 3-exchange property.

[Pic: Path in X including u and v, remove edges of u, and incoming edge for v from X] □

This gives a very simple approximation algorithm. By a more precise analysis of the structure of the problem, one can obtain different algorithms with a better ratio., e.g., $10/7 \approx 1.42$ obtained by Paluch (2020).

Finally, let us mention a prominent class of k-matroids. They evolve by intersection of k (1-)matroids.

Theorem 13. *The intersection of k matroids is a k-matroid.*

Proof. Consider the matroids $(\mathcal{I}_1, R), \ldots, (\mathcal{I}_k, R)$ over the same ground set R. The intersection is

$$\mathcal{I} = \mathcal{I}_1 \cap \ldots \cap \mathcal{I}_k.$$

We show $(\mathcal{I}, R)$ has the k-exchange property. Thus, by Theorem 12 it is a k-matroid.

Consider any $Y \subset X \in \mathcal{I}$ and an element $r \notin X$ with $(Y \cup \{r\}) \in \mathcal{I}$.

- Note that $X, Y, (Y \cup \{r\}) \in \mathcal{I}$ and hence in *all* of the $\mathcal{I}_1, \ldots, \mathcal{I}_k$
- By Theorem 12 all matroids have the (1-)exchange property:
  There is $s_1 \in X$ such that $((X \setminus \{s_1\}) \cup \{r\}) \in \mathcal{I}_1$.
  There is $s_2 \in X$ such that $((X \setminus \{s_2\}) \cup \{r\}) \in \mathcal{I}_2$.
  ...
  There is $s_k \in X$ such that $((X \setminus \{s_k\}) \cup \{r\}) \in \mathcal{I}_k$
- Consider $S = \{s_1, \ldots, s_k\}$. For every $i = 1, \ldots, k$, the set

  $$Z = ((X \setminus S) \cup \{r\}) \subseteq ((X \setminus \{s_i\}) \cup \{r\}) \in \mathcal{I}_i$$

  and, hence, $Z \in \mathcal{I}_i$.
- Hence, $Z = ((X \setminus S) \cup \{r\}) \in \mathcal{I}$, and $|S| \leq k$.

□

2.4 Algorithms for MinTSP

In this section, we focus on the MinTSP problem. Similar to the MaxTSP problem, we have a complete directed graph $\vec{K}_n$ with n vertices. Every ordered pair of vertices $u \neq v$ has non-negative distance $d(u, v) \geq 0$. Here the **goal** is to find a tour that visits every vertex exactly once and *minimizes* the sum of distances.

2.4.1 General MinTSP

We first consider the problem in full generality, i.e., we make no assumptions on the distances between the n vertices.

Theorem 14. *There is no polynomial-time algorithm with approximation ratio $\alpha(n)$ for* MinTSP *(unless $\mathsf{P} = \mathsf{NP}$), where $\alpha(n) > 1$ is an arbitrary number that can be represented in poly(n) bits.*

Proof. Consider any such algorithm with ratio $\alpha(n)$. We build a class of MinTSP instances, such that the algorithm can be used to decide the NP-complete problem Hamiltonian Cycle. Hence, if the algorithm exists, we can solve Hamiltonian Cycle in polynomial time and $\mathsf{P} = \mathsf{NP}$.

Hamiltonian Cycle: Given a directed, unweighted graph $G = (V, E)$, is there a Hamiltonian cycle, i.e., a simple cycle including all nodes of V?

We show an NP-hardness reduction. For any instance G of Hamiltonian Cycle with n vertices, we build an instance (n, d) of MinTSP with n vertices and the following properites:

1. The algorithm to build (n, d) from G runs in time polynomial in the input
2. The approximation algorithm for (n, d) can be used to decide if G has a Hamiltonian cycle.

Consider graph $G = (V, E)$ for Hamiltonian Cycle. Construct an instance for MinTSP with n vertices and distances as follows:

- If $(u, v) \in E$, then $d(u, v) = 1$.
- If $(u, v) \notin E$, then $d(u, v) = n \cdot \alpha(n)$.

[Pic: Example]

1.) This construction can be done in polynomial time: We consider only $O(n^2)$ vertex pairs, and for each pair we need to write at most *poly*(n) bits to represent the distance.

2.) Suppose there is an algorithm that computes a tour, which is an $\alpha(n)$-approximation.

1. G has a Hamiltonian cycle $\Rightarrow$ There is a tour of length n. Any tour that includes an edge of length $n \cdot \alpha(n)$ has approximation ratio at least $(n\alpha(n) + n - 1)/n > \alpha(n)$. Thus, the algorithm must return a tour of length n.
2. G has no Hamiltonian cycle $\Rightarrow$ Every tour contains at least one edge of distsance $n \cdot \alpha(n)$. Hence, the algorithm must return a tour of length at least $n \cdot \alpha(n)$.

The tour returned by the $\alpha(n)$-approximation algorithm for MinTSP can be used to read off the correct decision for Hamiltonian Cycle. □

Algorithm 7: MST-Tour for Δ-MinTSP

Input: number n of vertices, metric d

Consider the undirected, complete graph K_n with edge weights given by d
$T \leftarrow$ optimal solution to Min-Spanning Tree in (K_n, d)
Duplicate all edges of T to obtain $\hat{T}$
$C_{\hat{T}} \leftarrow$ Eulerian cycle of $\hat{T}$
Shortcut edges of $C_{\hat{T}}$ into TSP tour C
return C

There is a serious gap in optimal tour length between the instances of MinTSP that are positive (corresponding to graphs with Hamiltonian cycles) or negative (corresponding to graphs without Hamiltonian cycle). Later in the course, we might return to this aspect more systematically.

[Pic: Gap between positive and negative instances]

2.4.2 Metric MinTSP

The challenge in general MinTSP is that distances can have largely different values, even in a "local" sense. Consider, e.g., a triangle with vertices u, v, w. Then we might have $d(u, v) = d(v, w) = 1$, but $d(u, w) = n\alpha(n)$. A natural special case is the **metric** MinTSP (or Δ-MinTSP), in which the distances are a **metric**.

Definition 12. *A function d that assigns non-negative distances to pairs of elements from a ground set V is called a* metric *if it satisfies, for all $u, v, w \in V$,*

1. *Identity of indiscernibles: $d(u, v) = 0 \iff u = v$,*
2. *Symmetry: $d(u, v) = d(v, u)$, and*
3. *Triangle inequality: $d(u, w) \leq d(u, v) + d(v, w)$.*

[Pic: Example triangle inequality]

Our first two approximation algorithms build an *Eulerian cycle* and shortcut it to a TSP tour. An Eulerian cycle of a graph G is a traversal of all edges of the graph, i.e., a path that starts and ends in the same vertex and goes through each edge exactly once. Recall the following basic result:

Theorem 15. *An undirected, connected graph G has an Eulerian cycle. $\iff$ Every vertex in G has even degree.*

[Pic: Example Eulerian cycle]

Consider **Algorithm MST-Tour** (Algorithm 7). The algorithm uses a minimum spanning tree T in the graph K_n with weights given by distances.

- Symmetric distances, so consider K_n as *undirected* graph. Distances are edge weights.
- Duplicating all edges yields a graph $\hat{T}$ with even degree for every vertex.

Algorithm 8: Christofides-Serdyukov Algorithm for Δ-MinTSP

Input: number n of vertices, metric d

Consider the undirected, complete graph K_n with edge weights given by d
$T \leftarrow$ optimal solution to Min-Spanning Tree in (K_n, d)
$U \leftarrow$ set of vertices in T with odd degree
$M \leftarrow$ minimum-distance matching of vertices of U
$\hat{T} \leftarrow T \cup M$
$C_{\hat{T}} \leftarrow$ Eulerian cycle of $\hat{T}$
Shortcut edges of $C_{\hat{T}}$ into TSP tour C
return C

- The Eulerian cycle $C_{\hat{T}}$ of $\hat{T}$ can be obtained using a trivial greedy algorithm.
- Consider edge (u, v) in $C_{\hat{T}}$ leading to a vertex v that has been visited in the cycle before. We shortcut this and the next edge (v, w) of the cycle into an edge (u, w). The total distance of the tour can only decrease, since $d(u, w) \leq d(u, v) + d(v, w)$ by the triangle inequality.
- Applying this adjustment repeatedly, we obtain a tour C that visits every *vertex* once (rather than every edge). The total distance is at most that of the Eulerian cycle.

[Pic: Example run of algorithm, shortcutting step]

Theorem 16. *Algorithm 7 has approximation ratio 2 for* Δ-MinTSP.

Proof. Consider an optimal TSP tour C^*. Let $d(C^*) = \sum_{(u,v)\in C^*} d(u, v)$ be its total distance. Let $d(T) = \sum_{(u,v)\in T} d(u, v)$ be the total distance of T. First, we see that $d(C^*) \geq d(T)$:

- Suppose from C^* we remove the edge (u, v) with smallest distance
- Denote the remaining set of edges by $C_{u,v}$.
- Note: $C_{u,v}$ is a spanning tree of K_n. T is a minimum spanning tree of K_n.
- Hence, $d(C^*) \geq d(C_{u,v}) \geq d(T)$

Now since we duplicate edges, we have $2d(T) = d(\hat{T}) = d(C_{\hat{T}})$. Moreover, by shortcutting $C_{\hat{T}}$ via the triangle inequality, we observed above that $d(C_{\hat{T}}) \geq d(C)$. Overall, this implies

$$d(C) \leq d(C_{\hat{T}}) = 2 \cdot d(T) \leq 2 \cdot d(C^*).$$

□

The ratio of 2 emerges from duplicating all edges of T. The **Christofides-Serdyukov Algorithm** (Algorithm 8) is more careful in contructing a graph with an Eulerian cycle. We only add edges for the vertices U that have odd degee in T. This improves the ratio significantly.

Theorem 17. *Algorithm 8 has approximation ratio 1.5 for* Δ-MinTSP.

Proof. The proof is very similar to the proof of Theorem 16. The main difference is that M increases the total distance of $d(\hat{T})$ by at most $0.5 \cdot d(C^*)$:

- Consider C^*_U, a tour obtained by "shortcutting" C^* between the vertices of U
- Sum of vertex degrees in T is even.
  $\Rightarrow T$ must contain an even number of nodes with odd degree
  $\Rightarrow |U|$ is even.
- C^*_U has an even number of edges
- Split C^*_U into two perfect matchings of U, let M_U be the one with smaller distance
- M_U has total distance $d(M_U) \leq 0.5 \cdot d(C^*_U) \leq 0.5 \cdot d(C^*)$
- M is minimum-distance matching of U, so $d(M) \leq d(M_U)$.

[Pic: Tour C^*_U by shortcutting C^*, split into two matchings]

Combining this insight with the arguments above, we see that

$$d(C) \leq d(C_{\hat{T}}) = d(T) + d(M) \leq d(C^*) + d(M_U) \leq 1.5 \cdot d(C^*)$$

□

The running time of the algorithm is dominated by finding a min-distance matching M among vertices of U. It can be as high as $O(n^3)$. The algorithm was found by independently by Christofides and Serdyukov in 1976 and has been the algorithm with the best approximation ratio for metric TSP for more than 40 years. Very recently, Karlin, Klein, and Oveis Gharan (2021) obtained an algorithm with improved ratio – it beats 1.5 by a constant of ca. 10^{-36}.

There are other algorithms for Δ-MinTSP that iteratively build a solution. The **Nearest-Insertion** heuristic adds in each round a single vertex to an existing subtour: Start with the pair of vertices with smallest distance as subtour. Add in each round a vertex with smallest distance to any vertex in the current subtour, and insert it in a way to increase the total distance as little as possible.

Similarly, the **Farthest-Insertion** heuristic adds in each round a single vertex with largest distance to any vertex in the current subtour. It inserts the vertex in a way to increase the total distance as little as possible.

Nearest-Insertion has approximation ratio at most 2, Farthest-Insertion a ratio between 6.5 and $1 + \log n$.

Finally, many special cases of Δ-MinTSP allow algorithms with better approximation ratios:

- Graph TSP, i.e., $d(u, v)$ is the shortest path distance in an underlying unweighted, undirected graph $G = (V, E)$
  $\rightarrow$ can be approximated with ratio 1.4 (Sebö, Vygen, 2012)
- Planar Graph TSP, i.e., the underlying graph G is planar
  $\rightarrow$ there is a PTAS (Grigni, Koutsoupias, Papadimitriou, 1995)
- Euclidean TSP, i.e., vertices are points in $\mathbb{R}^k$, and distances are measured via the direct connections $d(u, v) = \sqrt{(u_1 - v_1)^2 + \ldots + (u_k - v_k)^2}$
  $\rightarrow$ there is a PTAS (Arora, 1996)

Algorithm 9: Three Greedy Algorithms for Bin Packing

Input: n sizes $g_1, \ldots, g_n \in [0, 1]$

open new bin $b = 1$

`/* Next-Fit: */`

for $i = 1, \ldots, n$ **do**
  if *i fits into bin b* **then** place i into b
  else open new bin $b + 1$, place i into $b + 1$, set $b \leftarrow b + 1$
return b

`/* First-Fit: */`

for $i = 1, \ldots, n$ **do**
  for *bin* $j = 1, \ldots, b$ **do** place item i into bin j if it fits
  if *i does not fit in any open bin* **then**
    open new empty bin $b + 1$, place i into $b + 1$, set $b \leftarrow b + 1$
return b

`/* First-Fit-Decreasing: */`

Sort items in non-increasing order of size: $g_1 \geq g_2 \geq \ldots \geq g_n$
Run First-Fit algorithm with items sorted in this order.

2.5 Bin Packing

Bin Packing is a fundamental optimization problem:

- n items, each item has size $g_i \in [0, 1]$.
- Up to n bins available.
- Size constraint: Each bin can fit items of total size at most 1
- Place all items into bins without violating the size constraints

Goal: Minimize the number of bins that are used

An instance is represented by $g_1, \ldots, g_n$. Let b^* be the smallest number of bins that are required to pack all items. The following algorithms implement different variants of a natural greedy approach (see Algorithm 9). They offer worst-case performance guarantees as follows:

1. **Next-Fit** $\rightarrow$ opens at most $b \leq 2b^* + 1$ bins
2. **First-Fit** $\rightarrow$ opens at most $b \leq 1.7b^* + 2$ bins
3. **First-Fit-Decreasing** $\rightarrow$ opens at most $b \leq \frac{3}{2} \cdot b^* + 1$ bins
   (in fact, even $b \leq \frac{11}{9} \cdot b^* + 2$ can be shown)

Let us first observe the guarantee for NextFit. Suppose b denotes the number of bins opened by the algorithm, and $G = \sum_{i=1}^{n} g_i$.

- We partition the b bins into pairs of consecutively opened bins.
- For every pair, the sum of sizes in the bins is at least 1 (otherwise, the later bin would not have been opened)

- This implies for even b that $b^* \geq G \geq b/2$.
- Similar, when b is odd: $b^* \geq G \geq (b-1)/2$.

Hence $b^* \geq (b-1)/2$, which implies $b \leq 2b^* + 1$.

Next-Fit and First-Fit are online algorithms, they can be applied even if the items arrive sequentially over time and must be placed before seeing the next item(s). Instead, First-Fit-Decreasing is an offline algorithm.

Theorem 18. *First-Fit-Decreasing (FFD) opens a number of b bins with $b \leq \frac{3}{2}b^* + 1$.*

Proof. Partition the items into four classes:

1. $A = \{i \mid g_i > 2/3\}$, large items
2. $B = \{i \mid 2/3 \geq g_i > 1/2\}$, medium-high items
3. $C = \{i \mid 1/2 \geq g_i > 1/3\}$, medium-low items
4. $D = \{i \mid g_i \leq 1/3\}$, small items

Note that FFD first assigns all items from A, then from B, then from C, finally from D.

[Pic: Classes]

Consider the subinstance of items $A \cup B \cup C$ and a run of FFD:

- FFD opens a bin for each item in A (called A-bin), any assignment must do this
- FFD opens a bin for each item in B (called B-bin), any assignment must do this
- At most one item from C can be in a B-bin
- FFD maximizes the number of items from C placed in B-bins ($\rightarrow$ Exercise)
- Remaining C-items are placed in pairs into bins (except maybe last one).

Hence, when FFD has completed the assignment of A, B, and C, it made an *optimal solution* for these items.

[Pic: A-bins, B-bins, items from C in B-bins]

Now consider the remaining items from D assigned in the last phase.

- If FFD opens no new bin, then it has an optimal assignment of all tasks, $b \leq b^*$.
- Suppose FFD opens at least one new bins.
- Then all bins must have load at least $2/3$, except maybe the last one.
- This implies: $b^* \geq G \geq (b-1) \cdot 2/3$
- Solving for b we get $b \leq \frac{3}{2}b^* + 1$

□

The guarantee consists of a multiplicative factor of $3/2$ and an additive offset. Let us first see whether we can obtain a better guarantee without the offset.

Theorem 19. *For any $\varepsilon > 0$, there is no polynomial-time algorithm with approximation ratio $\frac{3}{2} - \varepsilon$ for* Bin Packing*, unless* $\mathsf{P} = \mathsf{NP}$.

Proof. Consider an instance of the NP-complete decision problem Partition:

- n integer numbers $a_1, \ldots, a_n$. We define $A = \sum_{i=1}^{n} a_i$ and $[n] = \{1, \ldots, n\}$.
- We need to decide if there are sets $B_1 \subseteq \{1, \ldots, n\}$ and $B_2 = [n] \setminus B_1$ such that

$$\sum_{i \in B_1} a_i = A/2 = \sum_{i \in B_2} a_i.$$

Algorithm 10: APTAS for Bin Packing

Input: Set $[n]$ of items with sizes $g_1, \ldots, g_n \in [0,1]$, parameter $\varepsilon \in (0,1]$

$S \leftarrow \{i \mid g_i \leq \varepsilon\}$, $B \leftarrow [n] \setminus S$ `// sets of small and big items`

`// Partition` $[0,1]$ `into at most` $k = \lceil 1/\varepsilon^2 \rceil$ `intervals`

Sort items in non-increasing order of size.

Assign first $\lfloor \varepsilon^2 \cdot n \rfloor$ items to the first class

Repeat with next $\lfloor \varepsilon^2 \cdot n \rfloor$ items and second class, and so on, until all items assigned.

`// Solve auxiliary instance`

Auxiliary Size: Round up each item size to the largest one in its class

By enumeration, find optimal bin packing for B when every item has auxiliary size

`// Add small items`

Extend this packing by assigning the items of S using First-Fit

return bin packing of all items

Given an instance of Partition we construct an instance of Bin Packing with n items by setting $g_i = 2a_i/A$. Then $G = \sum_i g_i = 2$. Clearly,

- The optimum packing needs at least two bins.
- Partition instance is true
  $\iff$ Sets B_1 and B_2 describe numbers that sum to $A/2$, respectively
  $\iff$ Items from B_1 in the Bin Packing instance have total size $G/2 = 1$, same for B_2.
  $\iff$ Bin Packing instance has $b^* = 2$.

This shows that finding an optimal solution for Bin Packing in this class of instances is NP-hard. What about a $(3/2 - \varepsilon)$-approximation algorithm?

Suppose we have such an algorithm and run it on these instances. If the $b^* = 2$, the algorithm must return a solution with at most $(3/2 - \varepsilon) \cdot 2 < 3$ bins, so with 2 bins! Thus, with a $(3/2-\varepsilon)$-approximation algorithm for Bin Packing, we can correctly decide Partition. □

The hardness reduction applies only to instances with *small optima*. Note that for $b^* = 2$, an approximation ratio of $3/2$ allows $3/2 \cdot b^* = 3 = b^* + 1$ bins, i.e., it is equivalent to an additive offset of 1. In contrast to problems like Makespan Scheduling, the hardness of the Bin Packing problem does not scale to larger instances.

We examine if we can obtain a better guarantee when we accept an additive offset of 1. Every suboptimal solution for Bin Packing needs at least $b^* + 1$ bins. It is still unknown if there is a polynomial-time algorithm that always returns a solution with $b^* + 1$ bins. Here we give a slightly weaker result: A so-called **asymptotic PTAS (APTAS)**, a PTAS with an additive constant in the approximation guarantee (Algorithm 10)

Theorem 20. *For every constant $\varepsilon > 0$, Algorithm 10 is an APTAS for* Bin Packing. *It computes a solution in time polynomial in the input size (but exponential in $1/\varepsilon$) that opens*

only b bins with

$$b \leq (1 + 2\varepsilon) \cdot b^* + 1.$$

The APTAS divides the items into relevant (set B) and not-so-relevant (set S). For relevant items, it partitions the domain $[0, 1]$ into classes of roughly similar sizes – it uses $k \approx 1/\varepsilon^2$ intervals, each of which contains at most $n/k \approx \varepsilon^2 \cdot n$ items of B. If several items have the same size, they count as separate items. It then uses the **auxiliary size** for each item – it rounds every item size to the largest one of its class.

[Pic: Interval division]

The following step is the main one: We enumerate and optimize over all bin packings for the same instance, where every item size is replaced by its auxiliary size. How long does this take?

Lemma 4. *There are at most*

$$\binom{d+n}{d} \in O(n^d)$$

many bin packings with auxiliary item sizes that open a different numbers of bins, where

$$d = \binom{\lceil 1/\varepsilon^2 \rceil + \lfloor 1/\varepsilon \rfloor}{\lceil 1/\varepsilon^2 \rceil} \in O\left(\left(\frac{1}{\varepsilon}\right)^{1/\varepsilon^2}\right) .$$

Proof. We use that there is only a constant number of different auxiliary item sizes.

- There are only $k = \lceil 1/\varepsilon^2 \rceil$ possible auxiliary item sizes, one for each class.
- All items have auxiliary size at least ε.
- Hence, each bin can contain at most $r = \lfloor 1/\varepsilon \rfloor$ many items.
- A **bin type** is a vector $(a_1, \ldots, a_k)$, where a_i indicates the number of items from class i in the bin. How many different types of bins can we have?

Let us bound the number of different types of bins.

- A bin type is a vector of k non-negative integer numbers that sum to at most r, i.e., we count the number of vectors $(a_1, \ldots, a_k)$, such that each $a_i \in \mathbb{N}$ and $\sum_{i=1}^{k} a_i \leq r$.
- Given a type, let $t = \sum_{i=1}^{k} a_i$. We represent the type in unary coding with $k - 1 + t$ bits: Exactly t 1's, with a 0 separating classes (e.g., (4,2,3) = 11110110111)
- The number of such vectors is bounded by $\binom{k-1+t}{k-1}$ – the possible choices for the $k - 1$ 0's in the vector of $k - 1 + t$ bits
- In total the number of different types of bins is at most

$$d = \sum_{t=0}^{r} \binom{k-1+t}{k-1} = \sum_{T=k-1}^{k-1+r} \binom{T}{k-1} = \binom{k+r}{k} = \binom{\lceil 1/\varepsilon^2 \rceil + \lfloor 1/\varepsilon \rfloor}{\lceil 1/\varepsilon^2 \rceil} .$$

  where we use k and r as defined above.
- Since ε is a constant, d is a constant!

How can we count the number of possible bin packings?

- We have at most d many bin types.
- A bin packing can be seen as a vector $(b_1, \ldots, b_d)$, where b_i indicates the number of bins of type i in the packing.
- We count[1] the number of possible bin packings as above: The number of vectors with d non-negative integers such that they sum up to n.
- By repeating the analysis above, this results in at most

$$\binom{d+n}{d}$$

many bin packings that open a different number of bins.

□

When we enumerate the number of different bin packings with auxiliary item sizes, this dominates the running time of the algorithm, but it remains polynomial in n for constant ε. How can we bound the approximation factor?

Lemma 5. *The number b of bins opened by Algorithm 10 is $b \leq (1+2\varepsilon) \cdot b^* + 1$.*

Proof. Case 1: $B = [n]$ and $S = \emptyset$, no items with size at most ε.

- Let b^* be the number of bins opened in the optimal packing. Since every item has size at least ε, we see that

$$b^* \geq \sum_{i=1}^{n} g_i \geq \varepsilon \cdot n$$

- Consider an optimal packing for items with auxiliary sizes, where we *exclude the items of the largest class*
- This packing uses at most b^* items:
  Take optimal packing with original sizes. Replace $\lfloor \varepsilon^2 \cdot n \rfloor$ items of largest class with $\lfloor \varepsilon^2 \cdot n \rfloor$ rounded up items of second-largest class, and so on. Packing remains feasible with b^* bins, since replacement decreases every item size. Optimal packing uses at most b^* bins.
- Every item from the largest class opens at most one more bin.
- Hence, the algorithm opens b bins with

$$b \;\leq\; b^* + \lfloor \varepsilon^2 \cdot n \rfloor \;\leq\; b^* + \lfloor \varepsilon \cdot b^* \rfloor \;\leq\; (1+\varepsilon) \cdot b^*$$

Case 2: $S \neq \emptyset$.

- Let b^*_B be the number of bins opened in the optimal packing *of only the items in B*
- Just proved in Case 1: APTAS determines a packing of items in B with at most $(1+\varepsilon)b^*_B \leq (1+\varepsilon)b^*$ bins.
- The items in S are then assigned using FirstFit.

[1] In fact, this is an upper bound on the number of possible bin packings, since there are additional constraints coming from the number of auxiliary sizes of items that are not taken into account here. For example, there might not be n bins of the same type j possible, since the instance might not have enough items to fill all n bins with the items required according to type j.

- If no additional bin is opened, previous bound applies, $b \leq (1+\varepsilon)b^*$.
- Otherwise, every bin – except maybe the last one – has total size of at least $1-\varepsilon$.
- Hence, the total number b of open bins satisfies

$$b^* \geq \sum_{i=1}^{n} g_i \geq (b-1) \cdot (1-\varepsilon),$$

which implies

$$b \;\leq\; \frac{1}{1-\varepsilon} \cdot b^* + 1 \;\leq\; (1+2\varepsilon) \cdot b^* + 1.$$

□

2.6 Maximum Coverage

The Maximum Coverage problem is a basic problem with many applications. It is very closely related to the Set Cover problem that we will analyze in Section 5.3.3.

- Set $E = \{1, \ldots, m\}$ of elements, integer $k \leq n$
- Set system $\mathcal{S} = \{S_1, \ldots, S_n\} \subseteq 2^E$, i.e., a family of n subsets of E.
- Collection $C \subseteq [n]$ of subsets *covers* the elements $E(C) = \bigcup_{i \in C} S_i \subseteq E$
- Number of elements covered by C is denoted by $cov(C) = |E(C)|$
- **Goal:** Find a maximum cover, i.e., C with at most $|C| \leq k$ sets that maximizes $cov(C)$

Example: Elements $E = \{a, \ldots, j\}$, sets

- $S_1 = \{c, d, e, h\}$
- $S_2 = \{a\}$
- $S_3 = \{a, c, e, g, i\}$
- $S_4 = \{b, d, j\}$
- $S_5 = \{b, c, d, e, f\}$
- $S_6 = \{b, f, j\}$

For $k = 1$, maximum covers are $C = \{3\}$ and $C' = \{5\}$ with $cov(C) = cov(C') = 5$.
For $k = 2$, a maximum cover is $C = \{3, 5\}$ with $cov(C) = 8$. Other maximum covers for $k = 2$ are $C' = \{3, 6\}$ or $C'' = \{3, 4\}$.

Variants and generalizations of the Maximum Coverage problems have lots of applications in various fields like *viral marketing* or *machine learning*. Here is a (toy) example application: Suppose the elements E are potential customers of a product. There are n influencers, and each comes with a subset $S_i \subset E$ of followers. A company has a budget to hire up to $k \leq n$ of the influencers to advertise its product on their channel. If an influencer advertises the product, then all her followers will buy it (but each follower buys at most once). Which k influencers should the company hire to maximize the total number of customers that buy the product?

We apply a **Greedy algorithm** (Algorithm 11). In each iteration t, it picks a set S_{j_t} that maximizes $cov(C \cup \{j_t\}) - cov(C) = |S_{j_t} \setminus E(C)|$, the (marginal) increase in the number of covered elements.

