# Contents

Beginning of 08/26/2024

# 1 Linear Programming

## 1.1 Introduction to LP

Consider real variables $x_1, \dots, x_n \in \mathbb{R}$ and an objective function $z$. Suppose we want to maximize the following written in **canonical form**[1]:

$$\max z = c^T x \quad \text{subject to} \quad Ax \leqslant b \qquad \text{(canonical form)}$$

where

$$C = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}.$$

One simple example of this could be

$$\max 2x_1 + x_2 \qquad \text{subject to} \quad \begin{cases} -x_1 + x_2 \leqslant 4 \\ x_1 + x_2 \leqslant 8 \\ 2x_1 - x_2 \leqslant 8 \\ x_1 \leqslant 5 \\ x_1, x_2 \geqslant 0. \end{cases} \qquad (*)$$

Of course, we can also express our optimization problem in a **noncanonical form**, and the transformation is natural:

$$\min z = c^T x \iff \max -z = -c^T x,$$

$$\sum_{j=1}^{n} a_{i,j} x_j \geqslant b_i \iff -\sum_{j-1}^{n} a_{i,j} x_j \leqslant -b_i,$$

$$\sum_{j=1}^{n} a_{i,j} x_j = b_i \iff \Big(\sum_{j=1}^{n} a_{i,j} x_j \leqslant b_i\Big) \text{ and } \Big(-\sum_{j-1}^{n} a_{i,j} x_j \leqslant -b_i\Big),$$

and the list goes on. We can also decompose any number $x \in \mathbb{R}$ into $x = x_i^+ - x_i^-$ with $x_i^+ := \max(x, 0)$ and $x_i^- = -\max(-x, 0)$ to get rid of the concerns of signs.

One of the earliest motivations of LP is as follows.

Consider a bipartite graph $G = (A \cup B, E)$ with $|A| = r|B| = s$, and $|A \cap B| = 0$. Let $A$ denote the warehouse and $B$ the retail stores. Each node $a_i$ has a supply $s_i$, $i \in [n]$[2], and each $b_i$ has a demand $d_j$, $j \in [s]$. For each $(i, j) \in [r] \times [s]$, an edge $c_{i,j}$ describes the cost of transporting unit good from warehouse $a_i$ to retail store $b_j$. Given that supplies and demands equal, $\sum_i s_i = \sum_j d_j$, the goal is to compute the cheapest way of transporting goods.

*Solution.* To convert this question into LP, let $x_{i,j}$ denote the amount of goods transported from warehouse $a_i$ to store $b_j$. Our goal naturally is to minimize the total cost $\sum_{i,j} c_{i,j} \cdot x_{i,j}$. Since the total supply equals the total demand, the constraint is that each warehouse is fully emptied and each store is at its loading capacity. Thus

[1]The reason why this is "canonical" will be explained later as we introduce duality.
[2]$i \in [n]$ abbreviates $1 \leqslant i \leqslant n$ (or $0 \leqslant i \leqslant n$, whichever is appropriate.)

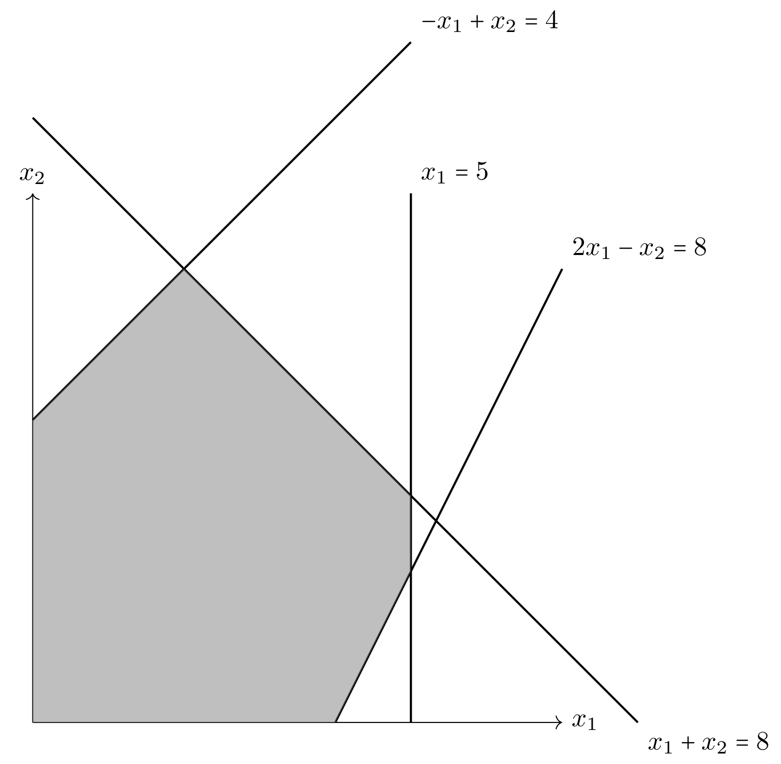
our question can be formulated as such:

$$\min_{(i,j)\in E} c_{i,j}\cdot x_{i,j} \qquad \text{subject to} \qquad \begin{cases} \sum_{j=1}^{s} x_{i,j} = s_i & i\in[r] \\ \sum_{i=1}^{r} x_{i,j} = d_j & j\in[s] \\ x_{i,j}\geqslant 0. \end{cases}$$

And of course, related questions include for example the maximization of profits: $\max_{i,j} P_{i,j}\cdot x_{i,j}$, and so on.

## The Geometry of LP

Let us revisit the example (*) shown above

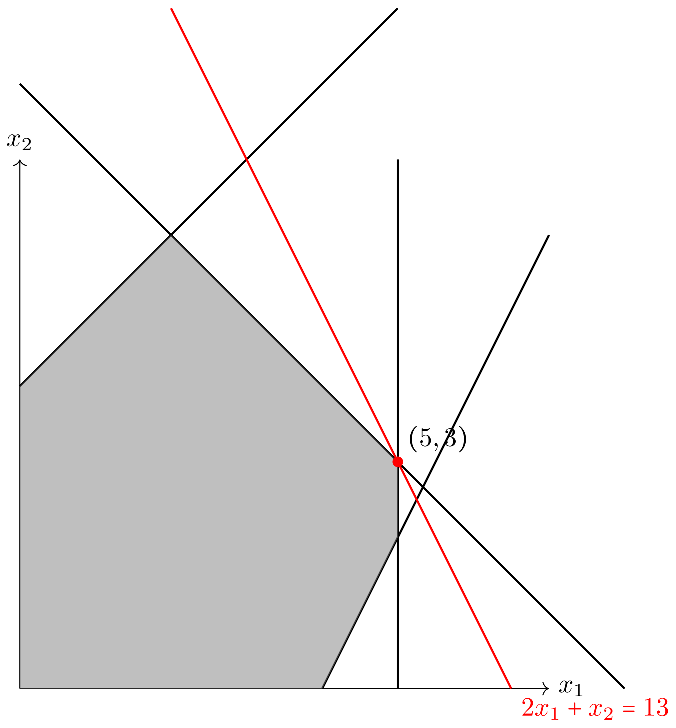


Note that the highlighted region, called the **feasible region** $F = \bigcap_i g_i$, is convex[3], where $g_i$s is obtained by taking the intersections of each constrained **halfspace**

$$g_i : \sum_{j=1}^{n} a_{i,j} b_j \leqslant b_i.$$

Finally, under the feasible region, we want to maximize $2x_1 + x_2$. To visualize this we consider a line orthogonal to the vector $(2,1)$ (which gives us any line of slope $-2$), since this line has form $2x_1 + x_2 = c$ for some $c$. The hyperplane (or in this case line) intersecting the region $F$ with highest $c$ will therefore give us the optimizer of $2x_1 + x_2$.

[3]A region $S$ is convex if for all $a,b\in S$, any linear combination $\lambda a + (1-\lambda)b, 0\leqslant\lambda\leqslant 1$, lies in $S$. Alternatively the line segment connecting $a$ and $b$ is contained in $S$.



In our case we obtained a unique solution, but note that a different objective function may yield infinite many solution, two solutions, or none. (In the linear case these are all possibilities.)
If the feasible set $F = \varnothing$ then there is no solution. On the other hand, if $F \neq \varnothing$ but $F$ is unbounded in the direction of the vector corresponding to our objective (in this case $(1,2)$), then our LP is unbounded. These are trivial cases that we want to disregard. Otherwise, if our LP is feasible and bounded, there exists an optimal solution, and

- there exists an optimal solution that is a vertex of $F$, and
- the set of optimal solutions is a convex set.

Beginning of 08/28/2024

## 1.2 LP Algorithms (Simplex & Others)

Recall from last time that our canonical form LP assumes the following form:

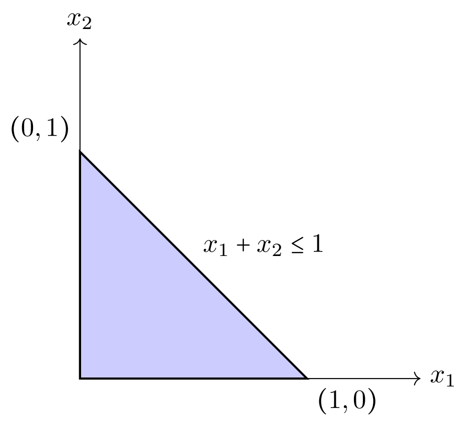
$$\max c^T x \qquad \text{subject to} \qquad \begin{cases} Ax \leqslant b \\ x \geqslant 0, \end{cases}$$

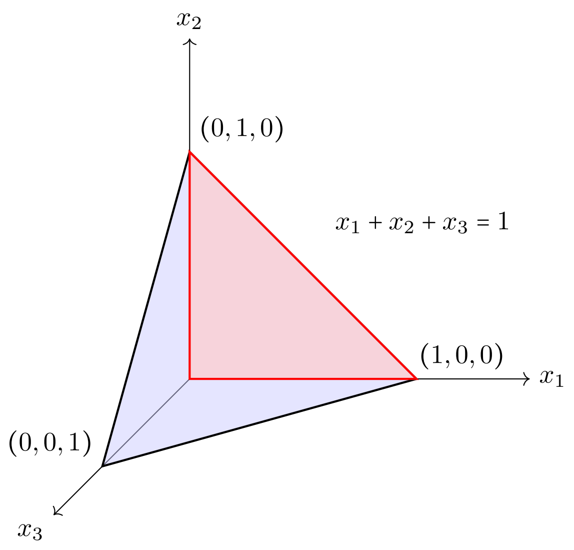
and we assume the problem is feasible (with feasible region $F$) and bounded (so it admits an optimal solution).
We convert the feasible region $F$ into a directed graph $G = (V, E)$ (with matching edges and vertices), where there exists an edge $u \to e$ if $c^T u < c^T e$.
To retrieve the maximum, a simple greedy algorithm would be as follows: for an arbitrary vertex $v$ in $V$, we can follow a path in $G$ until it reaches a sink, and we return the sink. Unfortunately a naïve approach would easily lead to an exponential runtime, so we need to work "smarter." This leads to the **simplex method**.