Algorithm 11: Greedy Algorithm for MAXIMUM COVERAGE

Input: Set $\mathcal{S}$ of n subsets, parameter $k \leq n$

$C_1 \leftarrow \emptyset$
for $t = 1, \ldots, k$ **do**
    `// maximize marginal increase in number of covered elements`
    $j_t \leftarrow \arg\max_{i \in [n] \setminus C} cov(C \cup \{i\}) - cov(C)$
    $C_{t+1} \leftarrow C_t \cup \{j_t\}$
return C_{k+1}

Algorithm 12: Maximize Submodular Function s.t. Cardinality Constraint

Input: Set $[n]$ of n options, submodular function f, parameter $k \leq n$

$C_1 \leftarrow \emptyset$
for $t = 1, \ldots, k$ **do**
    `// maximize marginal increase in objective function` f
    $j_t \leftarrow \arg\max_{i \in [n] \setminus C} f(C \cup \{i\}) - f(C)$
    $C_{t+1} \leftarrow C_t \cup \{j_t\}$
return C_{k+1}

How good is this algorithm? We will answer this question for a general selection problem with **decreasing marginal returns**.

Definition 13. *Given a set $[n]$ of n options, a* set function $f : 2^{[n]} \to \mathbb{R}$ *is called*

- ***normalized*** *if*
$$f(\emptyset) = 0$$
- ***monotone*** *if*
$$f(A) \leq f(B) \qquad \text{for each } A \subseteq B \subseteq [n]$$
- ***submodular*** *if we have decreasing marginal returns*
$$f(B \cup \{j\}) - f(B) \;\leq\; f(A \cup \{j\}) - f(A) \qquad \text{for each } A \subseteq B \subseteq [n], j \in [n]$$

  Submodular is sometimes defined by the alternative property
$$f(A \cup B) \;\leq\; f(A) + f(B) - f(A \cap B) \qquad \text{for each } A, B \subseteq [n].$$

Both definitions are equivalent (Exercise).

Note that the objective function $cov : 2^{[n]} \to \mathbb{N}$ of the MAXIMUM COVERAGE problem
$$cov(C) = |E(C)| = \left| \bigcup_{i \in C} S_i \right|$$
satisfies all three properties. Normalized and monotone are trivial. For submodular, suppose we add a set S_j to a cover B, and consider the marginal increase in the number of covered

elements. Now compare this to the marginal increase when adding S_j to a cover A, for some subset $A \subseteq B$. Clearly, A covers only a subset of the elements that B covers. Hence, $E(A) \subseteq E(B)$, and the marginal increase is:

$$\begin{aligned} cov(B \cup \{j\}) - cov(B) &= |E(B \cup \{j\})| - |E(B)| \\ &= |S_j \setminus E(B)| \\ &\leq |S_j \setminus E(A)| && A \subseteq B \Rightarrow E(A) \subseteq E(B) \\ &= |E(A \cup \{j\})| - |E(A)| \\ &= cov(A \cup \{j\}) - cov(A) \end{aligned}$$

[Pic: *cov* is submodular]

Consider the Constrained Submodular Maximization problem: Choose a subset $C \subseteq [n]$ to maximize a given normalized, monotone, and submodular set function $f(C)$ subject to a cardinality constraint $|C| \leq k$. This is a generalization of the Maximum Coverage problem. We apply a greedy algorithm (Algorithm 12) to solve this problem.

Theorem 21. *Algorithm 12 has an approximation ratio of* $\mathsf{e}/(\mathsf{e}-1) \approx 1.58$. *It runs in polynomial time if* $f(C)$ *can be computed in polynomial time for all* $C \subseteq [n]$.

This ratio is essentially best possible, even in the special case of Maximum Coverage: If there is a polynomial-time algorithm that computes an $(\mathsf{e}/(\mathsf{e}-1) - \varepsilon)$-approximation for some constant $\varepsilon > 0$, then $\mathsf{P} = \mathsf{NP}$.

The statement about the running time is trivial, so we only show the approximation ratio. We start with a technical lemma.

Lemma 6. *Suppose* C^* *is an optimal solution, i.e., a subset of* k *options that maximizes* $f(C^*)$. *In every iteration* $t = 1, \ldots, k$, *the increase in solution value is at least*

$$f(C_{t+1}) - f(C_t) \;\geq\; \frac{1}{k} \cdot (f(C^*) - f(C_t)).$$

Proof. Consider $C^* \setminus C_t = \{j_1^*, j_2^*, \ldots, j_p^*\}$ the options in C^* that are not in C_t. Then

$$\begin{aligned} & f(C^*) - f(C_t) \\ & \quad \leq f(C_t \cup C^*) - f(C_t) && \text{(by monotonicity)} \\ & \quad = f(C_t \cup \{j_1^*\}) - f(C_t) \;+\; f(C_t \cup \{j_2^*, j_1^*\}) - f(C_t \cup \{j_1^*\}) \\ & \qquad + \ldots \\ & \qquad + f(\underbrace{C_t \cup \{j_p^*, j_{p-1}^*, \ldots, j_1^*\}}_{= C_t \cup C^*}) - f(C_t \cup \{j_{p-1}^*, \ldots, j_1^*\}) && \text{(telescoping sum)} \\ & \quad \leq \sum_{i=1}^{p} f(C_t \cup \{j_i^*\}) - f(C_t) && \text{(by submodularity)} \\ & \quad \leq p \cdot (f(C_t \cup \{j_t\}) - f(C_t)) && \text{(by greedy choice)} \end{aligned}$$

$$\begin{aligned}
&\leq k \cdot (f(C_t \cup \{j_t\}) - f(C_t)) && \text{(since } p = |C^* \setminus C_t| \leq k\text{)} \\
&= k \cdot (f(C_{t+1}) - f(C_t)) && \text{(since } C_{t+1} = C_t \cup \{j_t\}\text{)}
\end{aligned}$$

□

[Pic: $C^* \setminus C_t$, telescoping sum, submodularity for individual terms w.r.t. C_t]

We now proceed to show the main result by induction.

Proof of Theorem 21. For every $t = 1, \ldots, k$, we show that at the end of iteration t

$$f(C_{t+1}) \geq \left(1 - \left(1 - \frac{1}{k}\right)^t\right) \cdot f(C^*).$$

- Base: We have $C_1 = \emptyset$ and $f(C_1) = 0$. Using Lemma 6,

$$f(C_2) = f(C_2) - f(C_1) \geq \frac{1}{k} \cdot (f(C^*) - f(C_1)) = \frac{1}{k} \cdot f(C^*) = \left(1 - \left(1 - \frac{1}{k}\right)\right) \cdot f(C^*)$$

- Hypothesis: Statement holds for some C_t with $2 \leq t \leq k + 1$.
- Step: Consider C_{t+1}:

$$\begin{aligned}
f(C_{t+1}) &\geq f(C_t) + \frac{1}{k} \cdot (f(C^*) - f(C_t)) && \text{(by Lemma 6)} \\
&= \frac{1}{k} \cdot f(C^*) + \left(1 - \frac{1}{k}\right) \cdot f(C_t) \\
&\geq \frac{1}{k} \cdot f(C^*) + \left(1 - \frac{1}{k}\right) \cdot \left(1 - \left(1 - \frac{1}{k}\right)^{t-1}\right) \cdot f(C^*) && \text{(by hypothesis)} \\
&= \left(\frac{1}{k} + 1 - \frac{1}{k} - \left(1 - \frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{t-1}\right) \cdot f(C^*) \\
&= \left(1 - \left(1 - \frac{1}{k}\right)^t\right) \cdot f(C^*)
\end{aligned}$$

This completes the induction. In the end,

$$f(C_{k+1}) \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot f(C^*) \geq \left(1 - \frac{1}{\mathsf{e}}\right) \cdot f(C^*),$$

i.e., the approximation ratio is at most $1/(1 - 1/\mathsf{e}) = \mathsf{e}/(\mathsf{e} - 1) \approx 1.58$. □

2.7 Shortest Common Superstring

2.7.1 Greedy Conjecture

A *string* is a sequence of letters from a finite alphabet Σ. The Shortest Common Superstring problem is given as follows.

- Set U with n strings over some finite alphabet Σ
- A *superstring* of U is a single string, where each string of U occurs as a substring.
- W.l.o.g. no string $u \in U$ is substring of another $u' \in U$ – otherwise simply drop u.
- **Goal**: Find a superstring S that has smallest total length $|S|$

Example Application: Shotgun Sequencing of DNA

- Sequential pass over DNA for sequencing is too costly and time-consuming
- Instead: Produce many fragments (substrings) of a DNA in parallel
- Fragments are almost disjoint
- How to recover the common DNA sequence from the fragments?
- Shortest superstring often solves the problem

Some definitions:

- $\text{overlap}(u, v) \in \mathbb{N}$ is *length* of longest suffix of u that is a prefix of v
- $\text{prefix}(u, v)$ is the prefix *string* of u not contained in v
- Hence: $|\text{prefix}(u, v)| = |u| - \text{overlap}(u, v)$
- We also define $\text{overlap}(u, u)$ via the longest *"non-trivial"* overlap of u with itself – the longest overlap with **length strictly smaller than** $|u|$. So $0 \leq \text{overlap}(u, u) < |u|$.

Superstrings and orderings:

- A bijection $\pi : \{1, \ldots, n\} \to U$ defines an ordering of the strings in U.
- Let $(\pi(1), \ldots, \pi(n))$ be the strings in the order of π.
- π induces a shortest superstring $S(\pi)$

$$S(\pi) = \text{prefix}(\pi(1), \pi(2)) \oplus \text{prefix}(\pi(2), \pi(3)) \oplus \cdots \oplus \text{prefix}(\pi(n-1), \pi(n)) \oplus \pi(n)$$

  where we use $\oplus$ for concatenation of two strings.

[Pic: prefix and overlap, ordering of strings induces a superstring]

These definitions imply several direct observations.

1. If S is a *shortest* superstring of U, then there is an ordering π such that $S = S(\pi)$.
2. The length $|S(\pi)|$ is given by

$$|S(\pi)| = \sum_{u \in U} |u| - \sum_{i=1}^{n-1} \text{overlap}(\pi(i), \pi(i+1))$$

To see observation 1., suppose S is a shortest superstring of U. Place every $u \in U$ at the position where it first occurs in S. Every letter of S must be covered by some $u \in U$, otherwise we can shorten S. Hence, S is a concatenation of prefixes from strings in U in some ordering. The overlap of subsequent strings is always as large as possible, since otherwise we could further shorten the string.

[Pic: String S, strings from U, overlap]

To see observation 2., note that

$$|S(\pi)| = \sum_{i=1}^{n-1} |\text{prefix}(\pi(i), \pi(i+1))| + |\pi(n)|$$

Algorithm 13: Greedy Superstring

Input: Set U of n strings

$R \leftarrow U$
while $|R| > 1$ **do**
  find two strings $u \neq v$ from R with maximal overlap(u, v)
  $w \leftarrow \text{prefix}(u, v) \oplus v$
  Replace u and v by w: $R \leftarrow ((R \setminus \{u, v\}) \cup \{w\})$
return single remaining $S \in R$

$$= \sum_{i=1}^{n-1} (|\pi(i)| - \text{overlap}(\pi(i), \pi(i+1))) + |\pi(n)|$$
$$= \sum_{i=1}^{n} |\pi(i)| - \sum_{i=1}^{n-1} \text{overlap}(\pi(i), \pi(i+1))) + |\pi(n)|$$

Finding a shortest superstring is equivalent to finding an **ordering of strings** in U that **maximizes the sum of overlaps** over all pairs of consecutive strings. First, consider the simple greedy algorithm (Algorithm 13). It merges two strings that have the largest overlap until a single string remains.

Seminal result in the literature (Blum et al, 1991):

Theorem 22. *Algorithm 13 has approximation ratio 4 for* Shortest Common Superstring.

Recent updates (Englert et al, Nov 2021):
- Greedy has approximation ratio at most $(13 + \sqrt{57})/6 < 3.425$.
- Poly-time algorithm with best-known ratio of $(37 + \sqrt{57})/18 < 2.475$

Major open conjecture: The approximation ratio of greedy is actually 2.

Instead of the greedy algorithm, we here study a different algorithm related to cycle covers.

2.7.2 Cycle Covers

Based on U, we construct a directed **overlap graph** G:
- Nodes are strings. A directed edge in E for every pair $u, v \in U$ (also for $u = v$)
- Complete directed graph over the strings, even with a loop for every string
- $(u, v) \in E$ has edge weight overlap(u, v)

[Pic: Example graph, overlap is edge weight]

Short superstrings are orderings are Hamiltonian paths:
- In a Hamiltonian path of G each node visited exactly once

Algorithm 14: Max-Weight Cycle Decomposition

Input: Set U of n strings

Construct overlap graph $G = (U, E)$, compute overlap(u, v) for all $u, v \in U$
$E' \leftarrow \emptyset$
repeat
  Pick $(u, v) \in E$ with largest weight overlap(u, v)
  $E' \leftarrow E' \cup \{(u, v)\}$
  $E \leftarrow E \setminus \{(u, w) \in E \mid w \in U\}$
  $E \leftarrow E \setminus \{(w, v) \in E \mid w \in U\}$
until $E = \emptyset$
return E'

- Ordering of strings is a Hamiltonian path, and vice versa.
- Total edge weight of the path is total overlap of strings
- Max-weight path has maximal overlap, hence, is best ordering, and shortest superstring
- Shortest superstring corresponds to a (max-weight) Hamiltonian path

In general, finding a max-weight directed Hamiltonian path is NP-hard. We here apply a weaker solution: a **directed cycle decomposition** of maximum weight.

- Directed cycle decomposition of $G = (U, E)$ is a subgraph (U, E')
- In (U, E') every node has exactly one outgoing and one incoming edge
- Hence, every component of (U, E') is a directed cycle or a single node with self-loop.
- Weight of cycle cover is total edge weight $\sum_{(u,v)\in E'} \text{overlap}(u, v)$

[Pic: Example Cycle Decomposition]

We proceed as follows:

1. Compute a max-weight cycle decomposition
2. Show how to build a superstring from a cycle decomposition
3. Bound the resulting approximation ratio

Computing a Max-Weight Cycle Decomposition. We use the greedy algorithm presented in Algorithm 14. Greedy computes an optimal decomposition for weights given by overlap(u, v) of strings $u, v \in U$. It is not necessarily optimal for general edge weights (a more advanced method based on matchings can be applied).

Theorem 23. *Algorithm 14 returns a cycle decomposition of maximal weight for every overlap graph.*

Towards the proof of the result, we first show the following structural lemma.

Lemma 7. *Consider any four strings u, v, u^*, v^*. If*

$$\textit{overlap}(u, v) \geq \textit{overlap}(u, v^*) \qquad \textit{and} \qquad \textit{overlap}(u, v) \geq \textit{overlap}(u^*, v)$$

then

$$\textit{overlap}(u, v) + \textit{overlap}(u^*, v^*) \;\geq\; \textit{overlap}(u^*, v) + \textit{overlap}(u, v^*).$$

Algorithm 15: Construct Superstring

Input: Set U of n strings

$E' \leftarrow$ cycle decomposition returned by Algorithm 14
for *each cycle* C_k *in* E' **do**
    Pick arbitrary string of C_k as u_1
    Construct string S_k by merging all strings along the cycle from u_1
$S \leftarrow \bigoplus_{C_k} S_k$
return S

Proof. Exercise. □

Proof of Theorem 23. By induction over $n = |U|$.

Base: $n = 1$, single self-loop, trivial.

Step: Suppose the algorithm is optimal for n nodes/strings. Consider $n + 1$ nodes/strings.

- Suppose $e_1 = (u, v)$ is edge with largest overlap(u, v) ($u = v$ possible)
- Suppose E^* is a max-weight cycle decomposition. If E^* includes e_1, great!
- Otherwise, there is are edges $(u, v^*), (u^*, v) \in E^*$ (leaving u and coming into v). Again $u = u^*$, $v = v^*$, etc. possible.
- Since e_1 has largest weight, we can apply the lemma to see

$$\text{overlap}(u, v) + \text{overlap}(u^*, v^*) \;\geq\; \text{overlap}(u^*, v) + \text{overlap}(u, v^*).$$

- Replace (u, v^*) and (u^*, v) by (u, v) and (u^*, v^*) in E^*
- New cycle decomposition, total weight can only increase → max-weight decomposition

→ There is a max-weight cycle decomposition that includes e_1.

The algorithm makes no mistake when it picks e_1 in the first step. Consider the next steps.

- Merge nodes u and v into a supernode.
- Keep incoming edges of u and outgoing edges of v (delete the others).
- This is equivalent to merging the strings u and v to a common superstring.
- It creates an overlap graph G' with n nodes.
- Cycle decomposition of $G' \rightarrow$ With e_1 becomes cycle decomposition of G
- Cycle decomposition of G with $e_1 \rightarrow$ cycle decomposition of G'.
- Hence, use greedy to compute an optimal cycle decomposition E' of G' (by induction)
- $E' \cup \{e_1\}$ is optimal cycle decomposition of G

□

[Pic: Node merge, correspondence of decompositions in G' and G]

Superstring from a Cycle Decomposition. Let us turn to the second step. Suppose we have computed a max-weight cycle decomposition. Given a cycle decomposition, we define for each cycle C_k a superstring S_k that contains the strings from the cycle. We then concatenate the superstrings S_k of all cycles C_k. (Algorithm 15)

Consider a single cycle C with strings $u_1, \ldots, u_m$ (numbered consecutively along the cycle)

- We pick an arbitrary string from C_k and define it as u_1
- Then construct two strings along the cycle starting from u_1:
  S_k results from merging strings $u_1, \ldots, u_m$ in that order
  v_k is the part of S_k from u_1 until the second start u_1
- v_k is a prefix of $S_k \oplus u_1$, so
$$|S_k| \leq |v_k| + |u_1|$$
- Possibly $|v_k| < |u_i|$ for some u_i. In fact, even $|v_k| < |u_i|$ for all $1 \leq i \leq m$ is possible.

[Pic: Strings, overlap, construction of S_k and v_k]

Consider the infinite concatenation of $v_k^\infty = v_k \oplus v_k \oplus v_k \oplus \ldots$
- v_k^∞ contains all strings $u_1, \ldots, u_m$ (even the ones with $|u_i| > |v_k|$)
- We say v_k is a **cycle cover** of strings $u_1, \ldots, u_m$. More generally, the collection of all $\{v_k \mid C_k \text{ cycle in } E'\}$ is a **cycle cover** of U, since every string of U occurs in the cyclic extension of at least one v_k.
- If another string of C_k is chosen as u_1, then v_k gets shifted cyclically.
- To obtain S_k, we can concatenate v_k until it contains all u_i and then break off the remaining letters.

Approximation Ratio. To bound the approximation ratio, we use the insights on v_k and S_k for each cycle, along with the property that E' is a max-weight cycle decomposition.

Theorem 24. *Algorithm 15 has approximation ratio 4 for the* Shortest Common Superstring *problem.*

Proof. Consider the cycle cover v_k of cycle C_k with strings $u_1, \ldots, u_m$.
- v_k is given by
$$\text{prefix}(u_1, u_2) \oplus \text{prefix}(u_2, u_3) \oplus \cdots \oplus \text{prefix}(u_{m-1}, u_m) \oplus \text{prefix}(u_m, u_1)$$
with a length of
$$\begin{aligned} |v_k| &= \sum_{i=1}^{m} |\text{prefix}(u_i, u_{i+1})| \qquad \text{(where we interpret } m+1 = 1 \bmod m\text{)} \\ &= \sum_{i=1}^{m} |u_i| - \text{overlap}(u_i, u_{i+1}) \qquad \text{(indices mod } m\text{)} \end{aligned}$$
- Clearly, by summing over all cycles $C_1, \ldots, C_\ell$ in the decomposition E'
$$\begin{aligned} \sum_{k=1}^{\ell} |v_k| &= \sum_{u \in U} |u| - \sum_{(u,v) \in E'} \text{overlap}(u, v) \\ &= \sum_{u \in U} |u| - \text{max weight of cycle decomposition} \end{aligned}$$

- We observed above that an optimal superstring $S^* = S(\pi^*)$ corresponds to a max-weight Hamiltonian path in the overlap graph, and

$$|S^*| = \sum_{u \in U} |u| - \text{max weight of Hamiltonian path}$$

- Add an edge to Hamiltonian path, turns into a Hamiltonian cycle, weight increases. Hamiltonian cycle is a cycle decomposition.
- Hence, the max-weight cycle decomposition only has more weight than the max-weight Hamiltonian path:

$$\text{max weight of Hamiltonian path} \leq \text{max weight of cycle decomposition}$$

- As a consequence:

$$|S^*| \geq \sum_{k=1}^{\ell} |v_k|$$

Hence, the concatenation of v_k yields a lower bound on the optimal length. Let us now upper bound the lengths of strings $S_1, \ldots, S_\ell$ whose concatenation is returned by our algorithm.

- First consider a special case: U has **non-periodic strings**.
- Non-periodic: In every cycle C_k there is a string u_i^k with $|v_k| > |u_i^k|$
- For the length

$$|S_k| \leq |v_k| + |u_i^k| \leq 2 \cdot |v_k|$$

  and hence $\sum_{k=1}^{\ell} |S_k| \leq 2 \cdot \sum_{k=1}^{\ell} |v_k| \leq 2 \cdot |S^*|$
- This even implies an approximation factor of 2.

- For arbitrary strings, let u_1^k be the first string from cycle C_k, for $k = 1, \ldots, \ell$
- Consider the set $U_1 = \{u_1^1, u_1^2, \ldots, u_1^\ell\} \subseteq U$ of "first" strings
- Let S_1 be a shortest superstring of U_1, and renumber the strings from U_1 in the order they appear in S_1
- We can lower bound the length of the optimal string S^* by

$$\begin{aligned}
|S^*| &\geq |S_1| && \text{(since } U_1 \subseteq U\text{)} \\
&= \sum_{k=1}^{\ell} |u_1^k| - \sum_{k=1}^{\ell-1} \text{overlap}(u_1^k, u_1^{k+1}) && \text{(numbered by ordering in } S_1\text{)} \\
&\geq \sum_{k=1}^{\ell} |u_1^k| - \sum_{k=1}^{\ell-1} (|v_k| + |v_{k+1}|) && \text{(by Lemma 8, see below)} \\
&\geq \sum_{k=1}^{\ell} |u_1^k| - \sum_{k=1}^{\ell} 2|v_k|.
\end{aligned}$$

Therefore

$$\sum_{k=1}^{\ell} |u_1^k| \leq |S^*| + 2 \sum_{k=1}^{\ell-1} |v_k| \ ,$$

and the approximation ratio is bounded by

$$|S| \leq \sum_{k=1}^{\ell} |S_k| \leq \sum_{k=1}^{\ell} |v_k| + |u_1^k| \leq |S^*| + 3\sum_{k=1}^{\ell} |v_k| \leq 4|S^*|$$

□

The proof uses an "Overlap-Lemma". Since our algorithm simply concatenates all S_k (without merge), it should be guaranteed that these strings cannot be merged to a large degree in S^*. This is the consequence of the following lemma, which we simply state here without proof.

Lemma 8. *If u_1^k has cycle cover v_k and u_1^{k+1} has cycle cover v_{k+1} (and v_k is not a cyclic shift of v_{k+1}), then*

$$\mathit{overlap}(u_1^k, u_1^{k+1}) \leq |v_k| + |v_{k+1}|.$$

Chapter 3

Dynamic Programming

In an abstract sense, a dynamic programming algorithm splits a problem P into subproblems $P_1, \ldots, P_k$. Based on the solutions for the subproblems, it can construct a solution for problem P. To obtain a solution for the subproblems it usually applies the split operation recursively, thereby creating further (hopefully smaller and easier) subproblems down the line. The challenge is to manange

- Dependencies: Ensure that subproblems created during the recursive split do not create cyclic requirements for their solutions
- Numbers: The number of subproblems that must be solved to complete the recursive application remains polynomial in the input size.

3.1 Weighted Interval Scheduling

We consider a weighted version of the Interval Scheduling problem:

- n tasks, each with a release date $r_i > 0$ and a deadline $d_i > r_i > 0$.
- Subset of intervals S is **non-overlapping** if for every pair $i, j \in S$, we have $r_i \geq d_j$ or $d_i \leq r_j$.
- **Weighted** version: Each task has a weight (= profit) $w_i \geq 0$.
- **Goal**: Find a set of non-overlapping intervals that maximizes the sum of task weights.

Consider the dynamic programming algorithm in Algorithm 16. It constructs $2n$ subproblems, one for every value r_i and d_i of release dates and deadlines. We assume w.l.o.g. that all times are distinct. Let $t_1, \ldots, t_{2n}$ be the sequence of all release dates and deadlines, sorted in non-decreasing order.

- Subproblem: For a time $t_j \in T$, we consider the optimal weight of a solution if we exclude all tasks i with deadline $d_i > t_j$.
- $W(t_j)$ is the optimal weight in subproblem t_j
- Base case: Clearly, $W(t_1) = 0$, since we exclude all tasks.
- How can we solve the subproblem for t_j using the result for smaller subproblems?

We distinguish two cases depending on whether t_j is a release date or a deadline.

- $t_j = r_k$, release date of some task k. Between t_j and t_{j-1} there is no deadline, subproblem the same as for t_{j-1}.

Algorithm 16: Dynamic Program for Weighted Interval Scheduling

Input: Set $[n]$ of tasks with r_i, d_i and w_i for each $i \in [n]$

`// Assume w.l.o.g. that all` r_i `and` d_i `are distinct`

$T \leftarrow \{r_i, d_i \mid i \in [n]\}$, the set of all release dates and deadlines

Consider $t_1 \leq t_2 \leq \ldots \leq t_{2n}$ all times in T sorted in non-decreasing order

Set $W(t_1) \leftarrow 0$

for $j = 2, 3, \ldots, 2n$ **do**

  if t_j *release date* **then** $W(t_j) \leftarrow W(t_{j-1})$

  else

    $k \leftarrow$ task corresponding to deadline $t_j = d_k$

    $W(t_j) = \max\{W(t_{j-1}), W(r_k) + w_i\}$

return $W(t_{2n})$

- Now suppose $t_j = d_k$, deadline of some task k:
- Suppose $k \in S_j^*$, an optimal solution to subproblem t_j. Then $S_j^* = S_i^* \cup \{k\}$, where S_i^* is an optimal solution for subproblem $t_i = r_k < t_j$.
- Otherwise $j \notin S_j^*$ for every optimal solution to subproblem t_j. Then $S_i^* = S_{i-1}^*$.

[Pic: Intervals, Subproblem, Decomposition of optimal solution if time is a deadline]

Theorem 25. *Algorithm 16 computes the total weight of an optimal solution for any instance of Weighted* Interval Scheduling.

Note that the running time is dominated by the sorting step, which in general can take $\Theta(n \log n)$ time. The for-loop of the dynamic program runs in time $O(n)$.

3.2 Knapsack

The Knapsack problem is a very central packing problem:

- Set $[n]$ of n items, each item i has size $g_i \geq 0$ and weight/profit $w_i \geq 0$
- Knapsack has size bound of G. W.l.o.g. $g_i \leq G$ for every $i \in [n]$
- A subset $S \subseteq [n]$ is *feasible* if it fits into the knapsack: $\sum_{i \in S} g_i \leq G$
- **Goal**: Find a feasible set that maximizes the sum of weights.

Since the basic computational model underlying our considerations is the standard Turing machine, standard normalization tricks allow us to assume that all numbers are given by **non-negative integers**. In fact, the following argument even work with real-valued sizes, but we exploit that weights are integers.