To use the simplex method we will use the **standard form LP**. We will introduce additional variables, called the **slack variables**, and convert each constraint as follows:

$$\sum_{j=1}^{n} a_{i,j}x_j \leqslant b_j \qquad \longrightarrow \qquad \sum_{j=1}^{n} a_{i,j}x_j + x_{n+i} = b_i, \qquad \text{where } x_{n+i} = 0.$$

For example, we can rewrite $x_1 + x_2 \leqslant 1$ as $x_1 + x_2 + x_3 = 1$, where $x_3 \geqslant 0$ is a slack variable. The former is a triangle in the first quadrant on $\mathbb{R}^2$, and the latter is the plane in $\mathbb{R}^3$ going through $(0.5, 0.5, 0)$ normal to $(1, 1, 1)$. Geometrically the slack variable introduces one additional region, and the feasible region can be recovered by taking the intersection of the resulting polyhedron and the plane with new dimension = 0.





An example of converting constraints to standard form:

$$\max x_1 + x_2 \qquad \text{subject to} \qquad \begin{cases} x_1 \leqslant 3 \\ x_2 \leqslant 2 \\ -x_1 + x_2 \leqslant 1 \\ x_1, x_2 \geqslant 0 \end{cases} \longrightarrow \begin{cases} x_1 + x_3 = 0 \\ x_2 + x_4 = 2 \\ -x_1 + x_2 + x_5 = 1 \\ x_1, \cdots, x_5 \geqslant 0. \end{cases}$$

Essentially the constraints, after introducing $n$ slack variables, can be written as $Ax = b, x \geqslant 0$ where $A$ is now $m \times (m + n)$. With an abuse of notation we denote the number of equations after introducing slack variables by just $n$, so $A$ is $m \times n$ This matrix has rank at most $m$, so the solution has at most $m$ nonzero entries.

Let basis $\mathcal{B} \subset \{1, \ldots, n\}$ be a subset of size $m$, and let $x_\mathcal{B}$ be the set of variables (called **basic variables**) corresponding to $\mathcal{B}$, i.e., $x_\mathcal{B} = \{x_i : i \in \mathcal{B}\}$, and likewise define $A_\mathcal{B}$ to be the $m \times m$ matrix where each column ($m$ total now) corresponds to a basis.

Let $\mathcal{N} := \{1, \ldots, n\} \backslash \mathcal{B}$ and define $x_\mathcal{N}, A_\mathcal{N}$ likewise. ($x_\mathcal{N}$ is called the set of **non-basic variables**.) Also define $c_\mathcal{B}$ and $c_\mathcal{N}$ analogously.

Then $Ax = b$ becomes

$$A_\mathcal{B}x_\mathcal{B} + A_\mathcal{N}x_\mathcal{N} = b,$$

where we set $x_j = 0$ for each $j \in \mathcal{N}$, so in essense we have $\mathcal{A}_\mathcal{B}x_\mathcal{B} = b$, or $x_\mathcal{B} = (A_\mathcal{B})^{-1}b$, since $A_\mathcal{B}$ is assumed to be invertible. This solution is called the **basic solution**. If also $x_\mathcal{B} \geqslant 0$ then it is further called a **basic feasible solution**.

**Proposition**

(1) For a fixed $\mathcal{B}$, the basic solution is unique, and

(2) $x_{\mathcal{B}}$ is feasible (i.e. $x_{\mathcal{B}} \geqslant 0$) if and only if $(x_{\mathcal{B}}, 0)$ (the $m+1$-dimensional point) is a vertex of $F$.

*The simplex method, as we will see, jumps from one vertex to another.*

The simplex method will "jump" between adjacent matrices whenever the objective $c_{\mathcal{B}}^T x_{\mathcal{B}}$ can increase:

**Algorithm 1:** The Simplex Algorithm

```
1 initialization: $\mathcal{B} \leftarrow$ initial basis of $\{1, \dots, n\}$ with $|\mathcal{B}| = m$.
2 while $\mathcal{B}$ is not optimal do
3     find a pair $(i, j)$ such that $i \in \mathcal{B}, j \notin \mathcal{B}$, with $c_{\mathcal{B}}^T x_{\mathcal{B}} < c_{\mathcal{B}'}^T x_{\mathcal{B}'}$
4         where $\mathcal{B}' = \mathcal{B} \cup \{j\} \backslash \{i\}$ (swap $i$ for $j$)
5     $\mathcal{B} \leftarrow \mathcal{B}'$
6 return $\mathcal{B}$
```

Rule of thumbs when swapping in/out indices for $\mathcal{B}$:

- Write every indexed equation in $\mathcal{B}$ in terms of variables in $x_{\mathcal{N}}$ (e.g. if $\mathcal{B} = \{3,4,5\}$, express $x_3, x_4, x_5$ in terms of $x_1, x_2$, and constants). Write basic variables in terms of non-basic variables. This way when all non-basic variables are zero, we get one objective value.
- Start by setting all slack variables as basic variables and original variables as non-basic.
- Check the objective function. Bring in a non-basic variable (currently in $\mathcal{N}$) with positive coefficients (so it contributes to the objective).
- Check the existing constraints, and see check the maximum value by which the new variable can increase without causing adverse effect. Remove the bottleneck constraint.

Let us now take a look at simplex in action on the following problem. Our first step is to convert the problem into standard form.

$$\max x_1 + 2x_2 \qquad \text{subject to} \qquad \begin{cases} x_1 + x_2 \leqslant 8 \\ x_2 - x_1 \leqslant 4 \\ x_1 \leqslant 5 \\ 2x_1 - x_2 \leqslant 8 \\ x_1, x_2 \geqslant 0 \end{cases} \implies \begin{cases} x_1 + x_2 + x_3 = 8 \\ -x_1 + x_2 + x_4 = 4 \\ x_1 + x_5 = 5 \\ 2x_1 - 2x + x_6 = 8 \\ x_1, \cdots, x_6 \geqslant 0. \end{cases}$$

Our original unknowns are $\{1,2\}$, so we set them to be non-basic, i.e. $\mathcal{N} = \{1,2\}$ and $\mathcal{B} = \{3,4,5,6\}$ (in this case $m = 4, n = 4+2$) since we introduce one slack variable for each inequality. And we ask, what would be a reasonable choice in $\mathcal{N}$ to swap into $\mathcal{B}$, and what to evict form $\mathcal{B}$? Well, to maximize $z = 0 + x_1 + 2x_2$, we want to increase both $x_1$ and $x_2$ since their coefficients are positive[4]. If we want to swap index 2 into $\mathcal{B}$, what should we evict? This

[4]Normally we would first increase $x_2$ because its coefficient is larger. But here I'm following the notes and start by swapping $x_1$ into non-basic.

is equivalent to asking the question, *what is the bottleneck that prevents us from increasing $x_1$ too much?* The first equation suggests $x_1$ can be increased by at most 8 before the constraint $x_3 \geqslant 0$ risks being broken. The second equation doesn't care since if $x_4 = 4+x_1-x_2$ and we increase $x_1$, $x_4$ always remains nonnegative. The third equation requires that the increment be bounded from above by 5, and the last equation requires the increment to be $\leqslant 4$. So the last equation is the bottleneck, meaning we want to swap out $x_6$. So the new $\mathcal{B}$ discards 6 and gains 1, namely $\mathcal{B} \leftarrow \{1,3,4,5\}$ and $\mathcal{N} \leftarrow \{2,6\}$. Next up we rewrite $x_1, x_3, x_4, x_5$ in terms of $x_2$ and $x_6$.

$$\begin{array}{rl r} x_3 & = 8 - x_1 - x_2 & \Delta x_{\max} = 8 \\ x_4 & = 4 + x_1 - x_2 & \Delta x_{\max} = \infty \\ x_5 & = 5 - x_1 & \Delta x_{\max} = 5 \\ x_6 & = 8 - 2x_1 + x_2 & \boxed{\Delta x_{\max} = 4} \\ \hline z & = 0 + \boxed{x_1} + 2x_2 & \end{array} \quad \overset{\text{(1 in, 6 out)}}{\Longrightarrow} \quad \begin{array}{rl} x_1 & = 4 + x_2/2 - x_6/2 \\ x_3 & = 4 - 3x_2/2 + x_6/2 \\ x_4 & = 8 - x_2/2 - x_6/2 \\ x_5 & = 1 - x_2/2 + x_6/2 \\ \hline z & = 4 + 5x_2/2 - x_6/2. \end{array}$$

Now increasing $x_6$ would cause $z$ to decrease, so that is bad. We only want to increase $x_2$ now, so 2 in. What out? The $\Delta x_{\max}$ are $\infty, 4/(3/2), 8/(1/2)$, and $1/(1/2)$, respectively, so we would swap the last equation out, i.e., 5 out. We repeat this process until all coefficients in the bottom equation of $z$ become negative, at which point we realize we can no longer jump to any neighboring vertex to optimize the value. So we are done. Below is a summary:

$$\overset{\text{(2 in, 5 out)}}{\Longrightarrow} \begin{array}{rl} x_1 & = 5 - x_5 \\ x_2 & = 2 - 2x_5 + x_6 \\ x_3 & = 1 + 3x_5 - x_6 \\ x_4 & = 7 + x_5 - x_6 \\ \hline z & = 9 - 5x_5 + 2x_6 \end{array} \quad \overset{\text{(6 in, 3 out)}}{\Longrightarrow} \begin{array}{rl} x_1 & = 5 - x_5 \\ x_2 & = 3 + x_5 - x_3 \\ x_4 & = 6 - 2x_5 + x_3 \\ x_6 & = 1 + 3x_5 - x_3 \\ \hline z & = 11 + x_5 - 2x_3 \end{array} \quad \overset{\text{(5 in, 4 out)}}{\Longrightarrow} \begin{array}{rl} x_1 & = 2 - x_3/2 + x_4/2 \\ x_2 & = 6 - x_3/2 - x_4/2 \\ x_5 & = 3 - x_3/2 - x_4/2 \\ x_6 & = 6 + x_3/2 - 3x_4/2 \\ \hline z & = 14 - 3x_3/2 - x_4/2. \end{array}$$

By this point we see that the maximum possible value of $z$ is $\leqslant 14$. And this value is attained because we can simply set $x_3, x_4 = 0$ and easily verify that the non-negativity constrainst of every other variable are simultaneously satisfied. So we conclude that $\max z = \max x_1 + 2x_2 = 14$.

**Simplex Runtime**

Our simplex runs in weakly polynomial time:

- The input size $\langle x \rangle$ is the number of bits to represent $x$, which is $\log x$. The input size would be $L = \sum \langle c_i \rangle + \sum \langle a_{i,j} \rangle + \sum \langle b_i \rangle$.
- The algorithm runs $L^{\mathcal{O}(1)}$ numbers of arithmetic operations. Since this runtime is polynomial with respect to the input length, the algorithm is weakly polynomial. (A strongly polynomial algorithm would have its runtime independent from $L$, only $m$ and $n$.)

## Some Other LP Algorithms

We will also introduce a few other known LP algorithms below:

**Ellipsoid algorithm** (works for convex optimization):

- The idea behind this is that we generate a sequence of "decreasing" ellipsoids that are guaranteed to contain an extrema. The main idea is that in each iteration, we use a hyperplane to discard a side that is infeasible, therefore allowing us to "zoom in" on the other side.

- We begin with an ellipsoid $E_0$ centered at $x^{(0)}$ that contains the feasible region $F$. In each iteration we compute a subgradient $g^{(k)}$ of $x^{(k)}$, the center of $E_k$, and note that

$$E_k \cap \{z : \langle g^{(k)}, (z - x^{(k)}) \rangle \leqslant 0\}$$

  is the half ellipsoid that contains a minimizer of $f$. This means we can "discard" the other "bad" half, and compute a smaller ellipsoid that contains this "good" half. It can be shown that

$$\frac{\text{Vol}(E_{i+1})}{\text{Vol}(E_i)} \leqslant \exp\left(-\frac{1}{2(n+1)}\right),$$

  which is a decent convergence measure.

**Interior point method**:

- The high-level idea of the interior point method is that we start inside $F$ and walk our path towards an optimal solution. In the process we try to avoid reaching the boundary $\partial F$ of $F$. This is done by introducing barrier functions in addition to the objective:

$$f_\mu(x) = c^T x + \underbrace{B(x)}_{\text{barrier function}}$$

  and a common choice is the "log barrier,"

$$f_\mu(x) = c^T x + \mu \sum_{i=1}^{m} \log(b_i - a_i x).$$

  When our value is close to the boundary, one of the term blows up to $-\infty$. The closer we are to our barrier, the more negative $B(x)$ becomes, and hence more punishment.

- When $\mu = 0$ we recover the standard LP. The other extreme, when $\mu = \infty$, we recover the analytic center of $F$.

- We follow the path $\{x^*_\mu \mid \mu > 0\}$. Starting from a large value of $\mu$ we slowly decrease $\mu$ by $\mu_{i+1} \leftarrow (1 - m^{-1/2})\mu_i$, and stop when $\mu \sim 2^{-L}$. This takes $\mathcal{O}(\sqrt{m}L)$ steps. If we use Newton's method at each step to compute $x_\mu$ (which takes cubic time), the total complexity is around $\mathcal{O}(m^{3.5}L)$.

Beginning of 09/04/2024

## 1.3 Duality

Consider (again) the following LP in canonical form

$$z^* = \max z = \max 2x_1 + 3x_2 \qquad \text{subject to} \qquad \begin{cases} 4x_1 + 8x_2 \leqslant 12 & (1) \\ 2x_1 + x_2 \leqslant 3 & (2) \\ 3x_1 + 2x_2 \leqslant 4 & (3) \\ x_1, x_2 \geqslant 0. \end{cases} \qquad (*)$$

Can we infer something about the optimal solution by directly manipulating the constraints? Well, we know one thing for sure: $2x_1 + 3x_2 \leqslant 4x_1 + 8x_2$, so the first constraint, without doing any arithmetic, tells us $z^* = 12$, and dividing (1) by 2 tells us $z^* \leqslant 6$. Similarly, $[(1) + (2)]/3$ tells us $2x_1 + 3x_2 \leqslant 5$, so $z^* \leqslant 5$, and $5/4 \cdot (1) + (3)$ tells us $8x_1 + 12x_2 \leqslant 19$, so dividing by 4 gives $z^* \leqslant 19/4$.

More generally, for any $y_1, y_2, y_3$,

$$\begin{aligned} y_1 \cdot (1) + y_2 \cdot (2) + y_3 \cdot (3) &= y_1(4x_1 + 8x_2) + y_2(2x_1 + x_2) + y_3(3x_1 + 2x_2) \\ &= (4y_1 + 2y_2 + 3y_3)x_1 + (8y_1 + y_2 + 2y_3)x_2 \\ &\leqslant 12y_1 + 3y_2 + 4y_3. \end{aligned}$$

Therefore, to maximize $2x_1 + 3x_2$, we are essentially transform the original LP into another with respect to the $y_i$'s:

$$\min 12y_1 + 3y_2 + 4y_3 \qquad \text{subject to} \qquad \begin{cases} 4y_1 + 2y_2 + 3y_3 \geqslant 2 & (1') \\ 8y_1 + y_2 + 2y_3 \geqslant 3 & (2') \\ y_1, y_2, y_3 \geqslant 0. \end{cases} \tag{**}$$

We call (*) the **primal LP** and (**) the **dual LP**. The duality lies in the sense that the constraints become the coefficients, and vice versa. More concisely, in linear algebraic notations,

$$(\text{Primal}) \max\{c^T x \mid Ax \leqslant b, x \geqslant 0\} \iff (\text{Dual}) : \min\{b^T y \mid A^T y \geqslant c, y \geqslant 0\}.$$

Why duality? In a nutshell the power of duality can be summarized in the weak and strong duality theorems:

**Theorem: Weak duality theorem**

If $x$ is a primal (maximization) feasible solution and $y$ is a dual (minimization) feasible solution, then $c^T x \leqslant b^T y$. Therefore, $\max c^T x \leqslant \min b^T y$.

*Proof.* Direct by assumptions.

$$c^T x \leqslant (A^T y)^T x = y^T Ax \leqslant y^T b = b^T y.$$ □

**Theorem: Strong duality theorem**

There are four cases for a primal-dual LP. The first three are trivial, and the last one is interesting:

(1) Both primal and dual are infeasible.

(2) Primal is infeasible and dual is unbounded.

(3) Dual is infeasible and primal is unbounded.

(4) <u>Both primal and dual are bounded and feasible, and in this case, the optimal values equal: $\max c^T x = \min b^T y$.</u>

*Proof sketch.* (2) If $D$ is unbounded, the value goes all the way to $-\infty$. But if $c^T x \leqslant b^T y$ and $b^T y$ is unbounded, then the only explanation is that the primal is infeasible.

(4) Technical details omitted. If a primal LP has a feasible solution and satisfies certain conditions (like convexity and closedness), then the corresponding dual LP also has a feasible solution such that the objective values match; here one invokes the separating hyperplanes theorem. □

How do we use this theorem for an optimization problem? We want to find $x, y$ such that (i) constraints are satisfied, and (ii) the **duality gap** is closed, i.e., finding $x, y$ such that

$$\text{(feasibility)} \begin{cases} Ax \leqslant b \\ A^T y \geqslant c \\ x, y \geqslant 0 \end{cases} \qquad \text{and} \qquad \text{(optimality)}\ b^T y \leqslant c^T x.$$

This gives us the optimal solution because the weak duality theorem already guarantees that $c^T x \leqslant b^T y$. Two inequalities combined and we obtain equality.