Theorem 26. *Algorithm 17 is an FPTAS for* Knapsack *that runs in time* $O(n^3/\varepsilon)$

We define $W = \sum_{i=1}^{n} w_i$ as the sum of all weights. We first describe a dynamic program for Knapsack. Consider the subproblems:

Algorithm 17: FPTAS for KNAPSACK

Input: Set $[n]$ of tasks with s_i, w_i for each $i \in [n]$, parameter $\varepsilon > 0$

$w_{\max} \leftarrow \max_{i \in [n]} w_i$
$s \leftarrow \varepsilon \cdot w_{\max}/n$
$\hat{w}_i = \lfloor w_i/s \rfloor$ `// auxiliary weights`

`// Dynamic program with auxiliary weights`
$\hat{W} = \sum_{i \in [n]} \hat{w}_i$
$G(0,0) \leftarrow 0$ and $G(0,x) \leftarrow \infty$ for $1 \leq x \leq \hat{W}$
for $i = 1, \ldots, n$ **do**
  for $W' = 1, \ldots, W$ **do**
    $G(i, W') \leftarrow \min(G(i-1, W'), g_i + G(i-1, W' - \hat{w}_i))$

`// Backwards run to determine items in packing`
$W^* \leftarrow \max\{W' \mid G(n, W') \leq G\}$
$S \leftarrow \emptyset$
for $i = n, \ldots, 1$ **do**
  if $G(i, W^*) = g_i + G(i-1, W^* - \hat{w}_i))$ **then**
    $S \leftarrow (S \cup i)$
    $W^* \leftarrow W^* - \hat{w}_i$
return S

- Subproblem: $G(i, W')$ is the smallest sum of sizes of any subset $S \subset \{1, \ldots, i\}$ that has total weight $\sum_{j \in S} w_j = W'$.
- We require a total weight W' and consider all subsets of the first i items that have exactly this weight. We want to know the smallest sum of sizes of any of those subsets.
- If there is no subset of $\{1, \ldots, i\}$ with total weight W', then set $G(i, W') = \infty$.
- We consider subproblems $G(i, W')$ for all $1 \leq i \leq n$ and $0 \leq W' \leq W$.

Base case and recursion:

- Base Cases: $G(0,0) = 0$ and $G(0,x) = \sum_{j=1}^{n} g_j + 1$ for all $1 \leq x \leq W, x \neq 0$.
- Recursion: For subproblem $G(i, W')$, is there a subset of smallest size that includes i?
- If yes, then $G(i, W^*) = g_i + G(i-1, W' - w_i)$, i.e., it combines task i with a smallest-weight subset of $\{1, \ldots, i-1\}$ that yields total weight exactly $W' - w_i$.
- If $W' - w_i < 0$, then this is impossible to achieve, and then $G(i-1, W' - w_i) = \infty$.
- If no subset includes i, then the best set is a subset of $\{1, \ldots, i-1\}$, i.e., $G(i, W') = G(i-1, W')$.
- Overall, the better option yields the correct result:

$$G(i, W') = \min\{G(i-1, W'), g_i + G(i-1, W' - w_i)\}$$

[Pic: Table representation of subproblems $G(i, W')$, Recursion example]

Note that the table can be filled starting with $i = 1$ and $W = 1$, and for each i going from $W' = 1$ to W. This takes time $O(nW)$.

Lemma 9. *There is a dynamic program to solve* Knapsack *in time* $O(nW)$.

This is *not polynomial time*, since the numbers in the input are represented in logarithmic encoding, i.e., the numbers w_i need only $O\left(\sum_{i=1}^{n} \log_2(w_i)\right)$ bits in the input – so W might be exponential in the input size. The running time of the algorithm is **pseudopolynomial**, it is polynomial if every number w_i gets represented in unary coding by w_i 1's.

Definition 14. *In the* unary input *of an instance, every numerical value is represented in unary encoding. The running time of an algorithm is called* pseudopolynomial *if it is bounded by a polynomial in the size of the unary input.*

To move from an exact solution in pseudopolynomial time to an approximate solution in polynomial time, we "make the weights small".

[Pic: Auxiliary weights]

Proof of Theorem 26. Given some $\varepsilon > 0$, we round each value w_i to the next multiple of a suitably chosen $s > 1$. This yields auxiliary weights $\hat{w}_i = \lfloor w_i/s \rfloor$. Using these weights, we apply the dynamic program.

With foresight, we choose $s = \varepsilon w_{\max}/n$. What is the impact on the running time?

- Sum of all auxiliary weights decreases to at most $\lfloor W/s \rfloor$.
- By Lemma 9, we solve the problem with auxiliary weights optimally in time $O(nW/s)$.
- Note that
$$\frac{n \cdot W}{s} = \frac{n^2 \cdot W}{\varepsilon w_{\max}} \leq \frac{n^2 \cdot n \cdot w_{\max}}{\varepsilon w_{\max}} = \frac{n^3}{\varepsilon}$$
which proves the running time of $O(n^3/\varepsilon)$.

What about the approximation guarantee? Suppose S is a packing computed by the algorithm and S^* an optimal solution. Then

$$\begin{aligned}
\sum_{i \in S^*} w_i \quad &= \sum_{i \in S^*} s \cdot \frac{w_i}{s} \leq \sum_{i \in S^*} s \cdot \left(\lfloor \frac{w_i}{s} \rfloor + 1 \right) \leq ns + s \cdot \sum_{i \in S^*} \hat{w}_i \\
&\leq ns + s \cdot \sum_{i \in S} \hat{w}_i && \text{(since } S \text{ optimal for } \hat{w}_i\text{)} \\
&\leq ns + s \cdot \sum_{i \in S} \frac{w_i}{s} = \quad \varepsilon w_{\max} + \sum_{i \in S} w_i && \text{(by definition of } s\text{)}
\end{aligned}$$

Using $w_{\max} \geq \sum_{i \in S^*} w_i$, we see

$$\sum_{i \in S} w_i \geq \sum_{i \in S^*} w_i - \varepsilon w_{\max} \geq (1 - \varepsilon) \sum_{i \in S^*} w_i$$

This implies that the approximation ratio is at most $1/(1 - \varepsilon) \leq 1 + 2\varepsilon$. □

3.3 Vertex Cover on Trees

In this section, we discuss a general approach for dynamic programming on tree graphs. The approach can be used to solve a large variety of optimization problems that are NP-hard on general graphs. We explain the idea in the context of the VERTEX COVER problem. In particular, we consider a more general version with vertex costs.

- $G = (V, E)$ is a tree, i.e., an undirected, connected graph without cycles
- Each vertex $v \in V$ has a weight/cost $w_v \geq 0$.
- Vertex cover $C \subseteq V$: For each edge $e \in E$ at least one incident vertex in C
- **Goal:** Find a cheapest vertex cover, i.e., one with smallest sum of costs

We construct a dynamic program.

- Root the tree in an arbitrary node r. Now each node (except r) has a parent node
- Nodes without children nodes are *leaves*
- We denote by T_v the subtree of v and all descendants of v
- A subproblem for every node/subtree. Construct solution "bottom-up":
- Start at the leaves. Given solutions for all children, compose solution for parent

At each node, we compute two values:

$$W^0(v) = \text{ cost of cheapest vertex cover of } T_v \text{ that does not include } v$$
$$W^1(v) = \text{ cost of cheapest vertex cover of } T_v \text{ that includes } v$$

The dynamic program proceeds as follows:

- If v is a leaf, then $W^0(v) = 0$ and $W^1(v) = w_v$
- Now suppose v has children $v_1, \ldots, v_k$.
- If v is not in the cheapest cover, all edges to the children must be covered by the children. Hence, they must all be in the cover:

$$W^0(v) = \sum_{i=1}^{k} W^1(v_i)$$

- If v is in the cheapest cover, the children do not have to be. Hence, for every child we pick whatever is cheaper for the subtree rooted at the child

$$W^1(v) = w_v + \sum_{i=1}^{k} \min(W^0(v_i), W^1(v_i))$$

[Pic: v out $\rightarrow$ every child must be in, otherwise may or may not (whatever better for cost)]

The cheapest cost of a vertex cover is given by $\min(W^0(r), W^1(r))$. To construct the cover, we proceed top-down.

- If the minimum is $W^0(r)$, then r stays out and all children of r must be in the cover.
- Hence, r forces all children into the cover.
- Generally, a node forced by the parent must join the cover.
- An unforced node is free to join or stay out, depending what yields the cheaper cost.

- If a node stays out, it forces all its children.
- This way we obtain an optimal vertex cover.

Theorem 27. *The Weighted* Vertex Cover *problem on trees can be solved optimally in polynomial time.*

3.4 Euclidean MinTSP

We discuss a PTAS for MinTSP in **Euclidean space**:

- n cities, every city i is a point $p_i \in \mathbb{R}^\ell$
- Distances are the Euclidean distances $d(p_i, p_j) = \sqrt{\sum_{k=1}^{\ell}(p_{ik} - p_{jk})^2}$
- For simplicity we restrict to the plane with dimension $\ell = 2$

Preparation. We "round" all city locations to obtain integer coordinates. Place a grid over the plane with some stepsize δ for horizontal and vertical steps.

- Integer coordinates represent multiples of δ
- Move every city to integer coordinates.
- δ too small $\rightarrow$ grid too fine, too many integers to consider, running time very high
- δ too large $\rightarrow$ grid too coarse, movement to integer coordinates heavily distorts the instance and deteriorates approximation quality

We achieve the right trade-off for the δ using moving, scaling, and rounding:

Moving: Place origin of the coordinate system s.t. smallest horizontal and vertical coordinate of any city is 0, resp.

Scaling: Scale δ s.t. largest horizontal *or* vertical coordinate of any city is n^2.

Rounding: Round down every coordinate of every city to the nearest integer:

$$(p'_{i1}, p'_{i2}) = (\lfloor p_{i1}/\delta \rfloor, \lfloor p_{i2}/\delta \rfloor) \qquad \text{for all citites } i \in [n]$$

Real coordinates and distances are all in terms of δ – but we will **ignore the common factor of** δ from now on, since it appears in all subsequent arguments and doesn't change things. After our preparation, all cities have integer coordinates in the square $[0, n^2] \times [0, n^2]$.

[Pic: Preparation of instance]

What is the impact of this adjustment on our PTAS?

- Running time: $O(n^2)$ coordinates in each dimension $\rightarrow$ at most $O(n^4)$ integer points in the grid (ok!)
- Approximation: For one dimension, there is one city at coordinate 0 and one at n^2 $\rightarrow$ Every tour for the (real) instance has distance at least $2n^2$
- Rounding changes every pairwise distance of two cities by at most $\pm 2\sqrt{2}$
- Consider any tour C with total length $d'(C)$ for the rounded and $d(C)$ for real locations.
- Then $|d(C) - d'(C)| \leq n \cdot 2\sqrt{2} = O(1/n) \cdot d(C^*)$
- PTAS will compute C with $d'(C) \leq (1 + \varepsilon) \cdot d'(C^*)$ in the rounded instance $\rightarrow$ For real distances, this tour satisfies $d(C) \leq (1 + \varepsilon + O(1/n)) \cdot d(C^*)$.

PTAS in the Rounded Instance (Idea) We outline a dynamic programming approach to compute a near-optimal tour for the rounded instance.

- Recursively partition the square $Q = [0, n^2] \times [0, n^2]$ into smaller squares
- Each recursion step partitions the current square in to four smaller squares
- A partition yields **four border lines** along all inner borders of the smaller squares
- Length of the border of a square on depth i is $n^2/2^i$.
- Stop recursion when square has 0 or 1 city inside. Happens at the latest when length of border is < 1.
- The recursion tree is a **quad-tree** – each square has either 0 or 4 children squares.
- How many recursive calls to go from border length n^2 to border length < 1?
  $\rightarrow$ Tree has depth at most $\lceil \log_2 n^2 \rceil < 1 + 2 \log_2 n$.

[Pic: Recursive partition of square, border lines of different depth]

We create a near-optimal tour by combining "partial tours" inside the squares.

- For a parent square, we combine partial tours computed for children squares.
- Towards this, we define "doors" or "portals" on the inner border lines of the square.
- A door allows the partial tour to enter/exit the square from/into neighboring squares.
- We require that partial tours leave a square *only* at a door. How many doors?
  Too many $\rightarrow$ too many potential meeting points, high running time
  Too little $\rightarrow$ restriction to doors creates distortion and bad approximation.
- For border line of a square place m equidistant doors.
- $m = \lceil (3 \log_2 n)/\gamma \rceil$ for **every border**, no matter how long the side length
- Distance between doors of a square of depth i is $n^2/(m \cdot 2^i)$

[Pic: Doors along the inner borders of squares]

Besides connecting cities inside the square, we must also allow partial tours to "cross through" the square when connecting cities of other squares far away, or to connect one or more cities to other cities far away. These options are captured by assembling partial tours at the doors.

A **visiting scheme** $\tilde{T}$ for a square is a set of pairs of the doors on the border of this square:

$$\tilde{T} = \{\{T_1, T_1'\}, \{T_2, T_2'\}, \ldots, \{T_r, T_r'\}\}$$

For the square, if a partial tour with the visiting scheme $\tilde{T}$ is used to build our final tour, then the tour enters the square via T_i and then leaves it via $T_{i'}$ (or vice versa), for each $i = 1, \ldots, r$.

Using this intuition, we can build a near-optimal tour bottom-up in the quad-tree. Our algorithm computes an optimal **legal tour**, i.e., a shortest tour that crosses borders of squares only at doors.

Base Cases. Consider a leaf square Q in the tree with at most 1 city. First, suppose Q has no city. We define $L_Q(\tilde{T})$ as the total length of segments resulting from all $\{T_i, T_i'\}$:

$$L_Q(\tilde{T}) = \sum_{i=1}^{r} d(T_i, T_i')$$

Second, if Q has one city c, then for every visiting scheme $\tilde{T}$, we include a "detour" to c on one connection $\{T_i, T_{i'}\}$ with the smallest overhead in length:

$$L_Q(\tilde{T}) = \min_{k=1,\dots,r} \left(d(T_k, c) + d(c, T'_k) + \sum_{i=1, j\neq k}^{r} d(T_i, T'_i) \right)$$

[Pic: Example schemes and tours for squares with 0 or 1 city]

Recursion. For parent square Q with children $Q_1, \dots, Q_4$ and visiting scheme $\tilde{T}$ of Q:

- Consider *all combinations* of visiting schemes for the children squares $(\tilde{T}_1, \tilde{T}_2, \tilde{T}_3, \tilde{T}_4)$
- We call a combination **compatible** with each other and with $\tilde{T}$ if all doors of these visiting schemes fit to each other, and a feasible partial tour for visiting scheme $\tilde{T}$ is created.
- In a compatible combination, there are no "loose ends", where partial connections break off, no "subtours" that create to a closed cycle (unless Q is the origin of the quad-tree), no "splits" where a tour splits into two branches, etc. pp.
- A compatible combination is evaluated by the total length of the segments
- For $\tilde{T}$ we only keep track of a *best* compatible combination:

$$L_Q(\tilde{T}) = \min_{(\tilde{T}_1, \tilde{T}_2, \tilde{T}_3, \tilde{T}_4) \text{ compatible}} L_{Q_1}(\tilde{T}_1) + L_{Q_2}(\tilde{T}_2) + L_{Q_3}(\tilde{T}_3) + L_{Q_4}(\tilde{T}_4)$$

Running Time. The main property to bound the running time is a bounded number of visiting schemes. For each given square, there are at most $2^{O(m)}$ many visiting schemes that must be considered (explanation below). This implies:

Lemma 10. *The recursion can be implemented in time polynomial in n.*

Proof. Suppose we only need to test $2^{O(m)}$ visiting schemes for each square.

- Running time for a given square Q: Enumerate and test all $(2^{O(m)})^5 = 2^{O(m)} = n^{O(1/\gamma)}$ possibilities of combinations of visiting schemes for Q and the four children. Checking compatibility is possible in polynomial time (negligible here).
- How many squares/nodes in the quad-tree? Tree has depth $O(\log n)$. Each city contained in a hierarchical sequence of squares that compose a path to the root. Thus, at most $O(n \log n)$ squares with cities. Each city can cause at most $O(\log n)$ squares without cities.

Total running time: $O(n \log n) \cdot 2^{O(m)} = n^{O(1/\gamma)}$. □

[Pic: $O(\log n)$ empty squares caused by two neighboring cities]

Now let us bound the number of visiting schemes for a square that must be considered to compute an optimal legal tour.

Proposition 1. *There is a shortest legal tour that*

1. *visits every door at most twice, and*
2. *never crosses (and only meets) in a door.*

Proof. If a tour comes to a door more than twice, it is easy to see that we shorten the tour. Hence, the shortest tour visits a door at most twice. Now suppose it crosses itself upon its visit at a door. Then by rerouting the parts, we can avoid a crossing (and merely cause a "meeting" in a point). □

[Pic: Multiple visits, crossing becomes meeting]

These observations allow to significantly reduce the number of visiting schemes that must be considered to compute an optimal legal tour.

Corollary 2. *We can restrict attention to visiting schemes $\tilde{T}$ such that*
- *the direct connections between two pairs $\{T_i, T_i'\}$ and $\{T_j, T_j'\}$ in $\tilde{T}$ do not cross, and*
- *every door appears at most in two pairs of $\tilde{T}$.*

To count the schemes $\tilde{T}$ with these properties, walk around the outer rim of the square starting in a specific door. In the beginning, all pairs of $\tilde{T}$ are closed. For each door T, write
- letter "(" for each closed pair from $\tilde{T}$ with T, mark that pair as open
- letter ")" for each open pair from $\tilde{T}$ with T, mark that pair as done.

This creates a string for each scheme $\tilde{T}$. This string has length at most $8m$ (at most $4m$ doors, each in at most 2 pairs), and it is a feasible expression of brackets (since connections of door pairs are non-crossing).

Theorem 28. *The number of feasible expressions of k brackets is the k-th Catalan number*

$$C_k = \frac{1}{k+1}\binom{2k}{k} = 2^{O(k)} \quad .$$

This provides an intuition why we need to consider at most $2^{O(m)}$ visiting schemes.

Approximation Ratio. We show: There is *some* legal tour of length at most $(1+\varepsilon)\cdot d(C^*)$. Since our algorithm computes the *optimal legal tour*, our tour is only shorter and also fulfills the bound.
- Take an optimal tour C^* in the instance. Prolong segments of C^* with detours: Move every crossing of a border and C^* to the closest door.
- The detour to the door for one crossing with border of depth i is at most $2n^2/(2^i m)$
- Along borders of smaller depth i, we have longer detours to the doors – but there are also less borders of this type.
- If the cities are placed badly, we cross the borders for small i very frequently.

[Pic: Example, many crossings of a border with long detours]

In a sense, the bad constellations mean we were "unlucky" with our grid – it was placed badly w.r.t. the optimal tour C^*. We **randomly perturb** the position of our initial grid. Then it is *unlikely* (in a probabilistic sense) that we experience large detours.
- A grid of length $2L$, where L is the smallest power of 2 larger than n^2
- Lower left corner (a, b) is put randomly at an integer grid point in $[-L, 0] \times [-L, 0]$

- Segment of length $d(x, y)$ from C^* can cross at most $d(x, y)$ horizontal and $d(x, y)$ vertical border lines
- Optimal tour has at most $2 \cdot d(C^*)$ crossings
- Due to random positioning, each line has probability $2^i/(2L) < 2^i/(2n^2)$ to become the (union of) borders of squares of depth i

[Pics: Random positioning of grid, crossings of a border, border lines of depth i]

For the optimal tour C^*, it is clear where the tour crosses the integer grid lines. Consider one fixed such crossing.

- Grid line is a border line of some depth. The depth level is random.
- Hence, the number of doors on the line and the detour to the closest door is random.
- For each depth level i: Line has level i with probability $2^i/(2L)$. In that case, the detour is at most $2n^2/(2^i m)$

Expected detour length for the given crosssing:

$$\begin{aligned}\mathbb{E}[\text{ detour of } C^* \text{ for single crossing }] &\leq \sum_{i=1}^{1+2\log_2 n} \frac{2^i}{2L} \cdot \frac{2n^2}{2^i m} = \sum_{i=1}^{1+2\log_2 n} \frac{2n^2}{2Lm} \\ &\leq (1 + 2\log_2 n) \cdot \frac{1}{m} \leq \frac{(1 + 2\log_2 n) \cdot \gamma}{3\log_2 n} \leq \gamma\end{aligned}$$

By linearity of expectation:

$$\mathbb{E}[\text{ detour of } C^* \text{ for all crossings }] \leq \gamma \cdot \mathbb{E}[\text{ number of crossings }] \leq \gamma \cdot 2d(C^*)$$

Lemma 11. *The expected detour of an optimal tour C^* when turning it into a legal tour is at most $2\gamma \cdot d(C^*)$.*

Corollary 3. *With probability at least $1/2$, the detour of an optimal tour C^* when turning it into a legal tour is at most $4\gamma \cdot d(C^*)$.*

The algorithm can be **derandomized**: Simply run the algorithm for every position (a, b) for the lower left position of grid in $[-L, 0] \times [-L, 0]$. This increases the running time by a factor of $O(n^4)$. Using $\varepsilon = 4\gamma$, we obtain the main result in this section.

Theorem 29. *For every $\varepsilon > 0$, with probability at least $1/2$ the derandomized PTAS obtains a tour that has length at most $(1 + \varepsilon + O(1/n)) \cdot d(C^*)$. The running time is in $n^{O(1/\varepsilon)}$.*

The running time is very large – impractical even for medium-sized instances. Nevertheless, the result shows that Euclidean MinTSP has substantially more structure, which can be exploited algorithmically and leads to better approximation in the worst case than for (general) MinTSP or Δ-MinTSP.

Chapter 4

Local Search

4.1 Clustering Problems

We introduce three variants of basic clustering problems. In these problems, we are given a set K of data points x in a metric space (e.g., points in the Euclidean plane). The goal is to group points into **clusters**, i.e., set of *similar* points. Tasks of this sort have many applications in data analysis, where we need to determine similarities among, say, voters, customers, products, etc. according to criteria such as content, behavior, characteristics, etc.

Consider the k-Center problem:

- A set K of n data points, along with a metric $d : K \times K \to \mathbb{R}_{\geq 0}$
- A number $k \leq n$ of clusters we should determine
- Given a subset $C \subseteq K$ of *centers*, we consider for each point $x \in K$ the distance to the closest center $\min_{c \in C} d(x, c)$
- **Goal:** Choose a subset C of $|C| = k$ centers to minimize the max-cost $\max_{x \in K} \min_{c \in C} d(x, c)$, i.e., minimize the maximum distance of any point to its closest center.

We study a natural greedy algorithm (Algorithm 18) for this problem.

[Pic: k-Center in Euclidean plane, largest radius of any cluster, example run of algorithm]

Theorem 30. *Algorithm 18 has an approximation ratio of at most 2 for* k-Center.

Proof. Suppose C^* is an optimal set of k cluster centers and C_A the set computed by the algorithm. Moreover, let

$$r^* = \max_{x \in K} \min_{c \in C^*} d(x, c)$$
$$r_A = \max_{x \in K} \min_{c \in C_A} d(x, c)$$

be the maximum distance to the closest center in C^* and C_A, resp.

- Assign each data point $x \in K$ to closest center $c \in C^*$
- This gives distance $d(x, c) \leq r^*$. Hence, by triangle inequality, any two points $x, x' \in K$ that both get assigned to c have distance $d(x, x') \leq 2r^*$.
- Now consider the centers from C_A.

Algorithm 18: Greedy for k-Center

```
Input: Set K of n data points, distance function d, integer 0 < k ≤ n
c_1 ← arbitrary point from K
C ← {c_1}
for i = 2, ..., k do
    c_i ← point with maximum distance to closest center from C
    C ← C ∪ {c_i}
return C
```

- If all the $c_a \in C_A$ are assigned to different centers from C^*, then $d(x, c_a) \leq 2r^*$ for every x assigned to c in C^*. Hence, $r_A \leq 2r^*$.
- Otherwise, there is a $c \in C^*$ that is closest center from C^* for two $c_A^1, c_A^2 \in C_A$.
- Suppose c_A^1 was chosen first by the algorithm. Later, when c_A^2 was chosen, it was a point with largest distance to the closest center from C_A up to this point. The distance was at most $d(c_A^1, c_A^2) \leq 2r^*$.
- Hence, finally every point has distance $r_A \leq d(c_A^1, c_A^2) \leq 2r^*$ to its closest cluster center.

[Pic: Clusters, optimal centers, algorithm centers, distance bounds] □

Theorem 31. *For every $\varepsilon > 0$, there is no efficient algorithm with approximation ratio $(2 - \varepsilon)$ for* k-Center *(unless* P = NP*).*

Proof. We present a reduction from the NP-complete decision problem Dominating Set:

- We are given an undirected graph $G = (V, E)$ and an integer number k.
- $D \subseteq V$ is *dominating* if, for every $v \in V$, D contains v or at least one neighbor of v
- **Goal:** Decide if there is a dominating set of cardinality at most k.

Note the difference to between dominating set and vertex cover: Intuitively, a dominating set is a variant of vertex cover, where we want to cover *every vertex* by (itself or) a neighbor, and not *every edge* by an incident vertex.

[Pic: Dominating Set, example]

We show that any $(2 - \varepsilon)$-approximation algorithm for k-Center can be used to correctly decide Dominating Set by proving a polynomial-time reduction. Given any instance of Dominating Set, we construct an instance of k-Center:

- We set $K = V$.
- The integer k remains the same.
- The distance is $d(u, v) = 1$ if $\{u, v\} \in E$ and $d(u, v) = 2$ otherwise.
- Hence, every solution for the k-Center instance yields max-cost 0, 1, or 2.

Clearly, the instance can be constructed in polynomial time. Now suppose, we can solve it with our algorithm and obtain a $(2 - \varepsilon)$-approximation.

- Dominating Set instance is yes-instance
  $\iff$ Set $D \subseteq V$ with $|D| = k$ contains either v or neighbor of v, for every $v \in V$
  $\iff$ Set $D \subseteq V$ with $|D| = k$ has a center of distance at most 1, for every $v \in V$
  $\iff$ k-Center instance has solution of max-cost 1 or 0.

Algorithm 19: Local Search for k-Median

Input: Set K of n data points, distance function d, integer $0 < k \leq n$

$t \leftarrow 0$
$M^0 \leftarrow$ arbitrary subset of k points from K
repeat
  Consider all sets of k points that are *neighbors* of M^t, i.e., that differ from M^t by exactly one center
  Let M be a neighbor set that yields strictly smaller sum-cost than M^t
  Set $M \leftarrow \perp$ if no strictly better neighbor set exists.
  if $M \neq \perp$ **then** $M^{t+1} \leftarrow M$ and $t \leftarrow t + 1$
until $M = \perp$
return M^t

- Hence, for any k-Center instance stemming from a yes-instance for Dominating Set, the algorithm must return a solution with max-cost at most $1 \cdot (2 - \varepsilon) < 2$, i.e., a solution with max-cost 0 or 1 (i.e., a feasible dominating set).
- For any no-instance, there are only solutions of max-cost 2 available.
- The algorithm determines a set of centers that represents a dominating set of size k whenever it exists.

[Pic: Approximation gap] □

A different variant is the k-Median problem:

- A set K of n data points, along with a metric $d : K \times K \to \mathbb{R}_{\geq 0}$
- A number $k \leq n$ of clusters we should determine
- Given a subset $M \subseteq K$ of *centers*, we consider for each point $x \in K$ the distance to the closest center $\min_{m \in M} d(x, m)$
- **Goal:** Choose a subset M of $|M| = k$ centers to minimize the sum-cost $\sum_{x \in K} \min_{m \in M} d(x, m)$, i.e., minimize the **sum over all points**, the distance to the closest center for each point.

We consider a *local search* algorithm (Algorithm 19) for this problem and state the main result without proof.

Theorem 32. *Algorithm 19 has approximation ratio at most 5 for* k-Median.

[Pic: Local search step]

4.2 Local Search

Algorithm 19 is an example of a general design template for so-called *strict local search* algorithms. Local search is a general heuristic to approach combinatorial optimization problems. The idea is an **iterative greedy algorithm** based on a notion of **neighborhood**.

Algorithm 20: Strict Local Search

```
1 Input: Instance I, objective function f, Neighborhood relation N
2 t ← 0
3 x_0 ← ComputeStartSolution(I)
4 repeat
5 |   y → FindBetterNeighbor(x_t)        // y = ⊥ if no better neighbor exists
6 |   if y ≠ ⊥ then x_{t+1} ← y and t ← t + 1
7 until y = ⊥
8 return x_t
```

- Consider an instance I of a minimization problem (similar for maximization)
- Let x be any feasible solution.
- For each feasible x, we choose a set $\mathcal{N}(x)$ of *neighboring feasible solutions*

The **strict local search** algorithm generalizes Algorithm 19 and is presented in generic form in Algorithm 20. The algorithm uses two subroutines:

- *ComputeStartSolution* computes some feasible starting solution.
- *FindBetterNeighbor* searches through the neighborhood $\mathcal{N}(x_t)$ of solution x_t to find any **strictly better** solution $y \in \mathcal{N}(x_t)$ with $f(y) < f(x_t)$.