Below is a general recipe that one may follow when converting primal to dual (and note that since this is symmetric, the dual of dual is the primal itself.)

$$\text{(Primal)} \max\{c^T x \mid Ax \leqslant b, x \geqslant 0\} \iff \text{(Dual)} : \min\{b^T y \mid A^T y \geqslant c, y \geqslant 0\}.$$

| | primal | dual |
|---|---|---|
| variables | $x_1 \cdots x_n$, $n$ total | $y_1 \cdots y_m$, $m$ total |
| constraints | $m$ total | $n$ total |
| matrix | $A$ | $A^T$ |
| objective function | $\max c^T x$ | $\min b^T y$ |
| constraints | $i^{\text{th}}$ constraint $\leqslant \mid \geqslant \mid =$ | $y_i \geqslant 0 \mid y_i \leqslant 0 \mid y_i \in \mathbb{R}$ |
| variables | $x_j \geqslant 0 \mid x_j \leqslant 0 \mid x_j \in \mathbb{R}$ | $j^{\text{th}}$ constraint $\geqslant \mid \leqslant \mid =$ |

### 1.3.1 The Optimal Transport Problem

Let's now consider again the optimal transport problem. We use the same setup: given a graph $G = (A \cup B, E)$, we have $\sum s_i = \sum d_j$ (total supply and demand equal), and we let $x_{i,j}$ model the amount of goods transported from $a_i$ (supplier $i$) to $b_j$ (consumer $j$), with unit cost $c_{i,j}$. Our objective:

$$\min_{(a_i,b_j)\in E} c_{i,j} x_{i,j} \qquad \text{subject to} \qquad \begin{cases} \sum_{j=1}^n x_{i,j} = s_i & i \in [n] \\ \sum_{i=1}^m x_{i,j} = d_j & i \in [m] \\ x_{i,j} \geqslant 0. \end{cases}$$

What would our dual look like?

- The dual will be a maximization problem.
- For every primal constraint $s_i$, we will have a dual variable $u_i$. Similarly a dual variable $v_j$ for each primal constraint $d_j$.
  - Since the primal constraints are =, these variables just need to be real values, i.e., $u_i, v_j \in \mathbb{R}$.
- What about dual constraints? These correspond to primal variables, which are edges.
  - The coefficients for $x_{i,j}$ in the primal LP are all 1's. So will the coefficients for the $u_i, v_j$'s in the dual.

Summarizing everything, the dual LP can be formulated as

$$\max\left[\sum_{i=1}^{m} s_i u_i + \sum_{j=1}^{n} d_j v_j\right] \qquad \text{subject to} \qquad \begin{cases} u_i + v_j \leqslant c_{i,j} & (i,j) \in [n] \times [m] \\ u_i, v_j \in \mathbb{R}. \end{cases}$$

How do we interpret this result? *Shadow prices*. Think of this as the answer to "how much is the value perceived by others?" Suppose we are approached by a shipper[5], saying they are willing to transport goods from $a_i$ to $b_j$, only charging us, per unit price, $u_i$ for loading and $v_j$ for unloading. For us, it seems reasonable that we only accept this deal if $u_i + v_j \leqslant c_{i,j}$, i.e., it's actually cheaper to hire this shipper than shipping the goods ourselves. On the other hand, the shipper also wants to maximize their profit, so the objective is $\max(s_i u_i + d_j v_j)$.

More generally, the primal-dual LP can be interpreted as *resource allocation* versus *price valuation*.

(1) (Resource allocation) You are a factory owner, and you want to maximize profit of producing $n$ types of goods. Your goal? $\max c^T x$, where $c$ models the profit vector and $x$ the quantity vector. "How many units of each item should I make to maximize overall profit, subject to the following constraints on raw meterials?"

- $b$ is the "total resources" vector: $b_j$ is the total available amount of raw material $j$.
- $A$ is the "ingredient matrix:" $a_{i,j}$ measures the amount of $i^{\text{th}}$ resources needed to produce one unit of item $j$.

(2) (Price valuation) Goal: you are a buyer and you directly go to factories to buy raw materials. The quantity you need is measured by the vector $b$ (i.e. $b_j$ units for item $j$), and you offer unit price $y_j$. Your goal? Minimize your total cost, i.e., $\min b^T y$.

- Your main challenge, of course, is to actually convince the factory owner to directly sell you the raw materials. This is analogous to thinking "hiring a shipper saves money" above. The factory owner earns a unit profit of $c_j$ by making item $j$ themselves, so what you offered for the same amount of ingredients to produce one unit of item $j$ must not be below that. Recall $a_{i,j}$ is the amount of $i^{\text{th}}$ raw material needed to produce one unit of item $j$. Therefore, $[a_{1,j}, a_{2,j}, \cdots, a_{m,j}][y_1, y_2, \cdots, y_m] \geqslant c_j$. And more generally, $A^T y \geqslant c$.

### 1.3.2 Two-Player Zero-Sum Game

Consider a game with two players. Player A can make $m$ moves, and B can make $n$ moves. Let $M = [m_{i,j}]$ be an $m \times n$ matrix such that $m_{i,j}$ describes the amount B pays to A if A makes move $i$, and B makes move $j$. An optimization naturally arises: A wants to maximize the money they earn from B, and B wants to minimize the money they pay to A. Question: what strategy should they follow?

We let $x \in [0,1]^m$ denote the probability distribution of A's next move, and let $y \in [0,1]^n$ model B's next move. What is the expected payoff? This quantity can be modeled by

$$\sum_{i,j} m_{i,j} \cdot \mathbb{P}(\text{A makes move } i) \cdot \mathbb{P}(\text{B makes move } j) = \sum_{i,j} x_i m_{i,j} y_j = x^T M y.$$

The **worst-case optimal strategy** from A's perspective is

$$\max_x \beta(x) = \max_x \min_y x^T M y. \tag{*}$$

[5]Which is why sometimes this dual is called a Shipper's problem.

Why? Assuming they play rationally, B always tries to minimize $x^T My$ assuming they know A's next move probability distribution, so $\beta(x)$ models A's gain from a certain strategy $x$ assuming an optimized adversary. Taking maximum over all $x$, this gives A the "best worst-case outcome," guaranteeing at least a certain amount of money, regardless of what B plays. On the other hand, B's perspective is to optimize the following:

$$\min_y \alpha(y) = \min_y \max_x x^T My. \tag{**}$$

The **Nash equilibrium** is attained in this case, meaning $\beta(x^*) = \alpha(y^*)$.

The question is, how do we compute $x^*$ and $y^*$? Not surprisingly we convert the problem into LP.

*LP for Two-Player Zero-Sum Games.* First fix a strategy $x$ picked by A. In B's perspective, the goal is to minimize $\min_y x^T My$, where $y$ is a probability distribution, i.e. $\sum_{i=1}^n y_i = 1$ and $y \geqslant 0$. Unfortunately, this is one step behind (**) since the strategy previously chosen by A is allowed to vary, and we have no control over $\max_x x^T My$.

For now, keep $x = (x_1, \cdots, x_m)$ fixed. To dualize $\min_y x^T My$, we note that it can be written as

$$\min_y b^T y \qquad \text{subject to} \qquad A^T y \geqslant c, y \geqslant 0 \tag{1}$$

where

$$b = M^T x, \quad A = \mathbf{1}, \quad \text{and } c = 1.$$

So the dualization formula (or the table) gives the dual for this particular $x = (x_1, \cdots, x_m)$:

$$\max x_0 \in \mathbb{R} \qquad \text{subject to} \qquad \mathbf{1}x_0 \leqslant M^T x \iff M^T x - \mathbf{1}x_0 \geqslant 0. \tag{2}$$

Therefore, consider this LP over all valid strategies $x$, we allow $x_1, \cdots, x_m$ to vary by modifying the constraints and treating each $x_i, i \geqslant 1$ also as variables:

$$\max x_0 \in \mathbb{R} \qquad \text{subject to} \qquad \begin{cases} M^T x - \mathbf{1}x_0 \geqslant 0 \\ \sum_{i=1}^m x_i = 1, x_i \geqslant 0. \end{cases} \tag{3}$$

The duality theorems tell us $(2) \leqslant (1)$ for each instances of $x = (x_1, \cdots, x_m)$ and strong duality tells us the optimal values match. Hence, when taken all possible probably distributions $x$,

$$\max_x \min_y b^T y = \operatorname{argmax}_x (3).$$

In other words, if $\tilde{x}_0$ is the optimal value of (3), which is attained by $(x_1, \ldots, x_n) = \tilde{x}$, then

$$\tilde{x}_0 = \max(3) = \max_x (1) = \max_x \min_y x^T My = \max_x \beta(x).$$

Using the same approach we can find $\tilde{y}_0$ and $\tilde{y} = (y_1, \cdots, y_n)$ such that $\tilde{y}_0 = \min_y \alpha(y)$.

# 2 Network Optimization

Beginning of 09/09/2024

## 2.1 Max-Flow, Min-Cut, and the Ford-Fulkerson Algorithm

*Note: a significant portion of the following section (on Max-Flow, Min-Cut, Ford-Fulkerson, etc.) is a combination of 532 lecture content and what I wrote previously for an undergrad algorithm class at USC [link]. I decided to do so, since due to the fast pacing of 532, some intuitions (e.g. why residual graph? what does it do?) were inevitably skipped.*

In this section we are back dealing with graphs, but instead of assigning a weight/cost to each edge, we introduce the notion of *edge capacities*. We will be using graphs to model *transportation networks*. For example, think of a highway system, where the edges, represented by highways, can each tolerate a specific amount of traffic (edge capacities), and they intersect at interchanges, represented by nodes, that serve the purpose of merging or diverting traffic.

There are several main components of such network models: (i) a graph representing the entire system, (ii) **edge capacities** representing the width of each road, (iii) a **source** node that only generates traffic (think of an arrival-only airport), (iv) a **sink** node that only absorbs traffic (think of an departure-only airport), and (v) the actual **flow** (traffic) traversing through the graph via the edges.

Formally, we define the flow as follows:

**Definition: Flow**

Let $G = (V, E)$ be a directed graph[6]with edge capacities $c(e)$ be given. Pick a **source** vertex $v \in V$ and a **sink** $t \in V$. A $\boldsymbol{s-t}$ **flow** is a function $f : E \to \mathbb{R}$ satisfying

- (Conservation) For each node $v \in V \backslash \{s,t\}$,

$$\sum_{(u,v)} c(u,v) = \sum_{(v,u)} c(v,u),$$

  i.e., for any non-source, non-sink vertex, the flow *into* into it equals the flow leaving it.

- (Capacity constraints) For each edge $(u,v)$, $0 \leqslant f(u,v) \leqslant c(u,v)$.