As long as a better neighboring solution y exists, the algorithm repeats the search in the neighborhood of y, otherwise the algorithm terminates.

Some issues when implementing strict local search:

1. How to choose the neighborhoods $\mathcal{N}(x)$?
   Often the most important part. Neighborhoods should be large to allow for good solutions. However, the larger the neighborhood, the more time we need to search it. In many problems, each solution x can be represented by a bit vector $b^x = (b_1^x, b_2^x, \ldots)$. Then a popular choice is the **k-flip neighborhood**

   $$\mathcal{N}_k(x) = \{y \mid \text{ vector } b^y \text{ differs from } b^x \text{ in at most } k \text{ positions } \}.$$

2. How to choose the starting solution?
   Intrinsically depends on the neighborhoods. We should avoid being in or close to a local optimum, otherwise the search is pre-determined and does not lead to meaningful improvements.

3. Which improving neighbor y is chosen?
   A popular choice for small neighborhoods is a best neighbor, i.e., one that decreases f by the largest amount. For larger neighborhoods, one uses additional heuristics, e.g., draws a random neighbor until a better one is found (or a pre-defined limit on the number of draws is reached)

Example: Gradient Descent

- Suppose you want to minimize a function $f : \mathbb{R} \to \mathbb{R}$
- Let f be continuously differentiable.

- Neighborhood of a point x: $\mathcal{N}(x) = \{y \mid |x - y| = \delta\}$ for some small $\delta > 0$
- Start at some initial point x_0
- Move left/right depending on the direction of steepest descent.

[Pic: Gradient Descent]

More formally, we base our decision on the gradient. The next lemma proves that if x is not a local optimum (i.e., $\nabla f(x) \neq 0$), then within a small neighborhood of x, we have a point y with $f(y) < f(x)$. This point can be found by moving in the direction of the negative gradient. This also works for multi-dimensional functions.

Lemma 12. *Consider any continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ and a point x with $\nabla f(x) \neq 0$. Then there is some $\eta > 0$ with*

$$f(x - \eta \cdot \nabla f(x)) < f(x)$$

where $\nabla f(x)$ is the gradient of f at point $x \in \mathbb{R}^n$.

Proof. We approximate f using linear Taylor polynomials and obtain, for small enough z,

$$f(x + z) = f(x) + \nabla f(x)^T \cdot z + O(z^T \cdot z).$$

Using $z = -\eta \cdot \nabla f(x)$ for small $\eta > 0$, we see that

$$f(x - \eta \cdot \nabla f(x)) = f(x) - \eta \|\nabla f(x)\|^2 + \eta^2 O(\|\nabla f(x)\|^2).$$

For sufficiently small η, this implies that $f(x - \eta \cdot \nabla f(x))$ is smaller than $f(x)$. □

Example: Vertex Cover (mostly for didactic reasons)

- Associate with each subset $C \subseteq V$ of vertices a characteristic vector $b^C = (b^C_v)_{v \in V}$, where $b^C_v = 1$ if $v \in C$ and $b^C_v = 0$ otherwise.
- We use the 1-flip neighborhood: Neighbors of set C are all subsets C' whose characteristic vector differs by at most one entry:

$$\mathcal{N}(C) = \{C' \subseteq V \mid \text{ there is at most one } v \in V \text{ with } b^C_v \neq b^{C'}_v\}$$

- Note that $C \in \mathcal{N}(C)$. For every other $C' \in \mathcal{N}(C)$ with $C' \neq C$ we have either one more vertex or one less vertex than in C.
- Say we use as feasible starting solution $C_0 = V$. Then in each round, strict local search will remove one vertex until every further removal leaves some edge uncovered.
- Depending on which vertices we remove first, this can lead to very good or very bad solutions.

[Pic: Local search for vertex cover on star or path graphs]

4.3 Complexity of Local Search

We saw in Theorem 32 that the local search in Algorithm 19 leads to a 5-approximation for the k-Median problem. However, is **local search a polynomial-time algorithm**? Clearly, each iteration of Algorithm 19 needs polynomial time. But how many steps do we need until we reach a local optimum?

The number of steps is polynomial for the local search of Vertex Cover – starting from $V = C$, the set of vertices only shrinks. Hence, the number of steps is at most n. The main problem is *which neighboring solution* we pick to reach a good solution in the end. However, for k-Median, the number of possible subsets is $\binom{n}{k}$, which for some values of k could be in the order of $\Theta(n^k)$. As such, it might even take long to reach *any arbitrary* local optimum.

We study the complexity of local search using a general class of polynomial-time local search problems called PLS.

Definition 15. *A **polynomial search problem** in the class* PLS *is given by a problem from* NPO *and a neighborhood relation $\mathcal{N}$ over feasible solutions, along with the following properties:*

- *There is a polynomial-time algorithm A that computes for each instance I a feasible starting solution x.*
- *Given any instance I and a feasible solution x, there is a polynomial-time algorithm B to decide if x is a local optimum w.r.t. $\mathcal{N}(x)$. If not, B returns a feasible neighboring solution $y \in \mathcal{N}(x)$ that is strictly better – formally, if f is the function to be optimized, then $f(y) < f(x)$ for a minimization problem and $f(y) > f(x)$ for a maximization problem.*

The definition describes all problems for which the components of the generic template in Algorithm 20 can be implemented in polynomial time, i.e., A implements ComputeStartSolution and B implements FindBetterNeighbor. More concisely, for every problem in PLS, we can use the standard algorithm to compute a local optimum:

1. Let I be the instance
2. Starting solution: $x = A(I)$
3. Loop: **while** $B(I, x) \neq \perp$ **do** $x \leftarrow B(I, x)$

In a local search problem from PLS, we are striving to compute a local optimum. Clearly, local optima can be found by running the standard algorithm. As long as the number of solutions is finite, the algorithm is guaranteed to terminate, so a local optimum exists. However, there are many problems, for which there exist classes of instances and initial solutions, such that the standard algorithm requires an exponential number of iterations.

In general, we are neither making a yes/no decision, nor are we optimizing and looking for a good solution in terms of the objective function. The problems here are *search problems*, not decision or optimization problems. How can be characterize the complexity of local search problems? We use a similar approach as for NP: We use polynomial-time reductions and identify PLS-complete problems, the hardest problems in PLS.

Definition 16. *Suppose P_1 and P_2 are two polynomial search problems. There is a* PLS-***reduction*** *from P_1 to P_2 (denoted $P_1 \leq_{\mathsf{PLS}} P_2$) if there are polynomial transformations Φ and Ψ such that*

- *If I is an instance of P_1, then $\Phi(I)$ is an instance of P_2.*
- *If x is a local optimum for instance $\Phi(I)$ of P_2, then $\Psi(I, x)$ is a local optimum for instance I of P_1.*

A problem $P \in$ PLS is PLS-***complete*** *if for every problem $P' \in$ PLS we have $P' \leq_{\mathsf{PLS}} P$.*

[Pic: Schema reduction, completeness]

PLS-reductions and PLS-completeness:

- We efficiently transform an instance of problem P_1 into an instance of problem P_2.
- Suppose we "solve" P_2 efficiently – i.e., find a local optimum in polynomial time.
- Then we transform the local optimum for P_2 into a local optimum for P_1.
- Hence, if we find local optima for P_2 in polynomial time, then we can do so also for P_1.
- In this sense, P_1 is "easier" than P_2, and P_2 "harder" than P_1.
- The PLS-complete problems are the "hardest" problems in PLS.

Example: Minimum Balanced Cut

- $G = (V, E)$ undirected graph, even number of nodes n
- Each edge $e \in E$ has a non-negative weight $w(e) > 0$.
- Partition vertex set into W and $U = V \setminus W$ with $|W| = n/2$
- Minimize sum of edge weights in the cut, i.e.,

$$f(W) = \sum_{e \in cut(W)} w(e) \qquad \text{where } cut(W) = \{\, \{u, v\} \in E \mid u \in W, v \notin W \}$$

- $2k$-flip-neighborhood: W and W' are neighbors if $|W \setminus W'| = |W' \setminus W| \leq k$.
- Each set W has at most $\binom{n/2}{k}^2$ neighboring sets.

[Pic: Example MBC, neighboring set]

Theorem 33. *It is* PLS*-complete to find a local optimum for Minimum Balanced Cut with the 2-flip neighborhood.*

Example: MinTSP

- Reconsider the MinTSP problem with n cities discussed above.
- Given tour C, remove k edges, and reconnect the remaining paths into new tour
- $2k$-flip neighborhood: Every tour resulting from C by removal of any subset of k edges and any subsequent reconnection
- There are at most $\binom{n}{k}$ choices for k edges. After removing k edges, we get k paths, and there are at most $k! \cdot 2^k$ tours that can evolve by reconnecting these paths.
- Each tour C has at most $\binom{n}{k} \cdot k! \cdot 2^k$ neighboring tours.
- Local search with $2k$-flip neighborhood for MinTSP is usually called k-**Opt**.

[Pic: Example TSP, 2-Opt, 8-Opt]

Theorem 34. *There is a constant k, such that it is* PLS*-complete to find a local optimum for* MinTSP *with the $2k$-flip neighborhood.*

Algorithm 21: Local Search for Facility Location

Input: Set C of clients, set F of locations, distance function d, opening costs f

$t \leftarrow 0$
$X^0 \leftarrow$ arbitrary subset of locations from F
repeat
  Consider all neighbor sets of opened facilities that differ from X^t by adding, removing, or swapping a single facility
  Let X be a neighbor set such that $cost(X) < cost(X^t)$
  $X \leftarrow \perp$ if there no strictly better neighbor set exists.
  if $X \neq \perp$ **then** $X^{t+1} \leftarrow X$ and $t \leftarrow t+1$
until $X = \perp$
return X^t

4.4 Facility Location

We consider local search for the Facility Location problem.

- C is a set of clients, F a set of possible locations for service facilities
- Distances $d(x, y)$ between all points in $C \cup F$ form a metric
- At each location $j \in F$ we can decide to open a facility or not
- Opening cost: $f_j \geq 0$ for every $j \in F$
- Connection cost of client i to location j: $d(i, j)$
- For a subset $X \subseteq F$ of opened facilities, each client connects to closest one, i.e., $\mathsf{dist}(i, X) = \min_{j \in X} d(i, j)$
- We define $next(i, X) = j$ if $d(i, j) = \mathsf{dist}(i, X)$, i.e., an opened facility closest to i

Goal: Open facilities to minimize sum of opening and connection costs

$$cost(X) = \sum_{j \in X} f_j + \sum_{i \in C} \mathsf{dist}(i, X) \ .$$

[Pic: Schema Facility Location]

Consider Algorithm 21, a local search algorithm for Facility Location similar to Algorithm 19 for k-Median. To define the neighborhood, we allow to add, remove or swap a single opened facility.

Let X^* be a global optimum and X be a local optimum in an instance of Facility Location. The main result is that $cost(X) \leq 3 \cdot cost(X^*)$. Put differently,...

Theorem 35. *Algorithm 21 has an approximation ratio of 3.*

We first show that a ratio of $3 - \varepsilon$ for constant $\varepsilon > 0$ impossible. Towards this end, we construct an instance with a costly local optimum.

- n clients, $n + 1$ service locations
- Distance in a tree: root facility, children clients, each client has a child facility

- $d(i,j)$ = shortest path length between i and j in the tree (this is a metric!)
- Opening cost of root $f_0 = 2n-2$, all other opening costs are $f_i = 0$
- $X^* = \{1,\dots,n\}$, all leaf facilities in the tree, $cost(X^*) = n$
- $X = \{0\}$, only root is open, $cost(X) = n + 2n - 2 = 3n - 2$
- Ratio is $3 - 2/n \to 3$ as $n \to \infty$
- Observe that X is a local minimum:
  Open a leaf: Same opening cost, same connection cost
  Close root: Infinite connection cost
  Swap root with a leaf: New cost $0 + 1 + (n-1)\cdot 3 = 3n-2$, same cost

[Pic: Tree, facilities, clients, distances, opening costs]

Concerning the upper bound, we here only show a ratio of 5, i.e., $cost(X) \le 5 \cdot cost(X^*)$. The bound of 3 in Theorem 35 can be shown by a more advanced analysis.

To show the bound of 5, we spilt $cost(X)$ into connection and opening costs and bound each part separately.

Lemma 13. *Let X be a local minimum and X^* be a global minimum. Then*

$$\sum_{i \in C} \mathsf{dist}(i,X) \le cost(X^*).$$

Proof. Consider $j \in X^* \setminus X$.

- Suppose we also open j in addition to X.
- Then $cost(X \cup \{j\}) \ge cost(X)$, since X is local minimum.
- Suppose for $X \cup \{j\}$ we connect to j exactly the clients i connected to j in X^*, i.e., where $next(i,X^*) = j$. We denote the resulting cost by $cost'(X \cup \{j\})$.
- Now $cost'(X \cup \{j\}) \ge cost(X \cup \{j\}) \ge cost(X)$. But $cost'(X \cup \{j\})$ and $cost(X)$ differ only by f_j and connection costs for clients i connected to j, so

$$\sum_{i\,:\,next(i,X^*)=j} \mathsf{dist}(i,X^*) + f_j \ge \sum_{i\,:\,next(i,X^*)=j} \mathsf{dist}(i,X)$$

  [Pic: Stars of clients connected to facilities, change when adding j to X]

- Summing this inequality for every $j \in X^* \setminus X$ and adding connection costs for clients connected to facilities in $X \cap X^*$, we get

$$cost(X^*) \ge \sum_{i \in C} \mathsf{dist}(i,X^*) + \sum_{j \in X^* \setminus X} f_j \ge \sum_{i \in C} \mathsf{dist}(i,X)$$

□

This bounds the connection costs in the local optimum. Let us now turn to the opening costs.

Lemma 14. *Let X be a local minimum and X^* be a global minimum. Then*

$$\sum_{j \in X} f_j \leq 2 \cdot \left(cost(X^*) + \sum_{i \in C} \mathsf{dist}(i, X) \right) \leq 4 \cdot cost(X^*).$$

Proof. The second inequality is a direct consequence of the previous lemma. We proceed to prove the first inequality. We now try to exchange a facility $j \in X$ by a closest facility $j^* \in X^*$.

- X is a local optimum, so $cost(X) \leq cost((X \cup \{j^*\}) \setminus \{j\})$
- The cost change from X to $(X \cup \{j^*\}) \setminus \{j\}$ can be bounded by assuming that only clients connected to j switch to j^*. This is not an improvement (otherwise the optimal cost for $(X \cup \{j^*\}) \setminus \{j\}$ would be one for sure), so

$$f_j + \sum_{i \,:\, next(i,X)=j} d(i,j) \;\leq\; f_{j^*} + \sum_{i \,:\, next(i,X)=j} d(i,j^*)$$

  or, equivalently,

$$f_j \leq f_{j^*} + \sum_{i \,:\, next(i,X)=j} (d(i,j^*) - d(i,j)) \tag{4.1}$$

How big is the difference between $d(i,j)$ and $d(i,j^*)$?

- Triangle inequality: $d(i,j*) \leq d(i,j) + d(j,j^*)$, so $d(i,j^*) - d(i,j) \leq d(j,j^*)$
- j^* is closest to j, so $d(j,j^*) \leq d(j,j')$ for all $j' \in X^*$, especially for $j' = next(i,X^*)$
- This implies

$$\begin{aligned} d(i,j^*) - d(i,j) \;&\leq\; d(j,j^*) \;\leq\; d(j, next(i,X^*)) \;\leq\; d(j,i) + d(i, next(i,X^*)) \\ &=\; \mathsf{dist}(i,X) + \mathsf{dist}(i,X^*) \end{aligned}$$

- Hence, using (4.1) above:

$$f_j \leq f_{j^*} + \sum_{i \,:\, next(i,X)=j} (\mathsf{dist}(i,X) + \mathsf{dist}(i,X^*)) \tag{4.2}$$

[Pic: Closest facility from X^*, distance bound]

Suppose every facility j^* is chosen as a closest one by exactly one facility in j. Then, by summing (4.2) over all $j \in X$, we sum over all f_{j^*} exactly once and obtain

$$\begin{aligned} \sum_{j \in X} f_j &\leq \sum_{i \in C} (\mathsf{dist}(i,X) + \mathsf{dist}(i,X^*)) + \sum_{j^* \in X^*} f_{j^*} \\ &= \sum_{i \in C} \mathsf{dist}(i,X) + cost(X^*) \\ &\leq 2 \cdot cost(X^*) \end{aligned}$$

where the last inequality uses the previous lemma. This proves a stronger bound than the one claimed in the present lemma, which would imply the ratio of 3 claimed in Theorem 35.

In general, we might count a facility $j^* \in X^*$ more than once as a closest facility. Towards this end, we consider a **primary facility** from X:

- Suppose $j^* \in X$ is the closest facility for facility $j \in X$.
- For j^*, consider $X(j^*) = \{j \in X \mid j^* \text{ closest facility from } X^* \text{ for } j\}$.
- j is a *primary* facility for j^* if it the facility from $X(j^*)$ closest to j^*
- Otherwise, j is a *secondary* facility.

We provide an upper bound on $cost(X)$ by removing any secondary facility $j \in X$. Consider its' closest facility j^* from X^*. Let $j' \in X$ be the primary facility of j^*. Suppose we remove j and all clients connected to j get connected to j'.

- Clearly, the opening cost decreases by f_j.
- What about the connection cost of a client i connected to j (with $next(i, X) = j$)?

$$\begin{aligned} d(i,j') - d(i,j) &\leq d(j,j^*) + d(j^*,j') \\ &\leq 2 \cdot d(j,j^*) && (j' \text{ primary, so closer to } j^* \text{ than } j) \\ &\leq 2 \cdot d(j, next(i,X^*)) && (j^* \text{ the one from } X^* \text{ closest to } j) \\ &\leq 2 \cdot (d(j,i) + d(i, next(i,X^*))) \\ &= 2 \cdot (\mathsf{dist}(i,X) + \mathsf{dist}(i,X^*)) \end{aligned}$$

- This is an upper bound on the increase in connnection cost for client i when we remove j, since i might only get smaller connection cost when being connected to its closest facility from $X \setminus \{j\}$.
- Since X is a local minumum we see that

$$f_j \leq \sum_{i \,:\, next(i,X)=j} 2 \cdot (\mathsf{dist}(i,X) + \mathsf{dist}(i,X^*)) \tag{4.3}$$

- Now we combine (4.2) and (4.3) to obtain

$$\begin{aligned} \sum_{j \in X} f_j &\leq \sum_{j^* \in X^*} f_{j^*} + 2 \cdot \sum_{i \in C} (\mathsf{dist}(i,X) + \mathsf{dist}(i,X^*)) + \\ &\leq 2 \cdot \left(cost(X^*) + \sum_{i \in C} \mathsf{dist}(i,X) \right) \\ &\leq 4 \cdot cost(X^*) \end{aligned}$$

□

While Algorithm 21 comes with a good approximation guarantee, the running time is not necessarily polynomial. We **change Algorithm 21** to accept a neighbor set X only if represents a **significant improvement step**. For a fixed value $\delta \in (0,1)$, we apply the following choice for neighbor set X:

$$\begin{gathered} X \text{ is a neighbor set such that } \varepsilon) \cdot cost(X) < (1-\delta) \cdot cost(X^t), \\ \text{or } X \leftarrow \perp \text{ if no such neighbor set exists.} \end{gathered}$$

Clearly, the adjusted algorithm converges only to an **approximate local optimum**, i.e., the final solution X^t satisfies that $(1-\delta) \cdot cost(X^t) \leq cost(X)$ for every neighbor $X \in \mathcal{N}(X^t)$.

Lemma 15. *Let X^0 be the initial solution and X^* an optimal solution. Algorithm 21 with significant improvement steps converges to an approximate local optimum in a number of steps at most*

$$\frac{1}{\delta} \cdot \log_2 \left(\frac{cost(X^0)}{cost(X^*)} \right).$$

Proof. How many significant improvement steps are possible?

- Starting with a solution of $cost(X^0)$, we decrease the cost to at most a factor of $(1-\delta)$ in every step. After t steps, the solution has value at most $cost(X^0) \cdot (1-\delta)^t$
- The cost cannot decrease below $cost(X^*)$. Hence, if

  $$cost(X^*) \geq cost(X^0) \cdot (1-\delta)^t$$

  the algorithm must have terminated. Solving for t, we see that if

  $$t \geq \log_{1/(1-\delta)} \left(\frac{cost(X^0)}{cost(X^*)} \right)$$

  the algorithm must be done. Applying bounds on the logarithm

  $$\log_{1/(1-\delta)} x = \frac{1}{\log_2 \frac{1}{1-\delta}} \cdot \log_2 x = \frac{1}{\log_2 \left(1 + \frac{\delta}{1-\delta}\right)} \cdot \log_2 x \leq \frac{1}{\frac{\delta}{1-\delta}} \cdot \log_2 x \leq \frac{1}{\delta} \cdot \log_2 x$$

  shows that if

  $$t \geq \frac{1}{\delta} \cdot \log_2 \left(\frac{cost(X^0)}{cost(X^*)} \right)$$

  the algorithm must have terminated.

□

For a constant $\varepsilon > 0$, we choose $\delta = \varepsilon/|F|$. With this choice, even the approximation ratio is almost preserved – the central bounds in (4.2) and (4.3) continue to hold if we increase the right-hand sides by $\delta \cdot cost(X)$. As we sum over (at most) $|F|$ facilities, we obtain a cost increase of $\varepsilon \cdot cost(X)$. We omit the details and state the main result.

Theorem 36. *Let X^0 be the initial solution and X^* an optimal solution. Algorithm 21 with significant improvement steps converges to a $(3+\varepsilon)$-approximate solution in at most*

$$\frac{|F|}{\varepsilon} \cdot \log_2 \left(\frac{cost(X^0)}{cost(X^*)} \right)$$

steps, which is a number polynomial in the input size.

Chapter 5

Linear Programming

5.1 Canonical Form and Examples

Linear programming is a very important area with a large number of practical applications. The goal is to optimize a linear objective function in variables $x_1, \ldots, x_n$, subject to a set of linear constraints that restrict the solution space. Here is a simple example with $n = 4$ variables and $m = 5$ constraints:

$$\begin{array}{rrcl} \text{Minimize} & 2x_1 - x_2 + 4x_4 & & \\ \text{subject to} & x_1 - 2x_2 + x_3 & \geq & 5 \\ & 2x_1 + 3x_2 - 4x_4 & \geq & -10 \\ & 3x_2 - 2x_3 & \geq & 2 \\ & x_2 & \geq & 0 \\ & x_4 & \geq & 0 \end{array} \tag{5.1}$$

For brevity, we will usually write linear programs using matrix-vector notation. With

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}, \quad \mathbf{c} = \begin{pmatrix} 2 \\ -1 \\ 0 \\ 4 \end{pmatrix} \quad \mathbf{A} = \begin{pmatrix} 1 & -2 & 1 & 0 \\ 2 & 3 & 0 & -4 \\ 0 & 3 & -2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 5 \\ -10 \\ 2 \\ 0 \\ 0 \end{pmatrix},$$

we can express the optimization problem in a more compact form

$$\begin{array}{rl} \text{Minimize} & \displaystyle\sum_{i=1}^{4} c_j x_j \\ \text{subject to} & \displaystyle\sum_{i=1}^{4} a_{ij} x_j \quad \geq \quad b_i \quad \text{for each } i = 1, \ldots, 5 \end{array} \qquad \text{or, simply} \qquad \begin{array}{rl} \text{Minimize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b}. \end{array}$$

We will work with these formulations throughout. For simplicity, we assume all parameters in $\mathbf{c}$, $\mathbf{A}$ and $\mathbf{b}$ are rational numbers.

Definition 17. *A linear program (LP) in* ***canonical form*** *with n variables and m constraints consists of a vector $\mathbf{x}^T = (x_1, \ldots, x_n)$ of variables, a vector $\mathbf{c} \in \mathbb{Q}^n$ of cost coefficients, a matrix $\mathbf{A} \in \mathbb{Q}^{m \times n}$ and a vector $\mathbf{b} \in \mathbb{Q}^m$. The goal is to*

$$\begin{aligned} &\textit{Minimize} && \mathbf{c}^T\mathbf{x} \\ &\textit{subject to} && \mathbf{A}\mathbf{x} \geq \mathbf{b}. \end{aligned}$$

Every linear optimization problem can be expressed using an LP in canonical form:

- If we want to **maximize $\mathbf{c}^T\mathbf{x}$**, we can **minimize $-\mathbf{c}^T\mathbf{x}$**.
- A **$\leq$-constraint** is equivalent to a **$\geq$-constraint**:

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \qquad \Longleftrightarrow \qquad \sum_{j=1}^{n} -a_{ij}x_j \geq -b_i.$$

Example: Production planning

- n products, m resources
- Product j can be sold at a price of $p_j \geq 0$ per unit.
- To produce a unit of product j we need a_{ij} units of resource i
- A supply of b_i units of resource i is available.
- **Goal:** Determine the units x_j of each product j to maximize the total income.

Linear program (in canonical form):

- Total income is $\sum_{j=1}^{n} p_j x_j$.
- Each resource i must not be overused, so $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$
- We cannot produce negative units of product j, so $x_j \geq 0$.

$$\begin{aligned} \text{Minimize} \quad & \sum_{j=1}^{n} -p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} -a_{ij}x_j \geq -b_i \quad \text{for each } i = 1, \ldots, m \\ & x_j \geq 0 \quad \text{for each } j = 1, \ldots, n \end{aligned}$$

Example: Weighted Vertex Cover

- Requires a *binary* decision for each node (makes the problem NP-hard)
- For simplicity, let $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$
- $x_v \in \{0, 1\}$ indicates if node v is included in vertex cover ($x_v = 1$) or not ($x_v = 0$).
- Node v has cost $w_v \geq 0$ when being in the cover.
- Each edge should be "covered". How to express this as linear constraint?
- $x_u + x_v \geq 1$ for each $e = \{u, v\} \in E$.

Integer linear program (ILP) for Weighted Vertex Cover:

$$\begin{aligned} \text{Minimize} \quad & \sum_{v \in V} w_v x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \quad \text{for each } \{u, v\} \in E \\ & x_v \in \{0, 1\} \quad \text{for each } v \in V \end{aligned} \tag{5.2}$$

Suppose we replace $x_v \in \{0,1\}$ by $0 \leq x_v \leq 1$.

- We obtain a linear program (not an integer linear one). It yields a **fractional** VERTEX COVER problem, where we can break each vertex into arbitrary pieces and include only a fraction of it in a cover.
- The linear program is a **linear relaxation** of (5.2) (more solutions are feasible). The optimal fractional solution can obtain a smaller objective value.

Since the 1980s there exist **polynomial-time algorithms to optimally solve linear programs**, i.e., algorithms that return an optimal vector $\mathbf{x}^*$ in time polynomial in n, m and the length of the (binary) encoding[1] of all parameters a_{ij}, c_i and b_j. We will give a high-level overview of some of these techniques, as well as structural properties that are useful for the application of LP-ideas in **approximation algorithms for NP-hard problems**.

5.2 Polytopes, Corners, and a Local Search Algorithm

Consider a linear program in canonical form. Some definitions/observations:

- An LP is **solvable** if there is at least one vector $\mathbf{x}$ with $\mathbf{Ax} \geq \mathbf{b}$.
- Such a vector is called a **solution**.
- Solution space $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Ax} \geq \mathbf{b}\}$ of the LP does *not depend on* $\mathbf{c}$.
- Consider a single constraint $\sum_{j=1}^n a_{ij}x_j \geq b_i$. The solution space $P_i = \{\mathbf{x} \in \mathbb{R}^n \mid \sum_{j=1}^n a_{ij}x_j \geq b_i\}$ of this constraint is a (closed) **half space**.
- Half space is **convex**, i.e., for all $\mathbf{x}, \mathbf{y} \in P_i$ we have $\{\lambda\mathbf{x} + (1-\lambda)\mathbf{y} \mid \lambda \in [0,1]\} \subseteq P_i$.
- $P = \bigcap_{i=1}^m P_i$ is the *intersection of half spaces* of all constraints
- P is a **polytope**. It is convex (since half spaces are convex).

Example: Consider the following LP with solutions from $\mathbb{R}^2$.

$$\begin{array}{lrcl} \text{Maximize} & x_1 + x_2 & & \\ \text{subject to} & x_1 + 2x_2 & \leq & 3 \\ & 2x_1 + x_2 & \leq & 3 \\ & x_1 & \geq & 0 \\ & x_2 & \geq & 0 \end{array} \tag{5.3}$$

[Pic: Half spaces from constraints, solutions, hyperplanes, optimal corner solution]

Optimization of an LP:

- Consider the vector $\mathbf{c}$. All solutions $\mathbf{x}$ with $\mathbf{c}^T\mathbf{x} = 0$ have the same value 0.
- $\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{c}^T\mathbf{x} = 0\}$ is a hyperplane orthogonal to $\mathbf{c}$
- Generally, $\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{c}^T\mathbf{x} = \alpha\}$ is a hyperplane with all solutions of some value $\alpha \in \mathbb{R}$.
- All these hyperplanes are parallel and orthogonal to $\mathbf{c}$.
- Maximization: Pick a hyperplane farthest in the direction of $\mathbf{c}$ that contains a solution to $\mathbf{Ax} \geq \mathbf{b}$ (minimization: in direction of $-\mathbf{c}$)

[1] A major open problem in the area is if there exists a *strongly* polynomial-time algorithm, whose running time depends only on n and m, but not on the size of the numbers (assuming that arbitrarily large numbers can be added or multiplied in a single step).

- Three possibilities for LPs:
  (a) **Infeasible** LP, empty solution space
  (b) **Unbounded** LP, solution space unbounded (in the direction of optimization)
  (c) **Bounded** LP, solution space bounded (in the direction of optimization)

[Pic: Infeasible, unbounded, bounded, dependence on direction of $\mathbf{c}$]

Infeasible/unbounded conditions can be identified along the way in the algorithm(s).
We concentrate on bounded LPs.

- Consider the solution polytope P.
- A side (in $\mathbb{R}^n$) of P is called *facet*. It separates the inside of P from the rest of $\mathbb{R}^n$.
- A *ridge* is the intersection of two facets.
- ...
- How many facets have to intersect for a **vertex** or **corner** of P?
- Since $\mathbb{R}^n$ has n dimensions, we need at least n intersecting facets, i.e., n half spaces from different constraints to create a corner.
- The intersecting sides have to be *linearly independent*, i.e., the coefficient vectors $\mathbf{a}_i$ of the constraints must be linearly independent.
- Why are corners interesting? There is **always at least one optimal solution** in one of the **corners** of P.

[Pics: Facets and corners in $\mathbb{R}^2$ and $\mathbb{R}^3$, examples for linearly dependent constraints]

Definition 18. *A **corner** or **vertex** of a convex polytope P is a point $\mathbf{x}$ defined by the following two equivalent criteria:*

1. *$\mathbf{x}$ is a point where n linearly independent constraints are fulfilled exactly.*
2. *There is no vector $\mathbf{y} \in \mathbb{R}^n, \mathbf{y} \neq \mathbf{0}$ such that $\mathbf{x} + \mathbf{y} \in P$ and $\mathbf{x} - \mathbf{y} \in P$.*

There are also **degenerate** corners, where more than n constraints are fulfilled simultaneously. If there are several sets of n linearly independent constraints, we have a "set of corners" in a single point. These corners require a special treatment in algorithm and analysis below. It is technical (but not very insightful), and for the most part we will ignore degeneracy of corners in our overview here.

Theorem 37. *If a bounded LP with n variables has at least n linearly independent constraints, then at least one optimal solution $\mathbf{x}^*$ is a corner of the solution polytope P.*

Note that both conditions together are necessary and sufficient:

- If the polytope P is unbounded in the direction of optimization, there is no optimal solution at all.
- If the LP is bounded but does not have n linearly independent constraints, then the polytope P contains at least one infinite line. The set of optimal solutions is infinitely large (since no corner or vertex can exist).

We proceed to give a high-level description of the **Simplex Algorithm**. It is a local search algorithm that moves from one corner to the next in order to optimize the objective function. By doing so, it actually manages to find a **globally optimal solution**!
To apply local search, we need a notion of neighborhood.

Definition 19. *Two corners* $\mathbf{x}$ *and* $\mathbf{x}'$ *of a convex polytope* P *are* ***neighbors*** *if there are* $n-1$ *linearly independent constraints that are exactly fulfilled by* $\mathbf{x}$ *and* $\mathbf{x}'$*.*

The Simplex algorithm moves to a neighboring corner that improves the objective function. It also involves a consistent tie-breaking to overcome "plateus" in the objective function (e.g., at degenerate corners) while avoiding to re-visit previous corners.

While the canonical form is intuitive, the algorithm uses as input an LP in **standard form**, which is more useful from a technical perspective.

Definition 20. *A linear program (LP) in* ***standard form*** *with* n *variables and* m *constraints consists of a vector* $\mathbf{x}^T = (x_1, \ldots, x_n)$ *of variables, a vector* $\mathbf{c} \in \mathbb{Q}^n$ *of cost coefficients, a matrix* $\mathbf{A} \in \mathbb{Q}^{m \times n}$ *and a vector* $\mathbf{b} \in \mathbb{Q}^m$*. The goal is to*

$$\begin{aligned} \textit{Minimize} \quad & \mathbf{c}^T\mathbf{x} \\ \textit{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

Every LP in canonical form has an equivalent formulation in standard form:

- For every inequality $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$ we introduce a **slack variable** s_i and obtain the two new constraints

$$\sum_{j=1}^{n} a_{ij}x_j - s_i = b_i \qquad \text{and} \qquad s_i \geq 0$$

- The standard from requires $x_j \geq 0$ for every $i = 1, \ldots, n$.
  If $x_j \geq 0$ in canonical form, no adjustment needed.
  If $x_j \leq 0$ in canonical form, we replace x_j by $-x_j^-$ throughout, and $x_j^- \geq 0$.
  If x_j has no restriction in canonical form, we replace x_j by $x_j^+ - x_j^-$ throughout, and $x_j^+ \geq 0$, $x_j^- \geq 0$.

Example: Consider the following LP and its transformation into standard form:

$$\begin{aligned} \text{Minimize} \quad & 2x_1 + 4x_2 \\ \text{subject to} \quad & x_1 + x_2 \geq 3 \\ & 2x_1 + x_2 \geq 14 \\ & x_1 \geq 0 \end{aligned}$$

$$\begin{aligned} \text{Minimize} \quad & 2x_1 + 4x_2^+ - 4x_2^- \\ \text{subject to} \quad & x_1 + x_2^+ - x_2^- - s_1 = 3 \\ & 2x_1 + x_2^+ - x_2^- - s_2 = 14 \\ & x_1 \geq 0 \\ & x_2^+, x_2^- \geq 0 \\ & s_1, s_2 \geq 0 \end{aligned}$$

Some observations:

- We introduced three more variables. Instead of 2-dimensional $(x_1, x_2) \in \mathbb{R}^2$ we now look for a solution in 5-dimensional $(x_1, x_2^+, x_2^-, s_1, s_2) \in \mathbb{R}^5$.
- When introducing s_1 and s_2, we also obtain two equality constraints. This keeps the dimension of solution space the same, also maintains the shape of the space.

- The transformation $x_j = x_j^+ - x_j^-$ can truly increase the dimension of the space and could also create new corners!

Example:

$$\begin{array}{rll} \text{Minimize} & \mathbf{c}^T\mathbf{x} & \\ \text{subject to} & x_1 + x_2 & \leq 1 \\ & x_1 & \geq 0 \\ & x_2 & \geq 0 \end{array} \qquad \begin{array}{rll} \text{Minimize} & \mathbf{c}^T\mathbf{x} & \\ \text{subject to} & x_1 + x_2 - s_1 & = 1 \\ & x_1 & \geq 0 \\ & x_2 & \geq 0 \\ & s_1 & \geq 0 \end{array}$$

[Pic: Solution space = triangle, embedding in 3-dimensional space]

Corners in standard form:

- Suppose an LP in standard form has m linearly independent equality constraints.
- Each equality constraint reduces the dimension of P by one (by linear independence)
- If the solution polytope $P \neq \emptyset$, then $P \subseteq \mathbb{R}^n$ has $n - m$ dimensions.
- Consider a corner $\mathbf{x}$ of P. Since $\mathbf{x} \in P$, the m equality constraints are fulfilled exactly.
- $\mathbf{x}$ fulfills $n - m$ additional constraints exactly (i.e., $x_j = 0$ for $n - m$ variables x_j), i.e., there is a set $N \subseteq \{1, \ldots, n\}$ of $n - m$ variables, and $x_j = 0$, for every $j \in N$.

For a given corner $\mathbf{x}$, we define basic and non-basic variables:

- $j \in N$ is called a **non-basic component**, x_j a **non-basic variable**
- The remaining $B = \{1, \ldots, n\} \setminus N$ are called a **basis**
- $j \in B$ is a **basic component**, x_j a **basic variable**.
- Note: If $\mathbf{x}$ is non-degenerate, then $x_j > 0$ for every basic $j \in B$.
- We call the column vector $\mathbf{a}^j$ for basic $j \in B$ a **basis vector**.

More on basic variables:

- $\mathbf{A}_B$ is the $(m \times m)$-submatrix of basis vectors from $\mathbf{A}$
- $\mathbf{A}_N$ the submatrix of the remaining (non-basis) column vectors from $\mathbf{A}$
- Linear independent rows of equality constraints $\Rightarrow$ linear independent basis vectors $\Rightarrow$ $\mathbf{A}_B$ is regular.
- $\mathbf{x}_B$ denotes the subvector of $\mathbf{x}$ of basic variables, $\mathbf{x}_N$ the one for non-basic variables
- Note that $\mathbf{b} = \mathbf{A} \cdot \mathbf{x} = \mathbf{A}_B\mathbf{x}_B + \mathbf{A}_N\mathbf{x}_N = \mathbf{A}_B\mathbf{x}_B$, since $\mathbf{x}_N = \mathbf{0}$
- We see that $\mathbf{x}_B = \mathbf{A}_B^{-1}\mathbf{b} \geq \mathbf{0}$

Neighboring corners and the Simplex algorithm (very high-level):

- Two neighboring corners $\mathbf{x}$ and $\mathbf{x}'$ differ by one exactly fulfilled constraint
- Their sets N and N' are $N' = (N \setminus \{k\}) \cup \{j\}$ for some $j, k \in \{1, \ldots, n\}, j \neq k$
- The bases B and B' change correspondingly. How to execute a basis change?
- We use $\mathbf{c}$, $\mathbf{A}_B^{-1}$ and $\mathbf{a}^j$ to determine whether or not including j into the basis is profitable
- Adding j to the basis yields a direction, along which we can increase the solution value.
- We move the solution in this direction as much as possible.
- We can move forever $\rightarrow$ LP is unbounded.
- Otherwise, a basic variable starts to violate $x_k \geq 0$. Then k gets evacuated, and the basis change is complete.

Algorithm 22: Deterministic Rounding for Weighted Vertex Cover

Input: Graph $G = (V, E)$, vertex costs $w_v \geq 0$

$\mathbf{x}^* \leftarrow$ optimal solution of LP (5.4)
$C \leftarrow \emptyset$
for *each* $v \in V$ **do**
  if $x_v^* \geq 1/2$ **then** $C \leftarrow C \cup \{v\}$
return C

Running Times:

- Each iteration of Simplex runs in polynomial time (quite demanding: matrix inversion)
- There exist polytopes in $\mathbb{R}^n$ and initial solutions, such that the algorithm must visit $\Omega(2^n)$ corners. It is **not a polynomial-time algorithm** in terms of worst-case time complexity.
- Still, often very fast and widely applied in practice!
- Simplex was designed in the 1940s. In 1979, first algorithm with guaranteed polynomial running time: **Ellipsoid** (not very practical though)
- Since mid-1980s: poly-time **Interior-Point** methods. Good for large LPs when approximately optimal solutions suffice

5.3 Rounding and Approximation

5.3.1 Vertex Cover and Independent Set

Recall the Weighted Vertex Cover problem:

- Undirected graph $G = (V, E)$
- $x_v \in \{0, 1\}$ indicates if node $v \in V$ is included in vertex cover ($x_v = 1$) or not ($x_v = 0$).
- Node v has cost $w_v \geq 0$ when being in the cover.
- Each edge should be "covered", i.e., $x_u + x_v \geq 1$ for each $e = \{u, v\} \in E$.

Linear relaxation for Weighted Vertex Cover (c.f. (5.2)):

$$\begin{array}{lllll} \text{Minimize} & \sum_{v \in V} w_v x_v & & & \\ \text{subject to} & x_u + x_v & \geq & 1 & \text{for each } \{u, v\} \in E \\ & x_v & \in & [0, 1] & \text{for each } v \in V \end{array} \tag{5.4}$$

Consider the **Rounding** algorithm (Algorithm 22). It computes $\mathbf{x}^*$, an optimal solution to the LP-relaxation. It includes a vertex $v \in V$ into the vertex cover if and only if $x_v^* \geq 1/2$.

Theorem 38. *Algorithm 22 has approximation ratio 2.*

Proof. For every $\{u, v\} \in E$ we have $x_v^* + x_u^* \geq 1$, so either $x_u^* \geq 1/2$, or $x_v^* \geq 1/2$, or both. Hence, the set $C \subseteq V$ returned by the algorithm is indeed a feasible vertex cover.

What about the approximation ratio?

- Consider an optimal vertex cover C^* and the cover C computed by the algorithm.
- Since $\mathbf{x}^*$ solves the LP relaxation optimally, we know

$$\sum_{v \in V} w_v x_v^* \leq \sum_{v \in C^*} w_v.$$

- For every $v \in C$, we know that $1/2 \leq x_v^*$, so $1 \leq 2x_v^*$.
- Hence,

$$\sum_{v \in C} w_v \leq \sum_{v \in C} w_v \cdot 2x_v^* \leq 2 \cdot \sum_{v \in V} w_v x_v^* \leq 2 \cdot \sum_{v \in C^*} w_v \ .$$

□

Interestingly, the linear relaxation (5.4) is **half-integral**, i.e., every corner of the solution polytope P is a vector with entries $\mathbf{x} \in \{0, 1/2, 1\}^n$.

- Suppose $\mathbf{x}$ has entries $x_v \notin \{0, 1/2, 1\}$.
- We define $V_< = \{v \in V \mid 0 < x_v < 1/2\}$ and $V_> = \{v \in V \mid 1/2 < x_v < 1\}$ as the set of all non-half-integral entries.
- Consider a vector $\mathbf{y}$ defined by

$$y_v = \begin{cases} \varepsilon & \text{if } v \in V_< \\ -\varepsilon & \text{if } v \in V_> \\ 0 & \text{otherwise} \end{cases}$$

- If $\mathbf{x} + \mathbf{y} \in P$ and $\mathbf{x} - \mathbf{y} \in P$, then $\mathbf{x}$ cannot be a corner by Definition 18.
- By choosing $\varepsilon > 0$ small enough:
  → We ensure that $0 \leq x_v \pm y_v \leq 1$ for each $v \in V$.
  → Constraints $x_u + x_v \geq 1$ for all edges remain satisfied even for the vectors $\mathbf{x} \pm \mathbf{y}$.
- Hence, $\mathbf{x} + \mathbf{y} \in P$ and $\mathbf{x} - \mathbf{y} \in P$, so $\mathbf{x}$ cannot be a corner.

Suppose we want to apply the same idea for the Independent Set problem:

- $x_v \in \{0, 1\}$ indicates if node $v \in V$ is included in indep. set ($x_v = 1$) or not ($x_v = 0$).
- Node v has profit 1 when being in the independent set.
- Each edge should be "conflict-free", i.e., $x_u + x_v \leq 1$ for each $e = \{u, v\} \in E$.

LP-relaxation for Independent Set

$$\begin{array}{lrcll} \text{Maximize} & \displaystyle\sum_{v \in V} x_v & & & \\ \text{subject to} & x_u + x_v & \leq & 1 & \text{for each } \{u, v\} \in E \\ & x_v & \in & [0, 1] & \text{for each } v \in V \end{array} \tag{5.5}$$

Lemma 16. *There are instances of* Independent Set *with n vertices such that the fractional optimum $\mathbf{x}^*$ and the optimal independent set S^* differ in value by a factor of $\Omega(n)$:*

$$\sum_{v \in V} x_v^* \quad \geq \quad n/2 \cdot |S^*|.$$

Proof. For the complete graph with n vertices, we have $|S^*| = 1$. Setting $x^* = 1/2$ yields a feasible solution to (5.5) with value at least $n/2$. □

5.3.2 Matching and Total Unimodularity

Consider the Weighted MAX-MATCHING problem:

- A matching is essentially an "independent set of edges"
- $x_e \in \{0,1\}$ indicates if edge $e \in E$ is included in matching ($x_e = 1$) or not ($x_e = 0$).
- Edge e has profit w_e when being in the matching
- Each vertex should be "conflict-free", i.e., $\sum_{u:\{u,v\}\in E} x_{\{u,v\}} \leq 1$ for each $v \in V$.

LP-relaxation for Weighted MAX-MATCHING

$$\begin{aligned} \text{Maximize} \quad & \sum_{e\in E} w_e x_e & & & \\ \text{subject to} \quad & \sum_{u:\{u,v\}\in E} x_{\{u,v\}} & \leq & 1 & \text{for each } v \in V \\ & x_e & \in & [0,1] & \text{for each } e \in E \end{aligned} \tag{5.6}$$

Some comments:

- The constraint matrix $\mathbf{A}$ here is the node-edge incidence matrix of G.
- Similar to INDEPENDENT SET, the LP (5.6) can have strange solutions (e.g., in the complete graph take every edge with $x_e = 1/(n-1)$).
- However, optimal fractional and integral solutions always differ in value by at most a constant factor (Exercise).
- There are even better conditions for *bipartite graphs*.

Definition 21. *A matrix* $\mathbf{A}$ *is called* ***totally unimodular*** *if for every square submatrix* $\mathbf{B}$ *of* $\mathbf{A}$*, the determinant is* $\det(\mathbf{B}) \in \{-1, 0, 1\}$.

We state the following results without formal proofs.

Theorem 39. *If the constraint matrix* $\mathbf{A}$ *of an LP is totally unimodular and the vector* $\mathbf{b}$ *has only integral entries, then each corner of the polytope* $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq 0\}$ *has only integral coordinates.*

Theorem 40. *Consider the incidence matrix* $\mathbf{A}$ *of a graph* G*:*

1. *G undirected:* $\mathbf{A}$ *is totally unimodular if and only if G is bipartite.*
2. *G directed:* $\mathbf{A}$ *is always totally unimodular.*

Corollary 4. *The LP-relaxation* (5.6) *for bipartite graphs has an optimal solution that is integral and represents an optimal solution to Weighted* MAX-MATCHING*.*

5.3.3 Set Cover

Weighted SET COVER problem:

- Set $E = \{1, \ldots, m\}$ of elements, set system $\mathcal{S} \subseteq 2^E$, i.e., a family of n subsets of E
- Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ denote the subsets. Each subset S_j has a weight/cost $w_j \geq 0$.
- Consider a set $C \subseteq \{1, \ldots, n\}$ of subset indices and the collection $\{S_j \mid j \in C\}$.
- C is a *set cover* if for every element $e \in E$ there is at least one $j \in C$ such that $e \in S_j$.

- Put differently, the union $\bigcup_{j \in C} S_j = E$, i.e., it *covers* all of E.
- **Goal:** Find a set cover C with minimal total cost $w(C) = \sum_{j \in C} w_j$.

Example: Elements $E = \{1, \ldots, 10\}$, sets

- $S_1 = \{3, 4, 5, 8\}$
- $S_2 = \{1\}$
- $S_3 = \{1, 3, 5, 7, 9\}$
- $S_4 = \{2, 4, 10\}$
- $S_5 = \{2, 3, 4, 5, 6\}$
- $S_6 = \{2, 6, 10\}$

$\{S_1, S_3, S_4, S_5\}$ is a set cover. $\{S_1, S_3, S_6\}$ is another set cover.

Vertex Cover is a special case of Set Cover:

- Edge $e \longleftrightarrow$ Element e
- Vertex $v \longleftrightarrow$ Set $S_v = \{e \in E \mid e = \{u, v\}\}$ of incident edges/elements
- Then any vertex cover is a set cover and vice versa.

LP-relaxation for Weighted Set Cover (c.f. (5.4)). Intention: $x_j = 1$ if $j \in C$, 0 otherwise.

$$\begin{aligned} \text{Minimize} \quad & \sum_{j=1}^{n} w_j x_j & & & \\ \text{subject to} \quad & \sum_{j : e \in S_j} x_j & \geq & 1 & \text{for each } e \in E \\ & x_j & \in & [0,1] & \text{for each } j = 1, \ldots, n \end{aligned} \tag{5.7}$$

We can extend Algorithm 22 to obtain a feasible solution for Set Cover. The approximation ratio is $f = \max_{e \in E} |\{j \mid e \in S_j\}|$, the maximum number of sets that a single element is contained in. The adjustment of the algorithm and the proof are left as an exercise.

Corollary 5. *There is a deterministic rounding algorithm for Weighted* Set Cover *with approximation ratio f.*

Note that $f = 2$ for Vertex Cover, since every element (edge) is contained in at most 2 sets (sets of incident edges for incident vertices).

If f is large, the deterministic rounding approach becomes unattractive. Instead, we present a **Randomized Rounding** idea in Algorithm 23. While the algorithm is not always guaranteed to return a feasible set cover, we can show the following guarantee. Suppose C^* is an optimal set cover, and $w(C^*) = \sum_{j \in C^*} w_j$ is the optimal cost.

Theorem 41. *After $T = \lceil \ln m + 1 \rceil$ repetitions, the probability that C is a feasible set cover is at least $1 - 1/\mathsf{e}$. The expected cost of C is $\mathbb{E}[w(C)] \leq \lceil \ln m + 1 \rceil \cdot w(C^*)$.*

Proof. We first bound the probability that the algorithm produces a feasible set cover.

- Consider a single element $e \in E$. e is contained in k sets. To reduce notation, assume w.l.o.g. these are sets $S_1, \ldots, S_k$.

Algorithm 23: Randomized Rounding for Weighted Set Cover

Input: Set system $\mathcal{S}$, set costs $w_j \geq 0$

$\mathbf{x}^* \leftarrow$ optimal solution of LP (5.7)
$C \leftarrow \emptyset$
for $t = 1, \ldots, T$ **do**
  $C_t \leftarrow \emptyset$
  for $j = 1, \ldots, n$ **do** include j into C_t with prob. x_j^*
  $C \leftarrow C \cup C_t$
return C

- Probability that S_j is not chosen: $1 - x_j^*$.
- Probability that none of the k sets is chosen (and e is uncovered): $\prod_{i=1}^{k}(1 - x_i^*)$
- Observe that

$$\sqrt[k]{\prod_{i=1}^{k}(1-x_i^*)} \quad \leq \quad \frac{1}{k}\sum_{i=1}^{k}(1-x_i^*) = 1 - \frac{\sum_{i=1}^{k} x_i^*}{k} \quad \leq \quad 1 - \frac{1}{k}$$

  where the first inequality is the AM-GM inequality, and the second inequality is due to the constraint for e in LP (5.7).
- This implies that in one round

$$\Pr[e \text{ uncovered}] = \prod_{i=1}^{k}(1-x_i^*) \leq \left(1 - \frac{1}{k}\right)^k \leq \frac{1}{\mathsf{e}} \qquad \text{where } \mathsf{e} \approx 1.58.$$

- Hence, using $T \geq \ln m + 1$, we see

$$\Pr[e \text{ uncovered after } T \text{ repetitions}] \leq \left(\frac{1}{\mathsf{e}}\right)^T \leq \left(\frac{1}{\mathsf{e}}\right)^{\ln m} \cdot \left(\frac{1}{\mathsf{e}}\right) = \frac{1}{m \cdot \mathsf{e}} \ .$$

- Now apply a union bound: The probability that *at least one* (or more) of the elements is uncovered is at most the sum of probabilities that *any single one* of the elements is uncovered,

$$\Pr[\text{at least one } e \in E \text{ uncovered after } T \text{ repetitions}] \leq \sum_{e \in E} \frac{1}{m \cdot \mathsf{e}} = \frac{1}{\mathsf{e}} \ .$$

  Thus, with probability at least $1 - 1/\mathsf{e}$, all elements are covered after T repetitions.

[Pic: Union bound]

Now consider the expected cost of the solution. Since $\Pr[j \in C_t] = x_j^*$, we see

$$\mathbb{E}[w(C_t)] = \sum_j w_j \cdot \Pr[j \in C_t] = \sum_j w_j x_j^* \leq w(C^*).$$

Using $C = \bigcup_t C_t$, we can bound

$$\mathbb{E}[w(C)] \leq \sum_{t=1}^{T} \mathbb{E}[w(C_t)] \leq \sum_{t=1}^{T} w(C^*) = T \cdot w(C^*),$$

which proves the theorem. □

5.3.4 Integrality Gap

In randomized rounding algorithms, we use a fractional optimum for an LP and round it to an intergral solution. In this process, the deterioration between optimal fractional and integral solutions is an important value. It describes how much value we must sacrifice from the optimal fractional value when we want to obtain any integral solution.

Consider an optimization problem with instances $\mathcal{I}$, which we formulate as a linear program. For an instance $I \in \mathcal{I}$, consider the linear program $LP(I)$. We denote by $OPT_{frac}(I)$ the optimal value of any fractional solution and $OPT_{int}(I)$ the optimal value of any integral solution of $LP(I)$. The integrality gap (for a problem) is the (worst-case) ratio of values of integral and fractional optima.

Definition 22. *For a minimization problem, the **integrality gap** of $LP(I)$ for instance $I \in \mathcal{I}$ is*

$$\frac{OPT_{int}(I)}{OPT_{frac}(I)} \geq 1.$$

The integrality gap of an LP for a minimization problem is defined as

$$\sup_{I \in \mathcal{I}} \frac{OPT_{int}(I)}{OPT_{frac}(I)}.$$

For a maximization problem, the integrality gaps are defined by

$$\frac{OPT_{frac}(I)}{OPT_{int}(I)} \text{ for any instance } I \in \mathcal{I}, \qquad \text{and} \qquad \sup_{I \in \mathcal{I}} \frac{OPT_{frac}(I)}{OPT_{int}(I)} \text{ for the problem.}$$

Examples:

- Vertex Cover:
  The standard LP (5.4) has an integrality gap of at most 2, since Algorithm 22 obtains an integral 2-approximation w.r.t. the fractional optimum. A clique with n vertices shows that the integrality gap can be as high as $2 - 1/n$.
- Independent Set:
  Lemma 16 shows that the integrality gap for LP (5.5) is at least $n/2$. It can be observed that the gap in any instance is at most $n/2$.
- Set Cover:
  Theorem 41 shows that Algorithm 23 with constant probability obtains an integral solution that is at most $T = O(\ln m)$ times more expensive than the fractional optimum $\mathbf{x}^*$ of LP (5.7). In particular, this implies that such an integral solution must always exist. Hence, the integrality gap of LP (5.7) is at most $T = O(\ln m)$.

5.3.5 MaxSat

Weighted MaxSat problem:

- n boolean variables $x_1, \ldots, x_n$, k clauses $C_1, \ldots, C_k$
- Each clause C_i is an "OR-of-literals" (e.g. $(x_1 \vee \bar{x_3} \vee x_5 \vee \bar{x_9})$)
- Each clause has a weight $w_i \geq 0$.
- **Goal:** Set each variable x_j to true (1) or false (0) in order to maximize the sum of weights of satisfied clauses.

A **deterministic 2-approximation algorithm** is very easy to obtain (Exercise).