We define

$$\partial f(v) := \sum_{e \text{ leaving v}} f(e) - \sum_{e \text{ into v}} f(e).$$

(The conservation constraint then says $\partial f(v) = 0$ for all non-source, non-sink vertices.) The **value** of the flow is defined as $|f| = \partial f(s)$, the amount of flow leaving $s$, which by conservation equals $-\partial f(s)$, the amount of flow into $t$.
The **Max-Flow** aims to calculate $\max|f|$, i.e.:

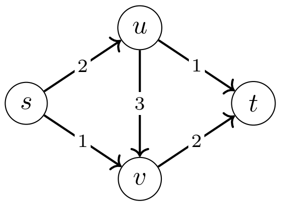
$$\max \partial f(s) \qquad \text{subject to} \qquad \begin{cases} \partial f(v) = 0 & \text{for all } v \neq s,t \\ 0 \leqslant f(e) \leqslant c(e) & \text{for all } e \in E. \end{cases}$$

[6]If the graph is directed, $(u,v)$ means the directed edge $u \to v$.

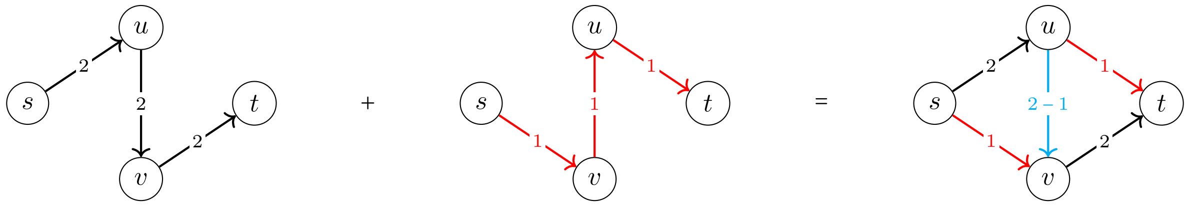
Naturally, we attempt a greedy-alike algorithm, using viewing a flow as sending various amounts of flow along various $s-t$ paths: while there is a $s-t$ path that we can "stuff" more flow into it, we do so. Put more formally:

**Algorithm 2:** Max-Flow: greedy approach

```
1 // initializations, etc.
2 while there is a $s-t$ path on which all edges have capacity remaining do
3     pick one such path $P$, and put as much flow as possible on it (i.e. $\min$ remain capacity)
4 // return
```



It is clear that this greedy approach returns a valid flow, because nowhere in the algorithm did we break the edge capacity constraints. However, as the example on the left demonstrates, it does not necessarily return the maximum flow. There is only one valid $s\to t$ path: $s\to u\to v\to t$, so we send a 2 units of flow along it. But then there is no path left, even though the following diagrams show that it is possible to construct a flow with $|f|=3$.



Greedy: $|f|=2$ Augment Max-Flow: $|f|=3$

The problem in this specific example is that the edges $s\to v$ and $u\to t$ are never used, because greedy considers $u\to v$ as a unidirectional edge. **We can solve this problem by "undoing" flows.** Specifically, we push 1 unit of flow along $s\to v$ because we want to increase $|f|$. But now, $v$ is receiving more flow than its output capacity, so we "undo" 1 unit of flow along $u\to v$. But then $u$ receives 2 units of flow from $s$ and is only currently outputting 1 to $v$, so we send the other surplus unit of flow to $t$. In essence, we created an additional, hidden flow through $s\to v\to u\to t$, even though the directed edge $v\to u$ does not exist in the graph.

*So far, I haven't been able to find an intuitive explanation of the idea of "undoing flows" using real-life examples. Using the highway example, it simply doesn't make sense if we ask some cars to drive backwards. And I don't think making a one-way road two-way is a proper explanation either, as that essentially makes the directed graph undirected. Any additional feedback would be appreciated.* However, one thing for sure is that the resulting flow still satisfies both flow properties.

To formalize these, we introduce **residual graphs** and two types of auxiliary edges: **forward** and **backward edges**.

**Definition: Residual Graphs**

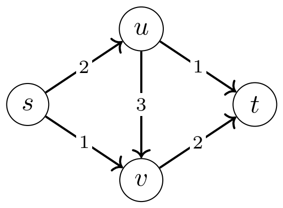
Given a capacity-embedded graph $G=(V,E)$ and an $s-t$ flow $f$ on it, the **residual graph** $G_f$ is defined as follows:

- The nodes are set to be $V$, identical to $G$.
- For each edge $e=(u,v)$ of $G$:
  - If $f(e)<c(e)$, there are $c(e)-f(e)$ "leftover" capacity, so we add a **forward edge** $(u,v)$ with

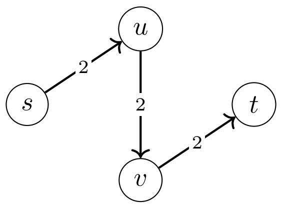
capacity $c(e) - f(e)$ to $G_f$.

- If $f(e) > 0$, then there are $f(e)$ amount of flow that we can "undo," so we add a **backward edge** $(v, u)$ [*note the direction*] with capacity $f(e)$ to $G_f$.

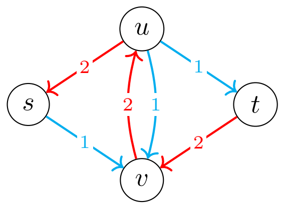
With these definitions, we now look at our simple example again. The residual graph corresponding to the greedy flow $f$ is drawn below, with blue edges representing forward edges, and red edges representing backward edges.



Original graph $G$



Greedy flow $f$



Residual graph $G_f$

Here is an alternate way to explain how we improved our greedy $f$ to Max-Flow. Notice that there is precisely one $s - t$ path remaining in this residual graph $s \to v \to u \to t$, and we see that this path can transmit up to $1$ unit of flow. This is precisely how we updated the original graph $G$: we **augmented** $1$ unit of flow along this path found in the residual graph $G_f$. Specifically, we pay attention to the unidirectional $u \to v$ path in the original graph $G$: the augmentation process essentially un-sends $1$ unit of flow from $u \to v$, since the $v \to u$ path in the residual graph $G_f$ is an backward edge. Therefore, **when augmenting $G$ using $G_f$, we need to pay attention to whether each edge is a forward or backward edge in $G_f$.**

Now we are ready to cook up the main algorithm for solving Max-Flow. This is known as the **Ford-Fulkerson algorithm**, proposed in the 1950s.

(The algorithm has too many lines to be fit into the remaining space of this page lol.)

**Algorithm 3:** Ford-Fulkerson Algorithm

```
1  Inputs: directed graph $G = (V, E)$ with edge capacities $c(e)$; source $s$, sink $t$
2  start with zero flow, i.e., $f(e) = 0$ for all edge
3  while residual graph $G_f$ contains an $s - t$ path do
       let $P$ be one such $s - t$ path
       augment($f, P$), update flow
       compute_residual_graph($G_f, f$), update residual graph
7  return flow $f$

8  Function compute_residual_graph(Graph $G = (V, E)$, flow $f$):
       start with empty edge set $G_f = (V, \varnothing)$
       for each edge $e = (u, v) \in E$ do
           if $f(e) < c(e)$ then add $(u, v)$ to $G_f$ with capacity $c(e) - f(e)$
           if $f(e) > 0$ then add $(v, u)$ [reversed order] to $G_f$ with capacity $f(e)$
       return residual graph $G_f$

14 Function augment(flow $f$, path $P$):
       let $\epsilon$ be the smallest residual capacity along path $P$ in $G_f$
       for each edge $e = (u, v) \in P$ do
           if $e$ is a forward edge then
               $f'(e) \leftarrow f(e) + \epsilon$
           else
               $f'(e) \leftarrow f(e) - \epsilon$ // backward edge
       return flow $f'$
```

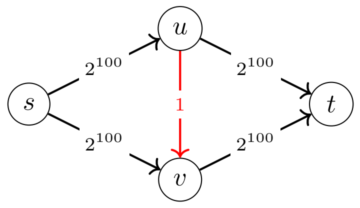
In some sense, the augmentation is a variation of the simplex algorithm when viewing Max-Flow as a LP. The following theorem establishes the correctness of the Ford-Fulkerson algorithm.

**Theorem**

If there are no more $s-t$ paths in $G_f$, then $f$ is a max flow. Consequently, Ford-Fulkerson algorithm correctly finds a maximum $s - t$ flow.

Without proving the theorem, we first analyze the runtime of Ford-Fulkerson, assuming all capacities are integers. An immediate observation is that all augmenting paths are integral as well.

We know each iteration will take $\mathcal{O}(m)$ time since we can use pathfinding algorithms like BFS to compute an $s - t$ path. But how many iterations? If we just pick *any* path to augment on, we can carefully construct "bad" graphs on which we can only slowly increase the capacity. Therefore the complexity is $\mathcal{O}(m \cdot |f^*|)$. This is *pseudopolynomial*.



An easy example to visualize this is shown on the right. If Ford-Fulkerson unfortunately chooses to start with $s \to u \to v \to t$, then it will iterate between augmenting along $s \to u \to v \to t$

and $s \to v \to u \to t$ forever. Each time the flow can only increase by $1$, since the residual edge $(u, v)$ or $(v, u)$ has capacity at most $1$. The input can be represented using around $100$ bits, but the Max-Flow has value $2^{101}$ and the runtime is... astronomical.

What if capacities can be irrational? Even more problems arise.

**Solutions?** We mention two heuristics to fix this problem:

(1) **Always use the widest path**, i.e., path $P$ with the largest minimum residual capacity among all $s-t$ residual paths. This reduces the number of iterations to $\mathcal{O}(m \log|f^*|)$, so the algorithm is now (weakly) polynomial. (This works for rational capacities.)

(2) **Always pick the shortest path** (known as **Edmonds-Karp Algorithm**), i.e., one with the minimum number of edges. This leads to a total of $\mathcal{O}(mn)$ iterations with each iteration taking $\mathcal{O}(m)$, free of $|f^*|$, making the algorithm *strongly polynomial*. (This also works for irrational capacities.)