Towards an improved approximation ratio, consider an ILP formulation. We use a variable $y_j \in \{0,1\}$ that corresponds to boolean variable x_j. For each clause C_i, we have a variable z_i, where $z_i = 1$ if C_i is satisfied, and 0 otherwise. The ILP reads

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^{k} w_i z_i \\ \text{subject to} \quad & \sum_{x_j \in C_i} y_j + \sum_{\bar{x_j} \in C_i} (1 - y_j) \;\geq\; z_i \qquad \text{for each } i = 1, \ldots, k \\ & y_j, z_i \;\in\; \{0,1\} \quad \text{for each } j = 1, \ldots, n, i = 1, \ldots, k \end{aligned} \tag{5.8}$$

For the LP-relaxation we replace $y_j, z_i \in \{0,1\}$ by $0 \leq y_j, z_i \leq 1$.

We analyze two rounding algorithms.

1. **Simple rounding**: For each variable x_j, independently set $x_j = 1$ with probability $1/2$, and 0 otherwise.
2. **LP rounding**: Solve the LP-relaxation optimally, let $(\mathbf{y}^*, \mathbf{z}^*)$ denote an optimal solution. For each variable x_j, independently set $x_j = 1$ with probability y_j^*, and 0 otherwise.

Let W_i be a random variable that measures the "expected weight of a clause", i.e.,

$$W_i = \begin{cases} w_i & \text{if } C_i \text{ is satisfied} \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 17. *Consider a clause C_i with ℓ literals. Then*

1. $\mathbb{E}[W_i] = (1 - 1/2^\ell) \cdot w_i$ *for simple rounding, and*
2. $\mathbb{E}[W_i] \geq \big(1 - (1 - 1/\ell)^\ell\big) \cdot z_i^* \cdot w_i$ *for LP rounding.*

Proof. Note that $\mathbb{E}[W_i] = \Pr[C_i \text{ satisfied}] \cdot w_i$, so we just need to bound $\Pr[C_i \text{ satisfied}]$.

Since clause C_i has length ℓ, for simple rounding we observe

$$\Pr[C_i \text{ not satisfied}] = \left(\frac{1}{2}\right)^\ell \qquad \text{and} \qquad \Pr[C_i \text{ satisfied}] = 1 - \frac{1}{2^\ell}.$$

For LP rounding, we assume w.l.o.g. that $C_i = (x_1 \vee x_2 \vee \ldots \vee x_\ell)$, which implies

$$\Pr[C_i \text{ not satisfied}] = \prod_{j=1}^{\ell} (1 - y_j^*)$$

and

$$\begin{aligned}\Pr[C_i \text{ satisfied}] &= 1 - \prod_{j=1}^{\ell}(1-y_j^*) \\ &\geq 1 - \left(\frac{1}{\ell}\sum_{j=1}^{\ell}(1-y_j^*)\right)^{\ell} = 1 - \left(1 - \frac{\sum_{j=1}^{\ell} y_j^*}{\ell}\right)^{\ell} \\ &\geq 1 - \left(1 - \frac{z_i^*}{\ell}\right)^{\ell} \\ &\geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.\end{aligned}$$

Here the first inequality follows from the AM-GM inequality (similar as in the proof of Theorem 41). The second one is due to the LP constraint for z_i^*. Finally, the last inequality follows, since $1 - \left(1 - \frac{z}{\ell}\right)^{\ell}$ is concave in $z \in [0,1]$ – it lies above the direct line between $1 - \left(1 - \frac{0}{\ell}\right)^{\ell} = 0$ and $1 - \left(1 - \frac{1}{\ell}\right)^{\ell}$. The point $\left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*$ corresponds to z_i^* on that direct line. □

[Pic: Concave function, direct line, point for z_i^*]

Comparing simple and LP rounding:

- For clauses with length $\ell \geq 3$, simple rounding seems much better. $1 - (1 - 1/\ell)^{\ell}$ goes to $1 - 1/\mathsf{e}$, whereas $(1 - 1/2^{\ell})$ goes to 1, for large $\ell \to \infty$.
- If there are only clauses with $\ell = 2$, we have $y_j^* = 1/2$, and the algorithms would be the equivalent.
- For clauses with $\ell = 1$, LP rounding is usually better.

For the **Best-of-Rounding** algorithm, we run once simple rounding and once LP rounding. From the two solutions, we return the one that yields the higher value.

Theorem 42. *Best-of-Rounding has approximation ratio* 4/3*.*

Proof. We underestimate the expected weight of the final solution of the algorithm: Assume that each of the two solutions is chosen uniformly at random with probability 1/2.

Then for W_i we obtain

$$\begin{aligned}\mathbb{E}[W_i] &\geq \frac{1}{2} \cdot \left(1 - \frac{1}{2^{\ell}}\right) \cdot w_i + \frac{1}{2} \cdot \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^* \cdot w_i \\ &\geq w_i \cdot z_i^* \cdot \frac{1 - \frac{1}{2^{\ell}} + 1 - \left(1 - \frac{1}{\ell}\right)^{\ell}}{2} \\ &\geq w_i \cdot z_i^* \cdot \frac{3}{4}\end{aligned}$$

since $1 - \frac{1}{2^{\ell}} + 1 - \left(1 - \frac{1}{\ell}\right)^{\ell} \geq 3/2$ for all $\ell \geq 1$.

Hence, overall

$$\mathbb{E}\left[\sum_{i \text{ satisfied}} w_i\right] = \sum_{i=1}^{k} \mathbb{E}[W_i] \geq \frac{3}{4} \cdot \sum_{i=1}^{k} w_i \cdot z_i^* \geq \frac{3}{4} \cdot \sum_{i \in C^*} w_i \ .$$

□

Finally, let us inspect the integrality gap of the LP relaxation of ILP (5.8). Consider the following instance:

$$(x_1 \vee x_2), (x_1 \vee \bar{x_2}), (\bar{x_1} \vee x_2), (\bar{x_1} \vee \bar{x_2})$$

where the weights of all clauses are $w_i = 1$. The fractional optimum has total weight 4, for integral solutions the best value is 3. Hence, the integrality gap is at least 4/3. It is also at most 4/3 (Why?).

5.3.6 Routing and Path Selection

Consider the Path Selection problem, a basic problem in computer networks. We want to route a given set of messages or packets and minimize the overlap of the chosen routes.

- Directed graph $G = (V, E)$
- r **commodities** given by source nodes $s_1, \ldots, s_r$ and sink nodes $t_1, \ldots, t_r$ in G
- A **path system** is a set of r paths $\{P_1, \ldots, P_r\}$, where P_i starts in s_i and ends in t_i
- **Congestion** of a path system is the maximum number of paths any edge $e \in E$ is contained in, i.e., $\max_{e \in E} |\{i \mid e \in P_i\}|$
- **Goal**: Find a path system with smallest congestion

[Pic: Example, graph, sources and sinks, path system, congestion]

We construct an ILP formulation:

- Variable $x_i(e)$ for each $i = 1, \ldots, r$ and $e \in E$. Interpretation: $x_i(e) = 1$ if P_i uses e, 0 otherwise.
- We formulate a necessary constraint $C_{i,w}$ for each node $w \in V$ and each commodity i:

$$\sum_{e \,:\, e \text{ starts in } w} x_i(e) - \sum_{e \,:\, e \text{ ends in } w} x_i(e) = \begin{cases} 0 & w \neq s_i, t_i, \\ 1 & w = s_i, \\ -1 & w = t_i. \end{cases} \tag{5.9}$$

  For each node $w \neq s_i, t_i$, path P_i arrives at and leaves from the node the same number of times. For the source s_i, path P_i has to leave exactly one more time that it returns to s_i. P_i comes to t_i exactly one more time than it leaves t_i.
- Minimize the objective function $\max_{e \in E} \sum_{i=1}^{r} x_i(e)$, which is not linear. We express this as linear function with additional constraints.

ILP for Path Selection:

$$\begin{array}{rlll} \text{Minimize} & W & & \\ \text{subject to} & \sum_{i=1}^{r} x_i(e) & \leq W & \text{for each } e \in E \\ & \text{Constraints } C_{i,w} \text{ in (5.9)} & & \text{for each } i = 1, \ldots, r, w \in V \\ & x_i(e) & \in \{0, 1\} & \end{array} \tag{5.10}$$

Algorithm 24: Randomized Rounding for Path Selection

Input: Directed graph $G = (V, E)$ including sources $s_1, \ldots, s_r$ and sinks $t_1, \ldots, t_r$

$\mathbf{x}^* \leftarrow$ optimal solution of LP relaxation of (5.10)
for *each commodity* $i = 1, \ldots, r$ **do**
    Decompose $\mathbf{x}_i^*$ into paths $P_{i,1}, P_{i,2}, \ldots$ with values $\alpha_{i,1}, \alpha_{i,2}, \ldots$ resp.
    Choose $P_{i,j}$ with probability $\alpha_{i,j}$
return path system with the chosen path for each commodity

The LP relaxation is obtained by replacing $x_i(e) \in \{0, 1\}$ by $0 \leq x_i(e) \leq 1$ for each $e \in E$.

How to round this LP? Where are the paths in the optimal solution $\mathbf{x}^*$?

- $\mathbf{x}^*$ results in a **flow** for commodity i:
  $\rightarrow$ Source s_i emits a flow of 1, target t_i takes in a flow of 1.
  $\rightarrow$ For every other node $w \neq s_i, t_i$, we have "flow conservation".
  $\rightarrow$ Every edge has non-negative flow and a flow capacity of 1.
- There must be at least one path from s_i to t_i of edges with non-negative flow.
- Pick an arbitrary such path, denote it by $P_{i,1}$. Let $\alpha_{i,1} = \min_{e \in P_{i,1}} x_i(e)$.
- Decrease by α_1 all $x_i(e)$ for $e \in P_{i,1}$ (all remain non-negative, at least one becomes 0).
- This decreases overall flow in-/outgoing at source/sink to $1 - \alpha_{i,1}$, resp. As long as remaining flow out of s_i is non-zero, argument can be repeated.
- Repeated application results in a *path decomposition* of $\mathbf{x}^*$ with paths $P_{i,1}, P_{i,2}, \ldots$ and flow values $\alpha_{i,1}, \alpha_{i,2}, \ldots$
- The process terminates after at most $|E|$ repetitions.
- Note: $\sum_j \alpha_{i,j} = 1$ and $\alpha_{i,j} \in [0, 1]$. The values $\alpha_{i,j}$ represent a probability distribution over paths $P_{i,j}$.

[Pic: Example network, flow, paths $P_{i,j}$, values for $\alpha_{i,j}$]

In Algorithm 24, we use the values $\alpha_{i,j}$ for rounding. The resulting algorithm obtains a good performance in congested networks, in which the optimal congestion W^* is not too small.

Theorem 43. *Let W^* be the smallest congestion of any path system. With probability at least $1 - \varepsilon$, Algorithm 24 obtains a path system with congestion at most*

$$W^* + \sqrt{3 \cdot W^* \cdot \ln(n^2/\varepsilon)} \ .$$

Proof. We first consider a single edge e and the number of paths going over e.

- For each path P_i we consider a random variable X_i^e
- $X_i^e = 1$ if P_i is chosen to go over e, and 0 otherwise.
- The (random) number of paths going over e is $W^e = \sum_{i=1}^r X_i^e$.
- Clearly, for path P_i we have

$$\Pr[X_i^e = 1] = \sum_{j : e \in P_{i,j}} \alpha_{i,j} \leq x_i^*(e).$$

- The expected congestion of edge e is

$$\mathbb{E}[W^e] = \sum_{i=1}^{r} \sum_{j:e\in P_{i,j}} \alpha_{i,j} \leq \sum_{i=1}^{r} x_i^*(e) \leq W_f^*,$$

  where $W_f^* = \max_e \sum_i x_i^*(e)$ is the optimal congestion of the fractional optimum $\mathbf{x}^*$.
- Hence, in expectation every edge has congestion at most W_f^*.
- What is the probability that the actual congestion significantly exceeds its expectation? Note that for edge e, the X_i^e are independent Bernoulli variables. For W^e we can apply a Chernoff bound:

$$\Pr\left[W^e \geq (1+\delta)W_f^*\right] \leq \mathsf{e}^{-W_f^*\cdot\delta^2/3} \ .$$

- We set

$$\delta = \sqrt{\frac{3\cdot\ln(n^2/\varepsilon)}{W_f^*}}$$

  and use a union bound to capture the probability that the maximum congestion of any edge e exceeds the bound:

$$\begin{aligned}\Pr\left[\max_{e\in E} W^e \geq (1+\delta)W_f^*\right] &\leq \sum_{e\in E} \Pr\left[W^e \geq (1+\delta)W_f^*\right] \\ &\leq \sum_{e\in E} \mathsf{e}^{-W_f^*\cdot\delta^2/3} = |E|\cdot\frac{\varepsilon}{n^2} < \varepsilon \ .\end{aligned}$$

- Hence, with probability at most ε the rounded solution has a congestion of at least $(1+\delta)W_f^*$. In turn, with probability at least $1-\varepsilon$, the congestion is at most

$$(1+\delta)\cdot W_f^* = W_f^* + \sqrt{3\cdot W_f^*\cdot\ln(n^2/\varepsilon)} < W^* + \sqrt{3\cdot W^*\cdot\ln(n^2/\varepsilon)} \ .$$

□

5.3.7 Unrelated Machine Scheduling

Unrelated Machine Scheduling generalizes Makespan Scheduling to non-identical machines:

- m (not necessarily identical) machines, n tasks
- Task $i \in [n]$ has processing time $p_{ij} \in \mathbb{N}$ on machine $j \in [m]$
- Assign each task i (completely) to some machine $j \in [m]$
- Load of machine j is $\ell_j = \sum_{i \text{ on } j} p_{ij}$
- Maximum load $\max_{j\in[m]} \ell_j$ is the makespan of the assignment
- **Goal:** Assign tasks to machines to minimize the makespan

Representation: Complete bipartite graph G, node sets are tasks and machines, edge weights are processing times. Assignment is a one-to-many matching in G.

[Pic: Complete bipartite, edge-weighted graph. Assignment as one-to-many matching]

The assumption that all $p_{ij} \in \mathbb{N}$ is w.l.o.g.: We could allow $p_{ij} \in \mathbb{Q}$ for all $i \in [n], j \in [m]$, since technically we assume that the input is representable by a finite bit vector on a Turing machine. When we multiply each processing time with the common denominator of all times, we obtain $p_{ij} \in \mathbb{N}$. This multiplication increases the representation size of the processing times only by a polynomial factor.

Towards an ILP for Unrelated Machine Scheduling, we assume $x_{ij} = 1$ if task i is assigned to machine j, and 0 otherwise. Our ILP reads

$$\begin{aligned} \text{Minimize} \quad & t \\ \text{subject to} \quad & \sum_{i=1}^{n} x_{ij} p_{ij} \leq t && \text{for each } j \in [m] \\ & \sum_{j=1}^{m} x_{ij} = 1 && \text{for each } i \in [n] \\ & x_{ij} \in \{0,1\} && \text{for each } i \in [n], j \in [m] \end{aligned} \tag{5.11}$$

Some remarks:

- First constraints: t is at least the makespan (minimization yields $t =$ makespan)
- Second constraints: Every task i assigned to exactly one machine.
- LP relaxation: Replace $x_{ij} \in \{0,1\}$ by $x_{ij} \geq 0$ (then second constraints imply $x_{ij} \leq 1$)
- The LP relaxation has a **huge integrality gap**[2]!

Let us apply the LP in a somewhat orthogonal way. We **guess an upper bound** T for the makespan and **prune** "assignment edges" (i, j) with $p_{ij} > T$. None of these edges can be used in an integral assignment with makespan T.

Now given the upper bound T on the makespan, let $E_T = \{\{i, j\} \in [n] \times [m] \mid p_{ij} \leq T\}$ be the pruned set of assignment edges. We find a feasible *fractional* assignment in the pruned graph $G_T = ([n] \cup [m], E_T)$. The following set of inequalities describe the polytope of feasible fractional assignments with makespan at most T in the pruned instance. We denote it by $LP(T)$:

$$\begin{aligned} \sum_{i:\{i,j\} \in E_T} x_{ij} p_{ij} &\leq T && \text{for each } j \in [m] \\ \sum_{j:\{i,j\} \in E_T} x_{ij} &= 1 && \text{for each } i \in [n] \\ x_{ij} &\geq 0 && \text{for each } \{i, j\} \in E_T \end{aligned} \tag{5.12}$$

Consider Algorithm 25. Some comments:

- First, build a greedy schedule for a simple upper bound T_g on the optimal makespan.
- Indeed, then T_g/m must be a lower bound on the optimal makespan (Exercise)
- Then find the smallest integer T^*, for which $LP(T)$ (5.12) has a feasible solution, by using binary search for T over the interval $[T_g/m, T_g]$.
- Pick a corner $\mathbf{x}$ of polytope $LP(T^*)$, assign all tasks that are integrally assigned in $\mathbf{x}$.

[2]Instance with single task and $p_{1j} = 1$ for every machine $j \in [m]$. Fractional optimum breaks task into m pieces of $x_{1j} = 1/m$, makespan is $1/m$. Integrality gap at least m.

Algorithm 25: Parametric Pruning for Unrelated Machine Scheduling

Input: Tasks, machines, processing times

for *each task* i **do**
  assign i greedily to $j_i = \arg\min_{j \in [m]} p_{ij}$ with smallest processing time for i
Let T_g denote the makespan of the greedy assignment
Perform binary search over the interval $[T_g/m, T_g]$ to find smallest integer T^*, for which $LP(T)$ (5.12) has a feasible solution
$\mathbf{x} \leftarrow$ corner of $LP(T^*)$
for *each entry* $x_{ij} = 1$ **do** assign task i to machine j
Construct graph $H(\mathbf{x})$ of fractionally assigned tasks and their machines
$M \leftarrow$ perfect matching in $H(\mathbf{x})$
for *each* $\{i, j\} \in M$ **do** assign task i to machine j
return assignment of tasks to machines

- For the remaining tasks, round the solution by constructing a graph $H(\mathbf{x})$ composed of these tasks and the machines they are fractionally assigned to.
- Find a perfect matching M of tasks and machines in $H(\mathbf{x})$, assign tasks as in M.

The main result in this section is the following.

Theorem 44. *Algorithm 25 runs in polynomial time and has approximation ratio 2.*

A coarse upper bound on the running time of the binary search are $O\left(\log \sum_{i,j} p_{ij}\right)$ rounds (since $[T_g/m, T_g] \subset [0, \sum_{i,j} p_{ij}]$). This is at most linear in the input size – the representation size of processing times already is at least $\sum_{i,j} \log p_{ij}$. The feasibility test of $LP(T)$ and the remaining steps can be executed in polynomial time. Hence, the running time is polynomial.

Towards a proof of approximation, we first establish useful properties for corner solutions $\mathbf{x}$ of polytope (5.12). We then discuss the definition of $H(\mathbf{x})$ and show it always contains a perfect matching. As the last step, we prove the approximation ratio.

Lemma 18. *Every corner* $\mathbf{x}$ *of* $LP(T)$ *has at most* $n + m$ *non-zero variables.*

Proof. We denote by $r = |E_T|$ the number of variables in $LP(T)$.

- Definition of corner: $\mathbf{x}$ satisfies r linearly independent constraints exactly
- $n + m$ constraints of the first two types in (5.12)
- At least $r - (n + m)$ constraints "$x_{ij} \geq 0$" fulfilled exactly
- At least $r - (n + m)$ many $x_{ij} = 0$, so at most $n + m$ non-zero variables.

□

Corollary 6. *In every corner* $\mathbf{x}$*, at least* $n - m$ *tasks are assigned integrally.*

Proof. We denote by $k = |\{\, \{i, j\} \in E_T \mid x_{ij} > 0\}|$ the number of non-zero variables.

- Previous lemma: $k \leq n + m$

- Task i fractional $\rightarrow$ at least two $x_{ij} > 0$. Task i integral $\rightarrow$ one $x_{ij} > 0$.
- Suppose we have α fractional and $n - \alpha$ integral tasks.
- Then $2\alpha + n - \alpha \leq k \leq n + m$, which implies $\alpha \leq m$

Number of integrally assigned tasks is $n - \alpha \geq n - m$. □

We define the notion of a **pseudo-forest**:

- A pseudo-tree is a connected graph with k vertices and at most k edges.
  $\rightarrow$ It is either a tree or a tree plus one edge.
- A pseudo-forest is a graph where every connected component is a pseudo-tree.

[Pic: Example pseudo-tree, pseudo-forest]

We define two graphs. In $G(\mathbf{x}) = ([n] \cup [m], E(\mathbf{x}))$ we have edges $E(\mathbf{x}) = \{\{i, j\} \in E_T \mid x_{ij} > 0\}$ corresponding to non-zero assignments of $\mathbf{x}$. In $H(\mathbf{x})$ we restrict $G(\mathbf{x})$ to tasks that are *fractionally* assigned in $\mathbf{x}$, i.e., to tasks i for which there are at least two entries $0 < x_{ij} < 1$. $H(\mathbf{x})$ contains all fractionally assigned tasks, their incident edges and their adjacent machines from $G(\mathbf{x})$.

We first consider $G(\mathbf{x})$.

Lemma 19. *In every corner* $\mathbf{x}$*, the graph* $G(\mathbf{x})$ *is a pseudo-forest.*

Proof. Consider a connected component C of $G(\mathbf{x})$.

- Restrict $\mathbf{x}$ to $\mathbf{x}_C$ containing only entries for the machines and tasks of C
- Let $\mathbf{x}_{-C}$ be the remaining entries of $\mathbf{x}$, i.e., $\mathbf{x} = (\mathbf{x}_C, \mathbf{x}_{-C})$
- We define $LP_C(T)$ as (5.12) restricted to C – more precisely, in $LP_C(T)$ we have only the constraints for tasks and machines of C, where in each constraint we sum only over the variables x_{ij} for i and j both in C.
- Clearly, $\mathbf{x}_C$ is a feasible solution for $LP_C(T)$.
- We claim that $\mathbf{x}_C$ is also a *corner* of $LP_C(T)$.
- Suppose not – $\mathbf{x}_C$ is an interior solution of $LP_C(T)$. Then $\mathbf{x}_C = \lambda \mathbf{x}_C^{(1)} + (1 - \lambda)\mathbf{x}_C^{(2)}$ for two feasible solutions $\mathbf{x}_C^{(1)}$ and $\mathbf{x}_C^{(2)}$ for $LP_C(T)$
- The extensions $\mathbf{x}^{(1)} = (\mathbf{x}_C^{(1)}, \mathbf{x}_{-C})$ and $\mathbf{x}^{(2)} = (\mathbf{x}_C^{(2)}, \mathbf{x}_{-C})$ are feasible for $LP(T)$
- Hence, $\mathbf{x} = \lambda \mathbf{x}^{(1)} + (1 - \lambda)\mathbf{x}^{(2)}$, so $\mathbf{x}$ is an interior solution for $LP(T)$ and not a corner – a contradiction!

This shows that for every connected component C, the vector $\mathbf{x}_C$ is a corner of $LP_C(T)$.

- Suppose C has n_C tasks, m_C machines, so $n_C + m_C$ nodes.
- $\mathbf{x}_C$ is corner solution, so by Lemma 18, $\mathbf{x}_C$ has at most $n_C + m_C$ non-zero variables.
- Edges in C are non-zero variables $\rightarrow$ C has at most $n_C + m_C$ edges and is connected
- C is a pseudo-tree!

Every component of $G(\mathbf{x})$ is a pseudo-tree. □

Now consider $H(\mathbf{x})$.

Lemma 20. *In every corner* $\mathbf{x}$*,* $H(\mathbf{x})$ *is a pseudo-forest and has a perfect matching* M*.*

Proof. First we show it is a pseudo-forest:

- Each task assigned integrally in $\mathbf{x}$ has exactly one incident edge in $G(\mathbf{x})$.
- Remove these tasks with their edges. This results in $H(\mathbf{x})$.
- For every component of $G(\mathbf{x})$, the number of edges removed equals the number of nodes (tasks) removed.
- Hence, every component of $H(\mathbf{x})$ remains a pseudo-tree.

Now consider a component of $H(\mathbf{x})$.

- Each task has degree at least 2 in $H(\mathbf{x})$. All leaves must be machines.
- Construct a matching M iteratively "bottom-up":
  Match a leaf machine to the single incident task. Remove both machine and task.
- Thereby, at each point in time, all leaves remain machines.
- Finally, either all tasks are matched to unique machines or we encounter a cycle.
- Cycle is even (since $H(\mathbf{x})$ is bipartite). Match tasks and machines alternatingly.

Hence, $H(\mathbf{x})$ indeed has a perfect matching. □

[Pic: Example graph, build matching "bottom-up" in each pseudo-tree]

Finally, we complete the proof of Theorem 44.

Proof of Theorem 44. Consider the optimal makespan $T_I^* \in \mathbb{N}$ of any integral assignment.

- The pruning for $LP(T)$ removes options, all of which are impossible for an integral assignment of makespan T.
- Any optimal integral assignment of makespan T_I is a feasible solution for $LP(T_I)$.
- T^* is the smallest integer s.t. $LP(T)$ is feasible $\rightarrow$ we have $T^* \leq T_I^*$.

Consider the makespan $T(\mathbf{x})$ of the corner $\mathbf{x}$ computed by our algorithm.

- Clearly, $T(\mathbf{x}) \leq T^*$, since $\mathbf{x}$ is feasible for $LP(T^*)$.
- Assigning only integral tasks of $\mathbf{x}$ yields $\ell_j \leq T(\mathbf{x})$ for each machine by restriction
- Using the perfect matching M, each machine j receives at most one more task i
- Processing time of this task is $p_{ij} \leq T^*$, because $\{i, j\} \in E_{T^*}$ survived the pruning

In total, every machine has load $\ell_j \leq T(\mathbf{x}) + T^* \leq 2 \cdot T^* \leq 2 \cdot T_I^*$. □

Chapter 6

Primal-Dual Algorithms

6.1 Duality

Linear programs have a possibly surprising duality structure – they always arise in pairs. The *dual LP* results from the task to find good bounds on the optimal solution value of an LP (here called the *primal* LP). Indeed, finding the best bound on the optimal value for the primal LP can be formulated as another, the *dual* LP. Primal and dual LPs are tightly connected. Duality yields a lot of elegant structure that is very useful for the design of approximation algorithms.

Suppose we want to find a good *lower bound* on the optimal value of this (primal) LP:

$$\begin{array}{rrcl} \text{Minimize} & 7x_1 + x_2 + 5x_3 & & \\ \text{subject to} & x_1 - x_2 + 3x_3 & \geq & 10 \\ & 5x_1 + 2x_2 - x_3 & \geq & 6 \\ & x_1, x_2, x_3 & \geq & 0 \end{array} \tag{6.1}$$

We use the constraints to generate lower bounds:

- The first constraint implies a simple lower bound of 10 for any feasible solution

  $$10 \leq x_1 - x_2 + 3x_3 \leq 7x_1 + x_2 + 5x_3$$

  since all x_1, x_2, x_3 are non-negative.
- Combining two constraints, we can increase the bound to 16:

  $$10 + 6 \leq (x_1 - x_2 + 3x_3) + (5x_1 + 2x_2 - x_3) = 6x_1 + x_2 + 2x_3 \leq 7x_1 + x_2 + 5x_3$$

- Looking into the coefficients more closely, we can strengthen the bound to 26:

  $$2 \cdot 10 + 1 \cdot 6 \leq 2 \cdot (x_1 - x_2 + 3x_3) + 1 \cdot (5x_1 + 2x_2 - x_3) = 7x_1 + 5x_3 \leq 7x_1 + x_2 + 5x_3$$

What is the optimal set of coefficients y_1, y_2 such that

$$y_1 \cdot 10 + y_2 \cdot 6 \leq y_1 \cdot (x_1 - x_2 + 3x_3) + y_2 \cdot (5x_1 + 2x_2 - x_3) \leq 7x_1 + x_2 + 5x_3?$$

- $y_1, y_2 \geq 0$, since otherwise the inequalities would flip.
- $y_1 + 5y_2 \leq 7$: Our lower bound shall count less of $x_1 \geq 0$ than the objective function.
- Similar constraints for x_2 and x_3.
- This yields a new optimization problem.

Find the best coefficients to maximize the lower bound for the optimal value of LP (6.1).

$$\begin{array}{rrcl} \text{Maximize} & 10y_1 + 6y_2 & & \\ \text{subject to} & y_1 + 5y_2 & \leq & 7 \\ & -y_1 + 2y_2 & \leq & 1 \\ & 3y_1 - 2y_2 & \leq & 5 \\ & y_1, y_2 & \geq & 0 \end{array} \tag{6.2}$$

This is the **dual LP** for the (primal) LP (6.1). Some remarks:

- Coefficients in objective of dual = coefficients in right-hand sides of primal.
- Coefficients in right-hand sides of dual = coefficients in objective of primal.
- Rows of primal constraint matrix = columns of dual constraint matrix.
- Minimization in primal → Maximization in dual
- By applying a similar reasoning to the dual LP (6.2), we recover the primal LP 6.1. **The dual of the dual is the primal**!

More generally, consider an LP in canonical form with $\mathbf{c} \in \mathbb{Q}^n$, $\mathbf{b} \in \mathbb{Q}^m$, $\mathbf{A} \in \mathbb{Q}^{m \times n}$, in which we also have $\mathbf{x} \geq \mathbf{0}$. Then, by the same reasoning as above, the pairs of primal and dual are

$$\begin{array}{rrcl} \text{Minimize} & \mathbf{c}^T\mathbf{x} & & \\ \text{subject to} & \mathbf{A}\mathbf{x} & \geq & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array} \qquad \begin{array}{rrcl} \text{Maximize} & \mathbf{y}^T\mathbf{b} & & \\ \text{subject to} & \mathbf{y}^T\mathbf{A} & \leq & \mathbf{c}^T \\ & \mathbf{y} & \geq & \mathbf{0} \end{array} \tag{6.3}$$

The set of dual constraints can be written as $(\mathbf{y}^T\mathbf{A})^T \leq (\mathbf{c}^T)^T$, which is $\mathbf{A}^T\mathbf{y} \leq \mathbf{c}$, the form we chose in LP (6.2). We will stick to the former notation for the next argument.

How are primal and dual solutions related?

1. For every solution of the primal LP we have $\mathbf{x} \geq \mathbf{0}$ and $\mathbf{A}\mathbf{x} \geq \mathbf{b}$. For every dual solution we have $\mathbf{y} \geq \mathbf{0}$. This implies $\mathbf{y}^T\mathbf{A}\mathbf{x} \geq \mathbf{y}^T\mathbf{b}$.
2. For every solution of the dual LP we have $\mathbf{y} \geq \mathbf{0}$ and $\mathbf{y}^T\mathbf{A} \leq \mathbf{c}^T$. For every primal solution, we have $\mathbf{x} \geq \mathbf{0}$. This implies $\mathbf{y}^T\mathbf{A}\mathbf{x} \leq \mathbf{c}^T\mathbf{x}$.
3. Plugging together the two bounds, we see that

$$\mathbf{y}^T\mathbf{b} \leq \mathbf{y}^T\mathbf{A}\mathbf{x} \leq \mathbf{c}^T\mathbf{x}.$$

The value of *every dual solution* is a lower bound on the value of *every primal solution*!

[Pic: primal solution values, dual solution values]

What if the maximization LP in (6.3) is the primal LP?

- We would want an *upper bound* on the optimal objective value.
- Use a linear combination of the constraints to generate an upper bound.

- Dual LP: Find an optimal combination of constraints for the smallest upper bound.
- Same three steps as above show: The dual is the minimization LP in (6.3).
- The dual of the dual is the primal!

We generically obtain dual linear programs by the following "recipe". A minimization LP turns into a maximization LP and vice versa. We exchange the role of right-hand side $\mathbf{b}$ and objective function $\mathbf{c}$. Variables in one LP correspond to constraints in the other:

- $y_i \geq 0$ corresponds to constraint $\mathbf{a}_i\mathbf{x} \geq b_i$,
- $y_i \leq 0$ corresponds to constraint $\mathbf{a}_i\mathbf{x} \leq b_i$, and
- y_i free corresponds to constraint $\mathbf{a}_i\mathbf{x} = b_i$.

and

- $x_j \geq 0$ corresponds to constraint $\mathbf{y}^T\mathbf{a}^j \leq c_j$,
- $x_j \leq 0$ corresponds to constraint $\mathbf{y}^T\mathbf{a}^j \geq c_j$, and
- x_j free corresponds to constraint $\mathbf{y}^T\mathbf{a}^j = c_j$.

Here $\mathbf{a}^j$ is the column vector of $\mathbf{A}$ with coefficients for variable x_j.

Schematically primal and dual relate to each other as follows:

$$\begin{array}{rrcl} \text{Minimize} & \mathbf{c}^T\mathbf{x} & & \\ \text{subject to} & \mathbf{a}_i\mathbf{x} & \geq & b_i \\ & \mathbf{a}_i\mathbf{x} & \leq & b_i \\ & \mathbf{a}_i\mathbf{x} & = & b_i \\ & x_j & \geq & 0 \\ & x_j & \leq & 0 \\ & x_j & & \text{free} \end{array} \quad \longleftrightarrow \quad \begin{array}{rrcl} \text{Maximize} & \mathbf{y}^T\mathbf{b} & & \\ \text{subject to} & y_i & \geq & 0 \\ & y_i & \leq & 0 \\ & y_i & & \text{free} \\ & \mathbf{y}^T\mathbf{a}^j & \leq & c_j \\ & \mathbf{y}^T\mathbf{a}^j & \geq & c_j \\ & \mathbf{y}^T\mathbf{a}^j & = & c_j \end{array} \tag{6.4}$$

Note that the transformation **goes in both directions** – it works from "left to right" and from "right to left" (i.e., the dual of the dual is the primal).

Examples: An LP in standard form

$$\begin{array}{rrcl} \text{Minimize} & 13x_1 + 10x_2 + 6x_3 & & \\ \text{subject to} & 5x_1 + x_2 + 3x_3 & = & 8 \\ & 3x_1 + x_2 & = & 3 \\ & x_1, x_2, x_3 & \geq & 0 \end{array} \qquad \begin{array}{l} \text{Minimize } \begin{bmatrix} 13 & 10 & 6 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ \text{s.t. } \begin{bmatrix} 5 & 1 & 3 \\ 3 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 \\ 3 \end{bmatrix} \\ \mathbf{x} \geq \mathbf{0} \end{array}$$

for which the dual becomes

$$\begin{array}{rrcl} \text{Maximize} & 8y_1 + 3y_2 & & \\ \text{subject to} & 5y_1 + 3y_2 & \leq & 13 \\ & y_1 + y_2 & \leq & 10 \\ & 3y_1 & \leq & 6 \\ & y_1, y_2 & & \text{free} \end{array} \qquad \begin{array}{l} \text{Maximize } \begin{bmatrix} y_1 & y_2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \\ \text{s.t. } \begin{bmatrix} y_1 & y_2 \end{bmatrix} \cdot \begin{bmatrix} 5 & 1 & 3 \\ 3 & 1 & 0 \end{bmatrix} \leq \begin{bmatrix} 13 & 10 & 6 \end{bmatrix} \end{array}$$

Another pair of primal and dual LPs

$$\begin{array}{rlcl} \text{Minimize} & 7x_1 - 11x_2 & & \\ \text{subject to} & x_1 + 2x_2 & \geq & 1 \\ & 2x_1 + x_2 & = & 5 \\ & 3x_1 + x_2 & \leq & -5 \\ & x_1 & \leq & 0 \end{array} \qquad \begin{array}{rlcl} \text{Maximize} & y_1 + 5y_2 - 5y_3 & & \\ \text{subject to} & y_1 + 2y_2 + 3y_3 & \geq & 7 \\ & 2y_1 + y_2 + y_3 & = & -11 \\ & y_1 & \geq & 0 \\ & y_3 & \leq & 0 \end{array}$$

along with a schematic view of the transformation:

	$x_1 \leq 0$	x_2 free	b_i and constraint
	A		
$y_1 \geq 0$	1	2	≥ 1
y_2 free	2	1	$= 5$
$y_3 \leq 0$	3	1	≤ -5
c_j and constraint	≥ 7	$= -11$	

We generalize the lower bounding argument from above to arbitrary LPs. Consider any solution $\mathbf{x}$ for a minimization LP and solution $\mathbf{y}$ for the corresponding maximization LP.

1. Consider the combination of constraint $i = 1, \ldots, m$ with $\mathbf{a}_i\mathbf{x}$ and b_i, and the corresponding variable y_i. The direction of the constraint inequality and (non-)negativity of y_i ensure that we always have

   $$y_i \cdot (\mathbf{a}_i\mathbf{x} - b_i) \geq 0.$$

   Summing over all these inequalities, we see that

   $$\sum_{i=1}^{m} y_i \cdot (\mathbf{a}_i\mathbf{x} - b_i) = \mathbf{y}^T\mathbf{A}\mathbf{x} - \mathbf{y}^T\mathbf{b} \geq 0$$

   so $\mathbf{y}^T\mathbf{b} \geq \mathbf{y}^T\mathbf{A}\mathbf{x}$ as in step 1. above.
2. For the combination of a variable x_j for $j = 1, \ldots, n$ and constraint with $\mathbf{y}^T\mathbf{a}^j$ and c_j, the direction of the constraint inequality and (non-)negativity of y_i ensure that we always have

   $$(c_j - \mathbf{y}^T\mathbf{a}^j) \cdot x_j \geq 0.$$

   Summing over all these inequalities, we see that

   $$\sum_{j=1}^{n} (c_j - \mathbf{y}^T\mathbf{a}^j)x_j = \mathbf{c}^T\mathbf{x} - \mathbf{y}^T\mathbf{A}\mathbf{x} \geq 0$$

   so $\mathbf{y}^T\mathbf{A}\mathbf{x} \leq \mathbf{c}^T\mathbf{x}$ as in step 2. above.

3. Overall, this implies $\mathbf{y}^T\mathbf{b} \leq \mathbf{y}^T\mathbf{A}\mathbf{x} \leq \mathbf{c}^T\mathbf{x}$ as above. Hence, the value of every solution of the primal [dual] minimization LP lower bounds the value of every solution of the dual [primal] maximization LP, resp.

Lemma 21 (Weak Duality)**.** *Consider a primal minimization LP. The objective value of every dual solution* $\mathbf{y}$ *is a lower bound on the objective value of every primal solution* $\mathbf{x}$*.*

Observations and consequences:

- The dual of the dual LP is the primal LP.
- If we transform the primal LP into an equivalent one (e.g. into standard form), then the dual of the transformed primal is equivalent to the dual of the original primal.
- If primal and dual solutions $\mathbf{x}$ and $\mathbf{y}$ have **the same objective function value** $\mathbf{y}^T\mathbf{b} = \mathbf{c}^T\mathbf{x}$, then Lemma 21 implies they must be **optimal solutions** for their respective LPs.
- Lemma 21 also implies:
  Primal unbounded ⇒ Dual infeasible.
  Dual unbounded ⇒ Primal infeasible.

The result in Lemma 21 can be strengthened to the main result in this section.

Theorem 45 (Strong Duality)**.** *If a primal LP is bounded and has an optimal solution* $\mathbf{x}^*$*, then the dual LP is bounded and has an optimal solution* $\mathbf{y}^*$*. The* optimal solutions have the same objective value

$$(\mathbf{y}^*)^T\mathbf{b} = \mathbf{c}^T\mathbf{x}^*.$$

Possible scenarios for primal and dual LPs:

		Dual		
		bounded	unbounded	infeasible
Primal	bounded	√		
	unbounded			√
	infeasible		√	√

For illustration an example, where both primal and dual LPs are infeasible:

$$\begin{array}{rrcl} \text{Minimize} & x_1 + 2x_2 & & \\ \text{subject to} & x_1 + x_2 & = & 1 \\ & 2x_1 + 2x_2 & = & 3 \end{array} \qquad \begin{array}{rrcl} \text{Maximize} & y_1 + 3y_2 & & \\ \text{subject to} & y_1 + 2y_2 & = & 1 \\ & y_1 + 2y_2 & = & 2 \end{array}$$

Complementary Slackness

A very useful property is **complementary slackness** of constraints and variables in primal and dual LPs. Intuitively, a (primal/dual) variable can only be non-zero if the corresponding (dual/primal) constraint holds with equality.

More formally, consider a pair of primal and dual programs. In the construction, we postulated that for primal and dual LPs we have

$$(c_j - \mathbf{y}^T\mathbf{a}^j) \cdot x_j \geq 0 \qquad \text{and} \qquad y_i \cdot (\mathbf{a}_i\mathbf{x} - b_i) \geq 0$$

for all $j = 1, \ldots, n$ and $i = 1, \ldots, m$, resp. Weak duality in Lemma 21 follows by observing the consequence

$$\mathbf{c}^T\mathbf{x} - \mathbf{y}^T\mathbf{b} = \sum_{j=1}^{n}(c_j - \mathbf{y}^T\mathbf{a}^j) \cdot x_j + \sum_{i=1}^{m} y_i \cdot (\mathbf{a}_i\mathbf{x} - b_i) \geq 0 \ .$$

By the inequalities above, the ≥ 0 condition holds for each of the $n + m$ summands individually. But for optimal $\mathbf{x}^*$ and $\mathbf{y}^*$, strong duality implies $\mathbf{c}^T\mathbf{x}^* - (\mathbf{y}^*)^T\mathbf{b} = 0$. This holds if and only if all the individual summands satisfy

$$(c_j - (\mathbf{y}^*)^T\mathbf{a}^j) \cdot x_j^* = 0 \qquad \text{and} \qquad y_i^* \cdot (\mathbf{a}_i\mathbf{x}^* - b_i) = 0$$

for all $j = 1, \ldots, n$ and $i = 1, \ldots, m$, resp.

This implies the **complementary slackness conditions** for optimal solutions $\mathbf{x}^*$ and $\mathbf{y}^*$:

Primal: For each j we have $x_j^* = 0$, or $(\mathbf{y}^*)^T\mathbf{a}^j = c_j$, or both. If the optimal primal solution has $x_j^* \neq 0$, then the corresponding dual constraint holds exactly. If the constraint does not hold exactly, we must have $x_j^* = 0$.

Dual: For each i we have $y_i^* = 0$, or $\mathbf{a}_i\mathbf{x}^* = b_i$, or both. If the optimal dual solution has $y_i^* \neq 0$, then the corresponding primal constraint holds exactly. If the constraint does not hold exactly, we must have $y_i^* = 0$.

Finally, for arbitrary primal and dual solutions, we define the **duality gap** as $\mathbf{c}^T\mathbf{x} - \mathbf{y}^T\mathbf{b}$.

6.2 Structure

Primal-dual algorithms are used to compute approximate solutions for problems that can be formulated as *integer* linear programs. The main ideas are as follows:

- Iteratively construct vectors $\mathbf{x}$ and $\mathbf{y}$ and decrease the duality gap
- Final vectors $\mathbf{x}$ and $\mathbf{y}$ are feasible primal and dual solutions
- Final vector $\mathbf{x}$ should be an *integer* vector
- If $\mathbf{c}^T\mathbf{x} \leq \alpha \cdot \mathbf{y}^T\mathbf{b}$, for some $\alpha > 0$, then $\mathbf{x}$ is an α-approximate solution for the (real, integer) problem, since by strong duality

$$\mathbf{c}^T\mathbf{x} \leq \alpha \cdot \mathbf{y}^T\mathbf{b} = \alpha \cdot \mathbf{c}^T\mathbf{x}_{LP}^* \leq \alpha \cdot \mathbf{c}^T\mathbf{x}_{ILP}^*$$

Further remarks:

- Many primal-dual algorithms maintain primal complementary slackness, i.e., they assign $x_j \neq 0$ only if the dual constraint $\mathbf{y}^T\mathbf{a}^j = c_j$. In contrast, dual complementary slackness is usually much less relevant.

- The LP (relaxation) is usually not solved to optimality. We only try to obtain pairs of *feasible* primal and dual solutions with small duality gap.
- In many problems, the dual variables y_i have a combinatorial interpretation. This can sometimes be used to obtain combinatorial algorithms without any use of LPs.
- There are primal-dual algorithms for many problems discussed above (e.g., for Set Cover, Vertex Cover, Facility Location, k-Median, Max-Matching, etc.)

We discuss a possible high-level schema for primal-dual algorithms.

- Primal: $\min \mathbf{c}^T\mathbf{x}$ s.t. $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$.
- Dual: $\max \mathbf{y}^T\mathbf{b}$ s.t. $\mathbf{y}^T\mathbf{A} \leq \mathbf{c}^T$ and $\mathbf{y} \geq \mathbf{0}$.
- W.l.o.g. we assume $\mathbf{A} \geq \mathbf{0}$, $\mathbf{b} \geq \mathbf{0}$, $\mathbf{c} \geq \mathbf{0}$

General structure:

1. Start with $\mathbf{x} = \mathbf{y} = \mathbf{0}$.
   $\mathbf{x}$ is not a feasible primal solution, but $\mathbf{y}$ is a feasible dual solution.
2. We want to maintain primal complementary slackness: $(c_j - \mathbf{y}^T\mathbf{a}^j) \cdot x_j = 0$.
3. Raise dual variables $\mathbf{y}$ until some constraint $\mathbf{y}^T\mathbf{a}^k \leq c_k$ gets satisfied exactly.
4. Raise x_k to satisfy (further) constraints of the primal
5. Repeat steps 3. and 4. until $\mathbf{x}$ is a feasible primal solution

6.3 Set Cover

6.3.1 Primal-Dual Algorithm

Recall the Weighted Set Cover problem from Section 5.3.3. We use a binary variable x_j to indicate if set S_j should be in the set cover or not. Then the LP relaxation (primal) and its dual LP are

$$\begin{array}{lrcll} \text{Min} & \sum_{j=1}^{n} w_j x_j & & & \\ \text{s.t.} & \sum_{j:e\in S_j} x_j & \geq & 1 & \text{for all } e \in E \\ & x_j & \geq & 0 & \text{for all } j \in [n] \end{array} \qquad \begin{array}{lrcll} \text{Max.} & \sum_{e\in E} y_e & & & \\ \text{s.t.} & \sum_{e\in S_j} y_e & \leq & w_j & \text{for all } j \in [n] \\ & y_i & \geq & 0 & \text{for all } e \in E \end{array} \tag{6.5}$$

Consider the primal-dual algorithm for Weighted Set Cover (Algorithm 26). Clearly, the algorithm can be implemented in polynomial time.

- We interpret the dual variable y_e as a "payment". Element e pays to "buy" sets that include e. Initially, no element pays anything, and the cover C is empty.
- Then we iteratively raise the payment of an uncovered element e until the payments suffice for the cost w_j of a set S_j that includes e. This set is bought and included into the cover C.
- Note, however, that e offers her payment y_e to *every* set that includes e. More formally, y_e is counted in every dual constraint for all sets S_j with $e \in S_j$.

Algorithm 26: Primal-Dual Algorithm for Weighted Set Cover

Input: Set system $\mathcal{S}$, set costs $w_j \geq 0$

Initialize $C \leftarrow \emptyset$, $\mathbf{x} \leftarrow \mathbf{0}$, and $\mathbf{y} \leftarrow \mathbf{0}$
while *there is* $e \in E$ *not covered by sets in* C **do**
    Increase y_e to the maximal value allowed by dual contraints:

$$y_e \leftarrow \min_{j:e\in S_j} \left(w_j - \sum_{\substack{e'\in S_j \\ e'\neq e}} y_{e'} \right)$$

    Let S_k be a set for which the dual constraint becomes tight by this increase
    `// Primal Compl. Slackness: Tight constraint allows` $x_k > 0$.
    Set $x_k \leftarrow 1$ and add $C \leftarrow C \cup \{k\}$
return C

- Later in the algorithm, more sets including e might be bought to cover other elements, but e needs to pay y_e for each of them, too – the other element only contributes what is needed to increase $\sum_{e'\in S_j} y_{e'}$ to w_j.
- A single payment y_e is a lower bound of what it costs to cover e. But how much do we overpay by the final cost of C in the algorithm?

[Pic: Some examples with output of the algorithm]

Recall from Section 5.3.3 the definition of $f = \max_{e\in E} |\{j \mid e \in S_j\}|$ the maximum number of sets that a single element is contained in. Similar to the deterministic rounding algorithm in Corollary 5, we obtain an approximation ratio of f.

Theorem 46. *Algorithm 26 has approximation ratio at most f.*

Proof. Consider the total payments of all elements.

- The total payments are exactly the total cost of the set cover, since elements pay exactly for each set in the cover

$$\sum_{j=1}^{n} x_j w_j = \sum_{j\in C} \sum_{e\in S_j} y_e \ .$$

- By the definition of f, the total payments are at most

$$\sum_{j\in C} \sum_{e\in S_j} y_e \leq f \cdot \sum_{e\in E} y_e \ ,$$

  since every single payment y_e can go to at most f sets.

Thus, with optimal primal and dual solutions we see

$$\sum_{j=1}^{n} x_j w_j = \sum_{j\in C} \sum_{e\in S_j} y_e \leq f \cdot \sum_{e\in E} y_e \leq f \cdot \sum_{e\in E} y_e^* = f \cdot \sum_{j} x_j^* w_j \ ,$$

i.e., the total cost of the computed cover C is at most f times the cost of an optimal (primal) solution $\mathbf{x}^*$ to the LP relaxation. □

The algorithm maintains *primal* complementary slackness conditions. And the *dual* ones?

- For the final solutions $\mathbf{x}$ and $\mathbf{y}$, we cannot have dual conditions as well – then the solutions would be optimal for the LPs, which is usually impossible when $\mathbf{x}$ is integral.
- However, instead of the exact ones

  $$y_e > 0 \quad \Rightarrow \quad \mathbf{a}_e \cdot \mathbf{x} = b_e$$

  we have *approximate versions* or the dual conditions

  $$y_e > 0 \quad \Rightarrow \quad \mathbf{a}_e \cdot \mathbf{x} \leq f \cdot b_e$$

  since $b_e = 1$ and $\mathbf{a}_e \cdot \mathbf{x} = \sum_{j:e \in S_j} x_j$ counts the number of sets in which element e is contained (at most f by definition).

The ratio of violation in dual complementary slackness and the approximation ratio coincide. This is no coincidence and a general condition!

Theorem 47. *Suppose we have primal and dual solutions* $\mathbf{x}$ *and* $\mathbf{y}$ *such that for the complementary slackness conditions, primal ones hold exactly:*

$$x_j > 0 \quad \Rightarrow \quad \mathbf{y}^T \mathbf{a}^j = c_j$$

and dual ones hold approximately with a ratio of β*:*

$$y_i > 0 \quad \Rightarrow \quad \mathbf{a}_i \cdot \mathbf{x} \leq \beta \cdot b_i$$

Then $\mathbf{x}$ *is a* β*-approximate solution for the primal LP.*

Proof. Exercise. □

6.3.2 Greedy Algorithm

Let us also consider a simple greedy algorithm for Set Cover (Algorithm 27). It can be analyzed using primal-dual arguments, although it does *not directly follow our standard schema* for primal-dual algorithms.

Notable facts for the algorithm:

- Clearly implementable in polynomial time.
- Obtains an **approximation ratio of at most** H_m (the m-harmonic number, at most $1 + \ln m$, even when f is large) and is **deterministic** (in contrast to randomized rounding in Algorithm 23)
- The ratio can be as high as H_m even in instances of Vertex Cover with $f = 2$.
- In general, Set Cover is NP-hard to approximate within a ratio of $(1 - \varepsilon) \ln m$, for every constant $\varepsilon > 0$. Hence, ratio is optimal unless P = NP.

Algorithm 27: Greedy Algorithm for Weighted Set Cover

Input: Set system $\mathcal{S}$, set costs $w_j \geq 0$

Initialize $C \leftarrow \emptyset$, $E_C \leftarrow \emptyset$
while $U \neq E$ **do**
  Determine set S_k that minimizes the cost per uncovered element $\frac{w_k}{|S_k \setminus E_C|}$
  Add $C \leftarrow C \cup \{k\}$ and update $E_C \leftarrow E_C \cup S_k$
return C

Theorem 48. *Algorithm 27 has approximation ratio at most $H_m \leq 1 + \ln m$.*

Proof. Reconsider primal and dual LPs (6.5) above.

- Suppose element $e \in E$ is covered first in iteration t by the inclusion of set S_j into C. Set the *payment* of e to
$$p_e = \frac{w_j}{|S_j \setminus E_C^t|} \ ,$$
where E_C^t denotes the set E_C in iteration t.
- Note that, overall, $\sum_{e \in E} p_e = \sum_{j \in C} w_j$.
- Set
$$y_e = p_e / H_m \ ,$$
where $H_m = \sum_{i=1}^{m} 1/i \leq 1 + \ln m$ is the m-th harmonic number.
- Consider any single set S_k. We denote $r = |S_k|$. Number the r elements $e \in S_k$ in non-decreasing order of the iteration when they are first covered.
- In the beginning of the iteration t where some element e_ℓ gets covered, we have at least $r - (\ell - 1)$ elements from S_k that remain uncovered.
- Suppose e_ℓ gets covered by the inclusion of some S_j into C. Then
$$p_{e_\ell} = \frac{w_j}{|S_j \setminus E_C^t|} \leq \frac{w_k}{|S_k \setminus E_C^t|} \leq \frac{w_k}{r - \ell + 1} \ ,$$
since the algorithm chooses S_j to minimize the cost per uncovered element.
- Thus, for the sum of all elements from S_k
$$\sum_{\ell=1}^{r} p_{e_\ell} \leq w_k \cdot \sum_{\ell=1}^{r} \frac{1}{r - \ell + 1} = w_k \cdot H_m \ .$$
- This implies
$$\sum_{e \in S_k} y_e = \sum_{e \in S_k} \frac{p_e}{H_m} \leq w_k \ ,$$
i.e., the dual constraint for S_k is fulfilled. $\mathbf{y}$ is a feasible dual solution!

Now interpret the set cover C as a primal solution $\mathbf{x}$ by setting $x_j = 1$ iff $j \in C$. Applying duality and using optimal primal and dual solutions we see

$$\sum_{j \in C} w_j = \sum_{j=1}^{n} x_j w_j = \sum_{e \in E} p_e = H_m \cdot \sum_{e \in E} y_e \leq H_m \cdot \sum_{e \in E} y_e^* = H_m \cdot \sum_{e \in E} x_j^* w_j \ ,$$

i.e., the total cost of the computed cover C is at most H_m times the cost of an optimal (primal) solution $\mathbf{x}^*$ to the LP relaxation. □

6.4 Shortest Path

We consider the standard single-source single-sink SHORTEST PATH problem:

- $G = (V, E)$ directed, connected
- $\ell_e \geq 0$ non-negative edge length for each $e \in E$
- Source node $s \in V$, target node $t \in V$
- **Goal**: Compute a directed s-t-path P that minimizes sum of edge lengths $\sum_{e \in P} \ell_e$

[Pic: Example]

What is a suitable ILP for SHORTEST PATH? We consider the notion of an **s-t-cut**: A cut is given by a subset $W \subset V$ of vertices such that $s \in W$ and $t \notin W$. It splits V into a pair of vertex sets $(W, V \setminus W)$, one containing s and the other t.

Consider an s-t-cut given by set W.

- Main observation: Every s-t-path P must "cross the cut" at least once.
- There is at least one edge $(u, v) \in P$ with $u \in W$ and $v \in V \setminus W$.
- We denote by $\delta(W) = \{(u, v) \mid u \in W, v \notin W\}$ the set of edges crossing the cut.
- We introduce a binary variable $x_e \in \{0, 1\}$ for each $e \in E$, where $x_e = 1$ means $e \in P$ (and $x_e = 0$ otherwise).
- Formally, our main observation is $\sum_{e \in \delta(W)} x_e \geq 1$.