***Proof of Ford-Fulkerson optimality*.** The proof involves a new concept known as a **cut** on graph, and under this context specifically, ***s* − *t* cuts**. It is defined as a partition $(A, A^c)$ of $V$ such that $s \in A$ and $v \in A^c$ (or equivalently $v \notin A$). The **capacity** of a cut is defined as the total capacity of edges *leaving* $A$:

$$c(A) = c(A, A^c) = \sum_{\substack{(u,v)\in E \\ u\in A \\ v\notin A}} c(u, v).$$

The intuition becomes immediately after proving the following two claims, for we can view cuts and flows as duals, and a Max-Flow corresponds to a Min-Cut.

Claim 1. If $f$ is an $s-t$ flow and $(A, A^c)$ an $s-t$ cut, then $|f| \leqslant c(A)$.

*Proof of claim 1.* The proof is just a series of algebra. Let $f$ and $(A, A^c)$ be given. Then by the conservation of internal nodes,

$$|f| = \partial f(s) = \partial f(s) + \sum_{v\in A\setminus\{s\}} \underbrace{\partial f(v)}_{=0} = \sum_{e \text{ leaving } s} f(e) - \underbrace{\sum_{e \text{ into } s} f(e)}_{=0 \text{ since } s \text{ is source}} + \sum_{v\in A\setminus\{s\}} \left( \sum_{e \text{ leaving } v} f(e) - \sum_{e \text{ into } v} f(e) \right). \tag{1}$$

What is the double sum at the end? For an edge $e = (u, v)$:

- If $u \in A, v \in A$, then $e$ appears twice in the sum, once as the "edge leaving $u$" with "+", and the other as "edge into $v$" with "−," so they cancel each other.
- If $u \notin A$ but $v \in A$, only the negative term "$e$ into $v$" appears.
- If $u \in A$ but $v \notin A$, only the positive term "$e$ leaving $u$" appears.
- If $u, v \notin A$, $f(e)$ won't appear at all.

(Note these cases also apply to edges incident on the source $s$.) Therefore,

$$|f| = \sum_{e \text{ leaves } A} f(e) - \sum_{e \text{ enters } A} f(e),$$

and because flows are nonnegative,

$$|f| = \sum_{e \text{ leaves } A} f(e) - \underbrace{\sum_{e \text{ enters } A} f(e)}_{\geqslant 0} \leqslant \sum_{e \text{ leaves } A} f(e) \leqslant \sum_{e \text{ leaves } A} c(e) = c(A, A^c). \tag{*}$$

END OF PROOF OF CLAIM 1.

CLAIM 2. $|f| = c(A)$ if and only if:

$$\begin{cases} f \text{ is a max flow} \\ c \text{ is a min cut} \end{cases} \quad \text{and} \quad \begin{cases} f(u,v) = c(u,v) & \text{for all edges } (u,v) \text{ leaving } A \\ f(u,v) = 0 & \text{for all edges } (u,v) \text{ entering } A. \end{cases}$$

*Proof.* Stare at (*) until this becomes clear. END OF PROOF OF CLAIM 2.

So we are done! Why? If there are no more $s-t$ paths in $G_f$, we naturally design a cut based on whether each vertex is reachable from the source:

$$S := \{v \in V : \text{ there still exists a valid } s-v \text{ path}\}.$$

For every edge $e = (u,v)$ crossing the cut $(S, S^c)$ [i.e. $u \in S, v \notin S$], we must *not* have a forward edge $u \to v$ in $G_f$, for otherwise appending the existing $s \to u$ path with $u \to v$ gives a path $s \to v$, proving $v \in S$. By the definition of forward edges this means $f(u,v) = c(u,v)$.
On the other hand, for ever edge $(u,v)$ entering $S$, we must have $f(u,v) = 0$, for we must *not* have a backward edge $u \to v$ in $G_f$ based on the same reasoning. Therefore the first claim implies

$$|f| = \sum_{e \text{ leaves } S} f(e) - \sum_{e \text{ enters } S} f(e) = \sum_{e \text{ leaves } S} c(e) - \sum_{e \text{ enters } S} 0 = c(S, S^c)$$

and the second claim shows $f$ is a max flow. □

**Remark.** A simple modification to the Ford-Fulkerson is that in each iteration, instead of updating the residual graph $G_f$, we simply remove the saturated flow. Rinse and repeat. It can be shown that this algorithm can be implemented in $\mathcal{O}(mn \log n)$ time. This is particularly good when $n$ is small, with a runtime of nearly $\mathcal{O}(n)$.

Beginning of 09/11/2024

## 2.2 The Push-Relabel Algorithm

The previous algorithms maintain a feasible flow $f$ and terminates when no such feasible $s-t$ path in $G_f$ exists. In PUSH-RELABEL, however, we maintain a **preflow**, such that (i) there is no $s-t$ flow *anytime* during the algorithm, and (ii) the algorithm terminates immediately when $f$ *becomes* a flow. First, definitions.

**Definition: Preflow**

A **preflow** is a function $f : E \to \mathbb{R}^{\geqslant 0}$ such that:

- (Capacity constraint) For each edge $(u,v)$, $0 \leqslant f(u,v) \leqslant c(u,v)$.

- For each $v \neq s$, the **excess** (flow in minus flow out)

$$\chi_f(v) = \sum_{(u,v)} f(u,v) - \sum_{(v,w)} f(v,w) \geqslant 0.$$

(Note this is $-\partial f(v)$.) This means there is a certain amount of congestion at each vertex.

For each vertex $v \in V$, we should also assign a non-negative ineteger **label** $h : V \to \mathbb{Z}^{\geqslant 0}$ that represents the "height" of a vertex. Whenever there is an excess, we adjust the label/height and push the flow forward. We initialize $h(s) = n = |V|$ and $h(t) = 0$.

Finally, we say an $s-t$ preflow $f$ is **compatible** with label $h$, written $f \uparrow h$, if

- (Source and sink conditions) $h(s) = n, h(t) = 0$
- (*Steepness condition*) For each edge $(u,v) \in E_f$ (residual edges), $h(v) \leqslant h(w) + 1$. Namely, if an residual edge is going downward, it must not be too steep (upward edges have no constraints).

The algorithm is based on the physical intuition that flow naturally finds its way downhill (and downhill only). The height difference $n$ can be interpreted twofold: it establishes sufficient difference between the source $s$ and the sink $t$ so there will be enough "momentum" to reach the sink, while the steepness condition ensures that the momentum is not too high (i.e. the flow shouldn't flow downhill via a path that is too steep) so it stops by the time it reaches $t$, effectively making $t$ a sink. To associate a flow with a compatible label, we will need the following claim.
**Claim**. If $f \uparrow h$, then $t$ cannot be reached from $s$ in $G_f$.

*Proof.* By the steepness condition, along the edge edge, the height can decrease by at most 1. On the other hand, any $s-t$ path is at most of length $n-1$, so $h(s) = n, h(t) = 0$ makes the existence of a path impossible. □

Consequently, by the correctness of Ford-Fulkerson, **if an $s-t$ flow $f$ is compatible with $h$, then $f$ is a Max-Flow.** To get started with modifying a preflow, we need some initialization first. Since $h(s) = n$, one easy way to ensure that $f$ is compatible with $h$ is by ensuring there are no edges in $E_f$ leaving $s$. That is, we saturate every single edge leaving $s$ a priori. Therefore, the following $s-t$ preflow and label $h$ are compatible:

$$f(e) = \begin{cases} c(e) & \text{for all edge } e \text{ leaving } s \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \begin{cases} h(s) = n \\ h(v) = 0 & \text{for } v \neq s. \end{cases} \tag{2}$$

So now we have a starting point. How do we push and relabel? Intuitively, we want to send the flow "downward." Let us consider the `push` function first. Say we have an edge $(u,v)$ in the residual graph $E_f$. We should "push" it if and only if (i) the starting node $u$ has some excess, *and* (ii) $u$ is "higher" than $v$, so the (pre)flow can indeed move downwards. And we can push as much flow as we want until we can't. What should our terminal condition be? Well, we certainly should stop pushing when $u$ runs out of excess. Alternatively, if the edge $(u,v)$ disappears in $E_f$, then we should stop too — for forward edges, this means saturating the edge, and for backward edges, draining it to zero.

Observe that we are still missing a piece — by initialization, $h(v) = 0$ for all non-source nodes. But we said the flow should flow downward, not being trapped somewhere! This means we need some function to modify the "terrains." To this end, we define the `relabel` function: if $v$ still has excess but the excess has nowhere to go because all of its neighbors as as high as $v$, then we increase $h(v)$.

Now we covered everything to cook up the Push-Relabel Max-Flow algorithm.

**Algorithm 4:** Push-Relabel Max-Flow Algorithm

```
Inputs: directed graph $G = (V, E)$ with edge capacities $c(e)$; source $s$, sink $t$
Initialization: define preflow $f$ and label $h$ as in (2)
--------------------------------------------------------------------
while there exists $v \neq t$ with excess $\chi_f(v) > 0$ do
    choose one such vertex $v$
    if there exists $(v, w) \in E_f$ such that $h(v) > h(w)$ then
        push$(v, w, h, f)$
    else
        relabel$(v)$
    return $(f, h)$
--------------------------------------------------------------------
Function push(vertices $v, w$, flow $f$, label $h$):
    # guaranteed: $v$ has excess, and $(v, w)$ is a downward edge w.r.t. $h(\cdot)$
    if $e = (v, w)$ is a forward edge then
        $\delta = \min\{\chi_f(e), c_f(e)\} = \min\{\chi_f(e), c(e) - f(e)\}$
        $f(e) \leftarrow f(e) + \delta$
    if $e = (v, w)$ is s a backward edge then
        $\delta = \min\{\chi_f(e), c_f(e)\} = \min\{\chi_f(e), f(e)\}$
        $f(w, v) \leftarrow f(w, v) - \delta$.
--------------------------------------------------------------------
Function relabel(vertex $v$):
    # the if below is guaranteed: $v$ has excess, but no neighboring $w$ (w.r.t. $E_f$) is lower
    if $\chi_f(v) > 0$ and $h(w) \geqslant h(u)$ for all $(v, w) \in E_f$ then
        $h(v) \leftarrow h(v) + 1$        # increase label for $v$ to gain momentum
```

One loop invariant of the described algorithm is that $(f, h)$ remains compatible throughout. The values of $h(s)$ and $h(t)$ are kept constant. On the other hand, `push`$(f, h, v, w)$ adds only one edge to the residual graph, and this edge satisfies the steepness condition. `relabel`$(v)$ is called only when $v$ *needs* momentum. By raising $h(v)$ by $1$, the steepness condition is also preserved.