[Pics: cut $(W, V \setminus W)$, set $\delta(W)$, path crossing the cut]

Let $\mathcal{S} = \{W \mid s \in W, t \notin W\}$ be the set containing all s-t-cuts. The ILP is

$$\begin{array}{lrcll} \text{Min} & \displaystyle\sum_{e \in E} x_e \ell_e & & & \\ \text{s.t.} & \displaystyle\sum_{e \in \delta(W)} x_e & \geq & 1 & \text{for all } W \in \mathcal{S} \\ & x_e & \in & \{0, 1\} & \text{for all } e \in E \end{array} \tag{6.6}$$

Lemma 22. *There is an optimal solution $\mathbf{x}^*$ of ILP (6.6) such that $E(\mathbf{x}^*) = \{e \in E \mid x^*_e = 1\}$ is a shortest s-t-path.*

Proof. Exercise. □

The linear relaxation uses $x_e \geq 0$ in (6.6). Here is the dual of the linear relaxation:

$$\begin{array}{lrcll} \text{Max} & \displaystyle\sum_{W \in \mathcal{S}} y_W & & & \\ \text{s.t.} & \displaystyle\sum_{W : e \in \delta(W)} y_W & \leq & \ell_e & \text{for all } e \in E \\ & y_W & \geq & 0 & \text{for all } W \in \mathcal{S} \end{array} \tag{6.7}$$

Algorithm 28: Primal-Dual Algorithm for Shortest Path

Input: Graph G, edge lengths ℓ_e, source s, target t

Initialize $F \leftarrow \emptyset$, $\mathbf{x} \leftarrow \mathbf{0}$, and $\mathbf{y} \leftarrow \mathbf{0}$
while *F does not contain an s-t-path* **do**
  $W \leftarrow \{v \in V \mid v \text{ reachable from } s \text{ via edges in } F\}$
  Increase y_W to the maximal value allowed by dual constraints:

$$y_W = \min_{e \in \delta(W)} \left(\ell_e - \sum_{\substack{W' : e \in \delta(W') \\ W' \neq W}} y_{W'} \right)$$

  Let e be an edge for which the dual constraint becomes tight by this increase
  `// Primal Compl. Slackness: Tight constraint allows` $x_e > 0$.
  Set $x_e \leftarrow 1$ and add $F \leftarrow F \cup \{e\}$
return s-t-path in F

The primal (dual) has a possibly exponential number of constraints (variables) corresponding to the number of s-t-cuts in G. But we never explicitly construct and solve these LPs!

Consider the primal-dual Algorithm 28 – it's almost literally the same as Algorithm 26 for Set Cover above! Some observations (c.f. the ones for Algorithm 26):

- We again interpret the dual variable y_W as a payment. The edge lengths are "transfer costs" to cross cuts along an edge e. Initially, we pay nothing and F is empty.
- Then we iteratively raise payments for some cut W that has not been crossed so far. If payments suffice to buy an edge $e \in \delta(W)$, this edge is bought and included into F. Then all cuts W' with $e \in \delta(W')$ are crossed by e.
- Cut W offers its payment y_W to *all* edges $e' \in \delta(W)$ simultaneously, i.e., y_W is counted in every dual constraint for all $e' \in \delta(W)$. In turn, when we buy edge e, we use all payments $y_{W'}$ of cuts W' with $e \in \delta(W')$ (since e crosses all these cuts).
- Later in the algorithm, when some other $e' \in \delta(W'')$ is bought to cross cut W'', then possibly $e' \in \delta(W'') \cap \delta(W)$ (since cut edges can overlap), and we would (over-)use payment y_W for yet another edge e'.
- For Set Cover, this overuse of payments yields the violation of dual complementary slackness and an approximation ratio of f. **We show this isn't a problem here!**

[Pic: Cuts, edges, payments, re-using payment of cut W for an edge $e' \in \delta(W'') \cap \delta(W)$]

Lemma 23. *The set F of edges is always a directed tree rooted in s.*

Proof. A simple induction. Basis: True initially since $F = \emptyset$. Hypothesis: F is directed tree rooted in s until iteration t. Step: In iteration $t+1$, we connect the node set W of tree F via some edge e to a node outside F. Hence, we extend F by e directed to a node outside F. This shows that $F \cup \{e\}$ is also a directed tree rooted in s. □

[Pic: Tree extension]

The main result of this section is the following.

Theorem 49. *Algorithm 28 computes an* optimal solution *to the single-source single-sink* Shortest Path *problem.*

Proof. Consider the final solution $\mathbf{x}$ and the final set F computed by the algorithm.

- For every $e \in F$ we have $x_e = 1$ and $\ell_e = \sum_{W:e\in\delta(W)} y_W$, i.e., primal complementary slackness conditions.
- The algorithm terminates when there is an s-t-path $P \subseteq F$. Let $\mathbf{x}^P$ be the primal solution corresponding to P ($x_e = 1$ if $e \in P$, and $x_e = 0$ else). Then

$$\begin{aligned}\sum_{e\in E} x_e^P \ell_e = \sum_{e\in P} \ell_e &= \sum_{e\in P} \sum_{W:e\in\delta(W)} y_W && \text{(primal compl. slackness)}\\ &= \sum_{W\in\mathcal{S}} \sum_{e\in P\cap\delta(W)} y_W && \text{(grouping together each occurence of } y_W\text{)}\\ &= \sum_{W\in\mathcal{S}} y_W \cdot |\delta(W)\cap P|\end{aligned}$$

- The last term can get large if P crosses some cut W *many times*, since then payment y_W is overused. We want to show: P **crosses** W **exaclty once if** W **has a positive payment** $y_W > 0$, or formally, $y_W > 0 \Rightarrow \sum_{e\in\delta(W)} x_e^P = 1$. This is **dual** complementary slackness!
- Consider a cut W that is crossed for the first time by path edge $e \in P$. When we increased $y_W > 0$, set W was the node set of tree F.
- Suppose we cross W by another edge $e' \in P$ later on. For this, there must be an edge e'' between e and e' to bring P back to a node in W. Upon addition of e'', set F stops being a directed tree rooted in s – a contradiction.

In summary, we thus have

$$\sum_{e\in E} x_e^P \ell_e = \sum_{W\in\mathcal{S}} y_W \cdot |\delta(W)\cap P| = \sum_{W\in\mathcal{S}} y_W$$

so primal solution $\mathbf{x}^P$ and dual solution $\mathbf{y}$ have the same objective function values. They must be optimal solutions for (I)LPs (6.6) and (6.7). □

[Pic: P returns to W to cross the cut several times]

Observe that Algorithm 28 is simply a primal-dual interpretation of the well-known greedy **Dijkstra algorithm**. Implementation in polynomial time is straightforward.

6.5 Facility Location

Reconsider the (metric) Facility Location problem from Section 4.4. There are sets F of locations and C of clients. For each pair $(i, j) \in C \times F$ there is a distance $d(i, j) \geq 0$, and for each $j \in F$ an opening cost $f_j \geq 0$. Distances form a metric. The goal is to open a subset

Algorithm 29: Primal-Dual Algorithm for Facility Location

Input: Sets of clients C and facilities F, distances d, opening costs f

Initialize $X \leftarrow \emptyset$, $t \leftarrow 0$, $A \leftarrow C$
Initialize primal solution $x_{ij} \leftarrow 0$, $y_j \leftarrow 0$, for all $i \in C, j \in F$
repeat
  Each active client $i \in A$ offers total payments of $\alpha_i = t$ to each $j \in F$:
    $\min(t, d(i,j))$ to connection cost $d(i,j)$, and
    if $y_j = 0$ closed, then $\beta_{ij} = \max(0, t - d(i,j))$ to opening cost f_j
  Increase time t until the next event happens:
  Event 1: $t = d(i,j)$ for active $i \in A$ and open $j \in X$
  Event 2: Subset of clients pay for f_j of some closed $j \notin X$
  if *Event 1 happens* **then**
    `// Compl. Slack: Connection paid allows` $x_{ij} > 0$
    set $x_{ij} = 1$, and remove $A \leftarrow A \setminus \{i\}$
  if *Event 2 happens* **then**
    $A_j = \{i \in A \mid t \geq d(i,j)\}$ `// active clients that contribute to` f_j
    `// Compl. Slack: Connection paid allows` $x_{ij} > 0$ `for each` $i \in A_j$
    `// Compl. Slack: Opening paid allows` $y_j > 0$
    if *there is* $j' \in X$ *with* $d(j,j') \leq 2t$ **then**
      leave j closed, set $x_{ij'} \leftarrow 1$ for all $i \in A_j$
    else
      open j, set $y_j \leftarrow 1$ and $X \leftarrow X \cup \{j\}$
      set $x_{ij} \leftarrow 1$ for all $i \in A_j$
    $A \leftarrow A \setminus A_j$
until $A = \emptyset$
return X

of locations and connect each client to an open location to minimize the total opening plus connection cost. A solution can be represented by a set $X \subseteq F$ of open facilities.

Consider an ILP formulation of the problem. We use binary variables $x_{ij} \in \{0,1\}$ to express if $i \in C$ is connected to $j \in F$, and $y_j \in \{0,1\}$ to express if $j \in F$ is open. We directly formulate the LP relaxation:

$$\begin{aligned} \text{Min} \quad & \sum_{j \in F} y_j f_j + \sum_{i \in C} \sum_{j \in F} x_{ij} d(i,j) & & \\ \text{s.t.} \quad & \sum_{j \in F} x_{ij} \geq 1 & & \text{for all } i \in C \\ & y_j - x_{ij} \geq 0 & & \text{for all } i \in C, j \in F \\ & x_{ij}, y_j \geq 0 & & \text{for all } i \in C, j \in F \end{aligned} \tag{6.8}$$

The first set of constraints requires that each client is connected to at least one facility. The

second set of constraints requires that whenever a client is connected to a facility, this facility must be open.

The dual of the LP relaxation is

$$\begin{array}{lrcll} \text{Max} & \sum_{j\in F} \alpha_i & & & \\ \text{s.t.} & \sum_{i\in C} \beta_{ij} & \leq & f_j & \text{for all } j \in F \\ & \alpha_i - \beta_{ij} & \leq & d(i,j) & \text{for all } i \in C, j \in F \\ & \alpha_i, \beta_{ij} & \geq & 0 & \text{for all } i \in C, j \in F \end{array} \tag{6.9}$$

Consider Algorithm 29, which has strong connections to primal and dual LPs:

- Start with all-zero primal and dual solutions. $\boldsymbol{\alpha}, \boldsymbol{\beta} = \mathbf{0}$ is dual feasible, and $\mathbf{x}, \mathbf{y} = \mathbf{0}$ fulfills primal complementary slackness (but is primal infeasible).
- We try to maintain both properties and also make $\mathbf{x}, \mathbf{y}$ primal feasible.
- Initially, all clients are active. We raise $\alpha_i = t$ for active clients at the same rate.
- First, a dual connection constraint will go tight when $\alpha_i = t = d(i,j)$. Then primal complementary slackness allows to raise $x_{ij} \geq 0$, i.e., i can get connected to $j \in F$. However, since we aim for primal feasibility, we do this only if j is *open*.
- If j was opened before (Event 1), then $y_j = 1$ and we connect i to j. We set $x_{ij} = 1$ and freeze the current dual variables α_i and β_{ij} (i.e., i stops being active).
- Otherwise, we continue to increase $\alpha_i = t > d(i,j)$. For dual feasibility, we must also raise $\beta_{ij} = \alpha_i - d(i,j) = t - d(i,j)$, the contribution of i to the opening cost of j.
- Then either some Event 1 happens for i (and a different facility), or a dual opening constraint $\sum_i \beta_{ij} = f_j$ goes tight (Event 2). Then primal complementary slackness allows to raise y_j, i.e., j is allowed to get opened.
- Now we can open j – but be careful! We include β_{ij}-contributions of inactive clients for the opening constraint of f_j (dual feasibility), but we don't want their payment, since we might have used it to justify opening earlier facilities...
- We check if there is some previously opened facility j' that is close ($d(j,j') \leq 2t$).
- If not, we open j, set $y_j = 1$ and connect all active clients in A_j to j by setting $x_{ij} = 1$ for each $i \in A_j$.
- Otherwise, we connect all clients from A_j to j' by setting $x_{ij'} = 1$ for each $i \in A_j$. We freeze their dual variables α_i and β_{ij} (i.e., they stop being active). This violates primal complementary slackness – but we maintain an *approximate* version, which will imply the approximation ratio.

[Pic: Algorithm, dual variable = payment of active clients, events, freezing of dual variables]

The main result of this section is the following theorem.

Theorem 50. *Algorithm 29 can be implemented in polynomial time and has an approximation ratio of 3.*

Proof. We first discuss the running time.

- Suppose an event has happened at time t.
- There are at most $|C| \cdot |F|$ many Events 1 and at most $|F|$ many Events 2.

- Given the current set of active clients, we can easily compute in polynomial time for each event E the value of $t_E \geq t$ when E would happen.
- As such, we can compute in polynomial time the next event that will happen.
- All computations when a given event happens can clearly be implemented efficiently.
- Since there are only polynomially many events in total, the algorithm requires polynomial running time.

We now turn attention to the approximation ratio.

Consider a client i at the end of the algorithm. We first show that i contributes at most one share $\beta_{ij} > 0$ to the opening cost of a facility j.

- Suppose for i there are two open facilities j_1 and j_2, and $\beta_{ij_1}, \beta_{ij_2} > 0$.
- Let j_2 be the facility that was opened later, and t be the time at which j_2 was opened.
- Since $\beta_{ij_1}, \beta_{ij_2} > 0$ we have $d(i, j_1) < t$ and $d(i, j_2) < t$. Hence, by the triangle inequality $d(j_1, j_2) \leq d(j_1, i) + d(i, j_2) < 2t$.
- t_2 is not opened by the algorithm – a contradiction. $\lightning$

Now consider the cost of the final solution $\mathbf{x}, \mathbf{y}$ and the "payments" made by all agents in the dual solution $\boldsymbol{\alpha}, \boldsymbol{\beta}$.

- Case 1: $i \in C$ is connected to a open facility j due to Event 1. Then $d(i, j) = \alpha_i$, i.e., i pays his connection cost. Since i is only connected to a single facility we have
$$\sum_{j' \in F} x_{ij'} d(i, j') = d(i, j) = \alpha_i.$$
- Case 2: $j \in F$ opened in Event 2. Then $y_j = 1$ and $x_{ij} = 1$ for all $i \in A_j$. By tightness of dual constraints, the active clients in A_j pay exactly for resulting opening and connection costs:
$$y_j f_j + \sum_{i \in A_j} \sum_{j' \in F} x_{ij'} d(i, j') = f_j + \sum_{i \in A_j} d(i, j) = \sum_{i \in A_j} \alpha_i \ ,$$
and by our observation above, agents in A_j do not pay for any other connection or opening costs.
- Case 3: $j \in F$ not opened in Event 2 at time t. There is open j' with $d(j, j') \leq 2t$, and we connect $x_{ij'} = 1$ for all $i \in A_j$. For every $i \in A_j$ we have $\alpha_i = t \geq d(i, j)$. By the triangle inequality
$$d(i, j') \leq d(i, j) + d(j, j') \leq t + 2t = 3\alpha_i$$
Hence, the total connection cost generated by "rerouting" the clients from A_j to j' is
$$\sum_{i \in A_j} \sum_{j \in F} x_{ij} d(i, j) = \sum_{i \in A_j} d(i, j') \leq \sum_{i \in A_j} 3\alpha_i \ .$$

Suppose $\mathbf{x}^*, \mathbf{y}^*$ and $\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*$ are optimal primal and dual solutions, resp. For the total cost of the solution computed by our algorithm we observe

$$\sum_{j \in F} y_j f_j + \sum_{i \in C} \sum_{j \in F} x_{ij} d(i, j)$$

$$\leq 3 \cdot \sum_{i \in C} \alpha_i \qquad \text{(apx. primal compl. slack., due to cases above)}$$

$$\leq 3 \cdot \sum_{i \in C} \alpha_i^* \qquad (\boldsymbol{\alpha}, \boldsymbol{\beta} \text{ dual feasible})$$

$$= 3 \cdot \left(\sum_{j \in F} y_j^* f_j + \sum_{i \in C} \sum_{j \in F} x_{ij}^* d(i,j) \right). \qquad \text{(strong LP duality)}$$

This proves an approximation ratio of at most 3. □

6.6 Steiner Forest

The Steiner Tree problem is a natural generalization of Min-Spanning Tree and single-source single-sink Shortest Path. We look for a min-cost set of edges that connects a *subset* $V_1 \subseteq V$ of the nodes (called terminals). The resulting tree can also contain other, so-called *Steiner* vertices – but these will be included only for reachability or cost minimization when connecting the set of terminals.

- $G = (V, E)$ *undirected* graph, edge costs $\ell_e \geq 0$ for each $e \in E$
- Partition of V into *terminals* $V_1 \subseteq V$ and *Steiner vertices* $V \setminus V_1$
- *Steiner tree* $T \subseteq E$: Set of edges such that $G = (V, T)$ has u-v-path for each $u, v \in V_1$
- **Goal:** Find a Steiner tree of smallest total cost $\sum_{e \in T} \ell_e$

[Pic: Example Steiner Tree]

We directly consider a generalization, the Steiner Forest problem. It is a classic example for more complicated network design problems, which have lots of applications in telecommunications, logistics, etc.

In Steiner Forest, we have k different *types* of terminals $V_1, \ldots, V_k$. We must select a set F of edges that, for each type i, connects all terminals of type i to each other:

- $G = (V, E)$ undirected graph, edge costs $\ell_e \geq 0$ for each $e \in E$.
- k vertex sets $V_1, \ldots, V_k \subseteq V$ of *terminals*
- *Steiner forest* $F \subseteq E$: Set of edges such that $G = (V, F)$ has u-v-path for each $u, v \in V_i$ and each $i = 1, \ldots, k$.
- **Goal:** Find a Steiner forest of smallest total cost $\sum_{e \in F} \ell_e$

Some remarks:

- F connects every pair of terminals $u, v \in V_i$ of the same type, but not necessarily pairs of terminals $u \in V_i$, $v \in V_j$ with different types $i \neq j$.
- Terminals of different types are connected only when this is needed for connecting some terminals of the same type or when it is cheaper in terms of costs.
- W.l.o.g. we can assume terminal sets are distinct: $V_i \cap V_j = \emptyset$ for $i \neq j$. (Why?)

[Pic: Example Steiner Forest, terminal types, distinct terminal sets]

Towards an ILP formulation, we extend the ILP (6.6) for Shortest Path. For each $W \subseteq V$, we define a function f that indicates if there must be an edge leaving set W

Algorithm 30: Primal-Dual Algorithm for Steiner Forest

Input: Graph $G = (V, E)$, edge costs ℓ_e, sets of terminals $V_1, \dots, V_k$

Initialize $F_1 \leftarrow \emptyset$, $\mathbf{x} \leftarrow 0$, $\mathbf{y} \leftarrow 0$, $t \leftarrow 1$
while F_t *not a feasible Steiner forest* **do**
  $\mathcal{C}_t \leftarrow \{W \mid W$ vertex set of a connected component in $(V, F_t)\}$
  $\mathcal{C}_t^1 \leftarrow \{W \in \mathcal{C}_t \mid f(W) = 1\}$
  Raise *all* y_W, $W \in \mathcal{C}_t^1$, *uniformly* until dual constraint tight for some $e' \in \delta(W), W \in \mathcal{C}_t^1$
  `// Primal Compl. Slackness: Tight constriant allows` $x_{e'} > 0$.
  $\Delta_t \leftarrow$ amount by which we raised each y_W, $W \in \mathcal{C}_t^1$
  $F_{t+1} \leftarrow F_t \cup \{e'\}$ and $x_{e'} \leftarrow 1$
  $t \leftarrow t + 1$
$F \leftarrow F_t$
while *there is* $e \in F$ *with* $F \setminus \{e\}$ *feasible* **do** $F \leftarrow F \setminus \{e\}$, $x_e \leftarrow 0$
return F

$$f(W) = \begin{cases} 1 & \text{there is } i \in [k] \text{ and } u, v \in V_i \text{ with } u \in W \text{ and } v \notin W \\ 0 & \text{otherwise.} \end{cases}$$

If $f(W) = 1$, we must connect at least one terminal inside W to at least one terminal outside W. Consider $\delta(W) = \{\{u, v\} \in E \mid u \in W, v \notin W\}$, the set of (undirected) edges in the cut $(W, V \setminus W)$. There must be at least one edge of $\delta(W)$ in every feasible Steiner forest F.

[Pic: Terminals, cut W, function $f(W) \in \{0, 1\}$]

We again use binary variables $x_e \in \{0, 1\}$ to indicate if $e \in F$ ($x_e = 1$) or not ($x_e = 0$). Let $\mathcal{S} = \{W \subset V \mid f(W) = 1\}$ be all *necessary cuts* of G, which every Steiner forest must cross.

Our resulting (I)LPs for Steiner Forest look identical to the ones for Shortest Path in (6.6) and (6.7) – the only difference lies in the definition of $\mathcal{S}$ based on function f. This, however, will make things more complicated.

$$\begin{array}{lll} \text{Min} & \sum_{e \in E} x_e \ell_e & \\ \text{s.t.} & \sum_{e \in \delta(W)} x_e \geq 1 & \text{for all } W \in \mathcal{S} \\ & x_e \in \{0, 1\} & \text{for all } e \in E \end{array} \qquad \begin{array}{lll} \text{Max} & \sum_{W \in \mathcal{S}} y_W & \\ \text{s.t.} & \sum_{W : e \in \delta(W)} y_W \leq \ell_e & \text{for all } e \in E \\ & y_W \geq 0 & \text{for all } W \in \mathcal{S} \end{array} \tag{6.10}$$

Consider Algorithm 30, which has similarities with Algorithm 28 above. Some remarks:

- Instead of one at a time, we raise *all* dual variables y_W by the same amount, for all W corresponding to connected components of (V, F_t) that need outside connections.
- In each iteration t, there are at most n connected components in (V, F_t). Hence, we need to raise at most n dual variables in this iteration.

- Also, in iteration t, at least one new dual constraint for some edge e' goes tight $\rightarrow$ at most $m = |E|$ iterations. Overall, at most nm non-zero dual variables in the end. Instead of all (exponentially many) y_W, we only keep track of the *non-zero* ones.
- In iteration t, we know the edges leaving the connected components of (V, F) and the dual variables that get raised. Hence, we can efficiently compute the smallest value Δ_t at which a dual constraint of a new edge becomes tight.
- The final pruning step in the while-loop can be implemented in polynomial time. It is necessary – otherwise F might contain way too many edges (Exercise).

[Pic: Example run of the algorithm, "moat growing" around terminals and components]

The observations above show the following result:

Lemma 24. *Algorithm 30 can be implemented in polynomial time.*

The main result of this section is the following theorem.

Theorem 51. *Algorithm 30 has an approximation ratio of 2.*

The algorithm maintains *exact primal* complementary slackness. The ratio 2 could result from *approximate dual* complementary slackness if for every $W \in \mathcal{S}$

$$y_W > 0 \quad \Rightarrow \quad \sum_{e \in \delta(W)} x_e \leq 2 \ .$$

Unfortunately, it's not so easy – we repeatedly raise *all* y_W of connected components, and it's unclear that the condition holds in the end *for every cut* W *with* $y_W > 0$. Instead, we show an "on average"-version for all cuts $W \in \mathcal{C}_t^1$, for which we raise y_W in an iteration t.

Lemma 25. *Given the final solution* F *(and* $\mathbf{x}$*) of the algorithm, we see that in the beginning of every iteration* t

$$\sum_{W \in \mathcal{C}_t^1} |\delta(W) \cap F| = \sum_{W \in \mathcal{C}_t^1} \sum_{e \in \delta(W)} x_e \leq 2 \cdot |\mathcal{C}_t^1|.$$

We first prove the theorem assuming the lemma is true.

Proof of Theorem 51. We start with the same arguments as in the proof of Theorem 49:

$$\begin{aligned}
\sum_{e \in E} x_e \ell_e = \sum_{e \in F} \ell_e &= \sum_{e \in F} \sum_{W : e \in \delta(W)} y_W && \text{(primal compl. slackness)} \\
&= \sum_{W \in \mathcal{S}} \sum_{e \in F \cap \delta(W)} y_W && \text{(grouping together each occurence of } y_W\text{)} \\
&= \sum_{W \in \mathcal{S}} y_W \cdot |\delta(W) \cap F|
\end{aligned}$$

(here approx. dual compl. slackness $|\delta(W) \cap F| \leq 2$ would be nice, but it's not true...)

In every iteration t, the dual variables y_W are raised by Δ_t each, for every $W \in \mathcal{C}_t^1$. In total, the increase in dual variables is

$$\sum_{W \in \mathcal{S}} y_W = \sum_t \sum_{W \in \mathcal{C}_t^1} \Delta_t = \sum_t \Delta_t \cdot |\mathcal{C}_t^1| \ .$$

Clearly, $\mathbf{y}$ remains dual feasible throughout the algorithm. Hence,

$$\begin{aligned}
\sum_{e \in E} x_e \ell_e &\leq \sum_{W \in \mathcal{S}} y_W \cdot |\delta(W) \cap F| = \sum_t \sum_{W \in \mathcal{C}_t^1} \Delta_t \cdot |\delta(W) \cap F| \\
&= \sum_t \Delta_t \cdot \sum_{W \in \mathcal{C}_t^1} |\delta(W) \cap F| \leq \sum_t \Delta_t \cdot \sum_{W \in \mathcal{C}_t^1} 2 \cdot |\mathcal{C}_t^1| && \text{(by Lemma 25)} \\
&= 2 \cdot \sum_t \sum_{W \in \mathcal{C}_t^1} \Delta_t \cdot |\mathcal{C}_t^1| = 2 \cdot \sum_{W \in \mathcal{S}} y_W && (\Delta_t \text{ raise for each } W \in \mathcal{C}_t^1) \\
&\leq 2 \cdot \sum_{W \in \mathcal{S}} y_W^* && (\mathbf{y} \text{ dual feasible}) \\
&= 2 \cdot \sum_{e \in E} x_e^* \ell_e && \text{(strong LP duality)}
\end{aligned}$$

□

Finally, let us argue why Lemma 25 is true.

Proof of Lemma 25. Consider the beginning of an iteration t. We consider the forest F and "contract" all edges from F_t (i.e., merge nodes to a super-node). This leaves us with H_t, a "forest of components" in iteration t:

- (V, F_t) is also a forest: Initially F_1 empty forest. Edge e' added only among different connected components $\rightarrow$ all F_t acyclic.
- Let $\mathcal{C}_t = \{W_1, \ldots, W_d\}$ be the (vertex sets of) connected components in iteration t.
- Construct the graph H_t: Add a vertex v_i for each set W_i. Add an edge $\{v_i, v_j\}$ for each $\{u, v\} \in F$ with $u \in W_i$ and $v \in W_j$.
- Since F is a forest, H_t must be a forest.

[Pic: Final forest F, forest F_t in iteration t, contracted forest H_t]

Each vertex v_i in H_t is *active* ($f(W_i) = 1$) or *non-active* ($f(W_i) = 0$).

- Every singleton node in H_t is an empty tree and has outdegree 0.
- Consider a vertex v_i of degree 1 in H_t. There is an edge $\{v_i, v_j\}$ in H_t. Why is $\{v_i, v_j\} \in F$? If $f(W_i) = 0$, then the edge would never be added (or deleted in the final pruning while-loop). Hence v_i must be active.
- A vertex v_i of degree 2 or more can be inactive – maybe $f(W_i) = 0$, but other components connect to each other by routing "through" v_i

Every *inactive* vertex has degree at least 2, so their average degree is *at least* 2. H_t is a forest with $1 \leq r \leq d$ trees, so average degree of all vertices is $2 - (2r/n) < 2$. Hence, the average

degree of *active* vertices in H_t is *at most* 2. Since active vertices correspond to components of $\mathcal{C}_t^1$ and their incident edges are from F, we see

$$\frac{1}{|\mathcal{C}_t^1|} \sum_{W \in \mathcal{C}_t^1} |\delta(W) \cap F| \leq 2 \ .$$

□

Algorithm 30 and Theorem 51 can be extended to min-cost forest problems where the connection requirements for subsets $W \subset V$ can be expressed using a function $f : 2^V \to \{0, 1\}$ with the following conditions:

1. $f(\emptyset) = f(V) = 0$
2. $f(W) = f(V \setminus W)$ for all $\emptyset \subseteq W \subseteq V$
3. $f(W \cup U) \leq \max\{f(W), f(U)\}$ for any two disjoint $W, U \subset V$

We call such a function f *proper*. For a proper f, there is an optimal solution of ILP 6.10 which corresponds to a forest $F \subseteq E$. Whenever there is a way to determine $f(W)$ for each component $W \in \mathcal{C}_t$ in polynomial time, we can implement each iteration t of Algorithm 30 in polynomial time. The proofs of Lemma 25 and Theorem 51 apply without modification for min-cost forest problems with a proper f.