Therefore, if the algorithm terminates, it will output a valid Max-Flow based on the previous claim.

## Runtime Analysis of Push-Relabel

Even though the output $(f, h)$ is compatible, it is not clear at all if this algorithm even terminates. To prove such claim, notice that the $h$-value of every non-sink vertex increases by $1$ every time it is relabeled, so it suffices to provide an upper bound for $h$. This is also nontrivial, and it is based on the following observation:

**Claim**. If $f$ is a preflow and $v$ has excess, i.e., $\chi_f(v) > 0$, then there is a path *from* $v$ *to* the source $s$ in $G_f$ [note the reversed direction].

*Proof.* Let $S$ be the set of nodes $v$ that can reach $s$ in $G_f$ (i.e. there exists a path *from $v$ to $s$*). To prove the claim we need to show that if $\chi_f(v) > 0$ then $v \in S$.

Since every $w \in S^c$ has no path to $s$, we know that an edge $(w,v)$ from $S$ into $S^c$ cannot exist in $G_f$. In other words, no edge $(v,w)$ leaving $S$ can have positive flow by the definition of residual graph. Using a similar technique to the one we used in proving Ford-Fulkerson, and recalling a preflow has nonnegative excess for every vertex,

$$0 \leqslant \sum_{v\in S^c} \chi_f(v) = \sum_{v\in S^c} \Big( \sum_{e \text{ into } v} f(e) - \sum_{e \text{ leaving } v} f(e) \Big) = \sum_{e=(u,v)} \begin{cases} 0 & \text{if } u,v \in S^c \\ f(u,v) & \text{if } u \in S, v \in S^c \\ -f(u,v) & \text{if } u \in S^c, v \in S. \end{cases}$$

Note that in the second case, since $(u,v)$ is leaving $S$, as noted earlier $f(u,v) = 0$. So

$$0 \geqslant -\sum_{e \text{ leaves } S^c} f(e) = \sum_{v\in S^c} \chi_f(v) \geqslant 0$$

and so no vertex in $S^c$ has positive excess. This proves the claim. □

Now consider any $v$. We know that $h(v)$ changes only when `relabel`$(v)$ is called, and (one of) the condition(s) to trigger such function call is if $v$ has excess. The previous claim implies that there is a path $P$ from $v$ to $s$. Since $P$ can be at most of length $n-1$, $h(v) - h(s) \leqslant n-1$, and thus

$$h(v) \leqslant 2n-1 \qquad \text{for all } v \in V.$$

This immediately implies that **the algorithm terminates** and that each vertex is relabeled at most $2n-1$ times. Next up, we compute bounds on the number of `push` operations. We say a `push`$(v,w,h,f)$ operation is considered *saturating* if $f(v,w) = c(v,w)$ after we push, i.e., the edge along which we push becomes saturated.

**Claim**. The number of saturating `push` is bounded by $2mn$; (not covered in lecture) the number of non-saturating `push` operations is bounded by $2n^2m$.

*Proof.* (Saturating pushes) Consider an edge $(v,w) \in E_f$. Once $(u,v)$ is saturated, we can only push along $(v,w)$ again if this edge reappears in $E_f$. That means before our next $(v,w)$ push, we must have pushed in the reverse direction $(w,v)$. But in order to push in this reverse direction, $w$ needs to be "higher" than $v$, so starting from $h(w) = h(v) - 1$, we need to at least increase the height of $w$ by 2, to $h(w) = h(v) + 1$. This means a saturating push from $v$ to $w$ can occur at most $n-1$ times since $\max h(w) \leqslant 2n-1$. Each edge $e \in E$ gives rise to two residual edges, and for each residual edge we can conduct saturating push at most $n-1$ times. That gives a total of $\leqslant 2mn$ saturating pushes.

(Non-saturating pushes) Drawing intuition from physics again, this time defining the **potential** of a $(f,h)$ pair to be the sum of heights of all nodes with positive excess:

$$\Phi(f,h) = \sum_{\chi_f(v)>0} h(v).$$

Initially $\Phi(f,h) = 0$. Consider what happens after a non-saturating push on $(v,w)$: after the push, $v$ will have no excess, and the best case is that $w$ now gains excess. But even so $h(w) = h(v) - 1$ implies $\Phi(f,h)$ decreased by 1. Now consider what happens after a saturating push. It does not change the label function $h(\cdot)$ but may give $w$ positive excess while retaining $v$'s status of having positive excess. Therefore, in each saturating push, $\Phi(f,h)$

can increase by at most $(2n-1)$. By the previous part, there are $\leqslant 2mn$ saturating pushes.
Finally, each `relabel` operation can increase $\Phi(f,h)$ by $1$ since it increases the $h$ value of some vertex. And we know that the number of calls of `relabel` $\leqslant 2n^2$ ($n$ nodes, with $h(v) \leqslant 2n-1$ for each $v$).
Combining all three parts and the observation that $\Phi$ remains nonnegative by definition, since $\Phi$ can be increased by at most $4mn^2$ during the algorithm, it can allow no more than $4mn^2$ non-saturating pushes. □

To wrap up these analyses, when implemented naïvely, Push-Relabel attains a time complexity of $\mathcal{O}(mn^2)$, which is asymptotically more efficient than varaints of Ford-Fulkerson, including Edmonds-Karp (which is $\mathcal{O}(m^2n)$). Further *theoretical* optimizations are possible: using **dynamic trees**, with proper modifications, Push-Relabel can be implemented in $\mathcal{O}(mn\log n)$.

Some recent works:

- In *approximate* Max-Flow where one relaxes the constraint to $f(u,v) \leqslant (1+\epsilon)c(u,v)$, Kelner et al. and Sherman independently produced almost-linear-time algorithms.
- In exact Max-Flow, Chen et al. gave an almost-linear-time algorithm assuming integer capacities.

Beginning of 09/15/2024

## 3 Minimum-Cost Flows

In this section, we consider a significant generalization of the Max-Flow problem. In addition to edge capacities, which we now use $u(e)$ to represent, we also assign a **cost** $u(e)$ to it. The same constraints apply: for internal nodes, flows are conserved, and the amount of flow through each edge must be bounded by the capacity constraint. We are familiar with Ford-Fulkerson and are therefore able to compute a maximum $s-t$ flow. The **Min-Cost Max-Flow** (MCF) asks us to compute the cheapest $s-t$ flow that attains this flow value. In equations, under the same flow constraints, we want to find

$$f^* = \arg\min_{f:|f|=Q^*} c(f) \qquad \text{where} \qquad Q^* = \max_{f \text{ is a flow}} |f|.$$

We first note that the shortest $s-t$ path is a specific instance of the MCF. The reduction goes as follows. We augment nodes $s', t'$, such that the directed edge $(s', s)$ has capacity $u(s', s) = 1$ and cost $c(s', s) = 0$. Likewise, $u(t, t') = 1$ and $c(t, t') = 0$. For each edge in the original graph $G$, set capacity to be $1$ and cost to be the edge length. Any Max-Flow from $s'$ to $t'$ has a value of $1$ (since it wants to saturate $(s', s)$, and the MCF must use as few edges as possible to travel from $s$ to $t$. Since $c(s', s) = c(t, t') = 0$, the cost of MCF is precisely the shortest distance from $s$ to $t$.

### 3.1 Min-Cost Circulation

In today's lecture, the main topic is to discuss the **minimum cost circulation** (MCC). In this problem there are no source and sink. Think of an electric circuit where currents are looping through it. Now, each edge $e$ has a real-valued (possibly negative) cost $c(e)$ and a capacity $u(e) \geqslant 0$ associated with it. Our goal is to minimize the total cost of the flow, or circulation:

$$\min c(f) := \min_{e \in E} c(e) f(e) \qquad \text{subject to} \qquad \begin{cases} \sum_{e \text{ leaves } v} f(e) = \sum_{e \text{ into } v} f(e) & \text{for every vertex } v \\ 0 \leqslant f(e) \leqslant u(e) & \text{for every edge } e. \end{cases}$$

In other words, all vertices must have a balance $\partial f(v) = 0$.
Before discussing MCF in more depth, we first show that MCC is a special case of MCF. Consider an instance of MCF on $G = (V, E)$ with source $s$, sink $t$. Construct $G'$ as a copy of $G$ with the following modifications[7]:

(1) Set every edge that originally belongs to $E$ to have cost $0$.

(2) Add a directed edge $(t, s)$. Set its capacity to be $\infty$ and cost $-1$.

Now if $f$ is a flow in $G$, then it induces a canonical circulation $f'$ in $G'$ defined by

$$\begin{cases} f'(t, s) = |f| \\ f'(e) = f(e) \text{ for all } e \neq (t, s). \end{cases}$$

Since every other edge in $G'$ has cost $0$, the cost of $f'$, $c(f') = -|f|$. Conversely, for any circulation $f'$ on $G'$, restricting its domain to $E = E' \setminus \{(t, s)\}$ gives a valid $s-t$ flow whose value is $|f|$. So the reduction is complete.

[7]In lecture we used the same idea but slightly different values for the additional edge. The construction I used here is based on Jeff Erickson's book/notes.

Like how we can break down an $s-t$ flow into the superposition of constant flows along various $s-t$ paths, we can view circulation as a superposition of cycles. From this perspective, it is immediately clear that if $G$ does not contain a negative *cost* cycle, then the min-cost circulation is the trivial one $f(e) = 0$ for each $e$, for we will be charged extra sending any flow along any cycle.

What to do when there are negative cost cycles? Fill them, of course. We solved Max-Flow by using Ford-Fulkerson, where we iteratively augment along any existing $s-t$ path in the residual graph. Here, the intuition remains exactly the same: we want to create a residual graph $G_f$ to keep track of how much we can increase or "undo" cycles, and iteraitve remove cycles until there are no negative cycles left. This is known as the **cycle cancellation algorithm**.

As usual, we assume that the capacities and costs are integers. Since the augment reduces the objective by at least $1$ each iteration, a very crude bound to the total number of iterations is the magnitude of the min cost, which can further be crudely bounded by $m \cdot u_{\max} \cdot |c_{\max}|$. Recall that we can use Bellman-Ford to detect negative cycles in $\mathcal{O}(mn)$ time. Therefore the algorithm runs in $\mathcal{O}(m^2 n \cdot u_{\max}|c_{\max}|)$.

But which negative cycle should we choose? One natural thought is to choose the most negative one, as augmenting along this cycle reduces the objective the most. However, finding the most negative cycle is NP hard since a natural reduction to Hamiltonian path exists by constructing a new graph whose edge weights are $-1$. A workaround that we propose is to find the *minimum mean cost negative cycle* ,where the mean cost of a cycle is the total cost divided by the number of edges inside it. The cool thing about using mean cost is that if we increase the cost of *every* edge in $G_f$ by $\delta$, then every cycle's *mean* cost increases by the same amount. Therefore, to find the minimum mean cost negative cycle, we simply need to conduct a binary search on $\delta$ and find the threshold above which there are no negative mean cost cycles anymore. With proper implementation, this helps us identify the min mean cost negative cycle we want. (With modified Bellman-Ford this can even be achieved in $\mathcal{O}(mn)$.)

**Algorithm 5:** Cycle Cancellation Algorithm for MCC

```
Inputs: directed graph $G = (V, E)$ with capacities $u(e) \geqslant 0$, costs $c(e) \in \mathbb{R}$
----------------------------------------
start with zero flow/circulation, i.e., $f(e) = 0$ for all edge
while residual graph $G_f$ contains a negative cycle do
    let $C$ be one such cycle
    augment($f, C$), update flow
    compute_residual_graph($G_f, f$), update residual graph
return flow/circulation $f$
----------------------------------------
Function compute_residual_graph(Graph $G = (V, E)$, flow $f$):
    start with empty edge set $G_f = (V, \varnothing)$
    for each edge $e = (v, w) \in E$ do
        if $f(e) < u(e)$ then add $(v, w)$ to $G_f$ with capacity $c(e) - f(e)$ and cost $c(v, w)$
        if $f(e) > 0$ then add $(w, v)$ [reversed order] to $G_f$ with capacity $f(e)$ and cost $-c(w, v)$
    return residual graph $G_f$
----------------------------------------
Function augment(flow $f$, negative cycle $C$):
    let $\epsilon$ be the smallest residual capacity along path $C$ in $G_f$
    for each edge $e \in C$ do
        if $e$ is a forward edge then
            $f'(e) \leftarrow f(e) + \epsilon$
        else
            $f'(e) \leftarrow f(e) - \epsilon$ // backward edge
    return flow $f'$
```

**Optimality Proof**

To directly prove the optimality of the algorithm above is hard. Instead, we will introduce two additional notions on the residual graph $G_f$:

- A **price** (or potential) is a real-valued vertex function $p : V \to \mathbb{R}$, and it gives rise to the
- **Reduced cost** $c_p(v, w)$ of an edge: $c_p(v, w) = c(v) + c_f(v, w) - c(w)$.

It immediately follows from definition that the reduced cost is antisymmetric: $c_p(v, w) = -c_p(w, v)$, and that the total reduced cost of a cycle equals the cost itself. The following theorem establishes the optimality condition as well as some intuition behind the price function:

**Theorem**

The following are equivalent (TFAE):

(1) $f$ is a MCC,

(2) $G_f$ has no negative cycles, and

(3) There exists a price function $p$ such that $c_p(v,w) \geqslant 0$ for all $(v,w) \in E_f$.

*Proof.* (1) $\Rightarrow$ (2). The contrapositive is already stated int he algorithm: if $G_f$ contains a negative cycle then we can augment along it, creating a flow/circulation $f'$ with cheaper total cost.

(2) $\Rightarrow$ (3). The intuition is that *we can view $p(\cdot)$ as a shortest distance function, with respect to some fixed vertex, say $v$.* We define $p(w)$ to be the cheapest cost to travel from $v$ to $w$ in $G_f$. Then triangle inequality suggests that for all $(i,j) \in E_f$, $p(j) \leqslant p(i) + c_f(i,j)$. Rewriting this gives

$$c_p(i,j) = c_f(i,j) + c(i) - c(j) \geqslant 0.$$

(3) $\Rightarrow$ (1) is more involved and is not covered in lecture.. The following proof references Theorem 5.3 of Williamson's book *Network Flow Algorithms.*[8]
Let $\tilde{f}$ be any other flow/circulation and consider the difference $f' = \tilde{f} - f$. By assumptions,

$$\begin{aligned} c(\tilde{f}) - c(f) = c(f') &= c(f') + \sum_{v \in V} p(v) \underbrace{\left( \sum_{e \text{ leaves } v} f'(e) - \sum_{e \text{ into } v} f'(v) \right)}_{=0 \text{ by conservation}} \\ &= c(f') + \sum_{(u,v)\in E} (p(u) - p(v)) f'(u,v) \\ &= \sum_{(u,v)\in E} (c(u,v) + p(u) - p(v)) f'(u,v) \\ &= \sum_{(u,v)\in E} c_p(u,v) f'(u,v) \geqslant 0. \end{aligned}$$

□

Beginning of 09/18/2024

## 3.2 Minimum-Weight Bipartite Matching

We first recall the Optimal Transport problem and will introduce the Minimum-Weight Bipartite Matching problem later as a special instance of Optimal Transport. For this section, we assume that $G = (A \sqcup B, E \subset A \times B)$ is a bipartite graph[9]. To simplify notations, we further assume $|A| = |B| = n$ so $A = \{a_1, \cdots, a_n\}$ and likewise $B = \{b_1, \cdots, b_n\}$. Additionally, we will normalize $A$ and $B$ by viewing the values as probability distributions. So we define distributions $\mu$ and $\nu$ on them, respectively, such that $\mu(a_i) = \mu_i, \nu(b_j) = \nu_j$, and $\sum_i \mu_i = \sum_j \nu_j = 1$. We define the **transport map** (or transport **coupling**) as follows:

$$T : E \to \mathbb{R}_{\geqslant 0} \qquad \text{such that} \qquad \begin{cases} (\text{first argument fixed}) \sum_j T(a, b_j) = \mu(a) & \text{for each } a \in A \\ (\text{second argument fixed}) \sum_i T(a_i, b) = \nu(b) & \text{for each } b \in B. \end{cases}$$

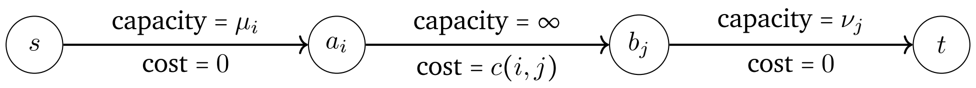
[8]It is worth noting that in Williamson's book, before defining a circulation, for every directed edge $(u,v)$, the reverse edge $(v,u)$ is also added to the graph, with $f(v,u) = -f(u,v)$. In that definition, $c(u,v)f(u,v) + c(v,u)f(v,u) = 2c(u,v)f(u,v)$, and the overall cost of the flow is $1/2 \cdot \sum_{(u,v)} c(u,v)f(u,v)$. These are useful in formally analyzing the runtime of this algorithm, but since the proof is lengthy, it was not covered in lecture and hence the additional edges are not necessary here.

[9]It can be shown that our problem can be defined on more general graphs, but here we focus on bipartite ones. The symbol $\sqcup$ denotes disjoint union to emphasize the bipartiteness.

Naturally, we want to minimize the cost, i.e.,

$$\min_{(a,b)\in E} c(a,b)T(a,b). \tag{OT Primal}$$

Before analyzing Optimal Transport, let us first observe that Optimal Transport is a special case of MCF, in particular, *uncapacitated* MCT (meaning edges have no capacity constraints). To convert an instance of Optimal Transport is easy: add a source node $s$, a sink $m$, and for each $a_i, b_j$, connect the nodes as follows:



A max flow will saturate every single outgoing edge from $s$, so the flow has value 1. The capacities of each $(s, a_i)$ and $(b_j, t)$ ensures that the conditions for $T(a,b)$ are satisfied, so each flow corresponds to a valid transport coupling. The cost of the flow is precisely cost $\sum_{(a,b)} c(a,b)T(a,b)$.

Back to the LP analysis. To construct the dual, we define one variable for each node in $A \cup B$. Corresponding to the transport coupling, we define a **potential function** $\varphi(x)$ on these dual variables. The dual is given by

$$\max\left[\sum_i \varphi(a_i)\mu_i + \sum_j \varphi(b_j)\nu_j\right] \qquad \text{subject to} \qquad \begin{cases}\varphi(a_i) + \varphi(b_j) \leqslant c(a_i,b_j) & \forall (a_i,b_j) \in E\\ \varphi(x) \in \mathbb{R}\end{cases} \tag{OT Dual}$$

**Minimum-Weight Bipartite Matching**

For the following analysis, we will assume that $\mu_i = \nu_j = 1/n$. If so, we may WLOG scale $T$ by a factor of $n$ and assume that our primal is now

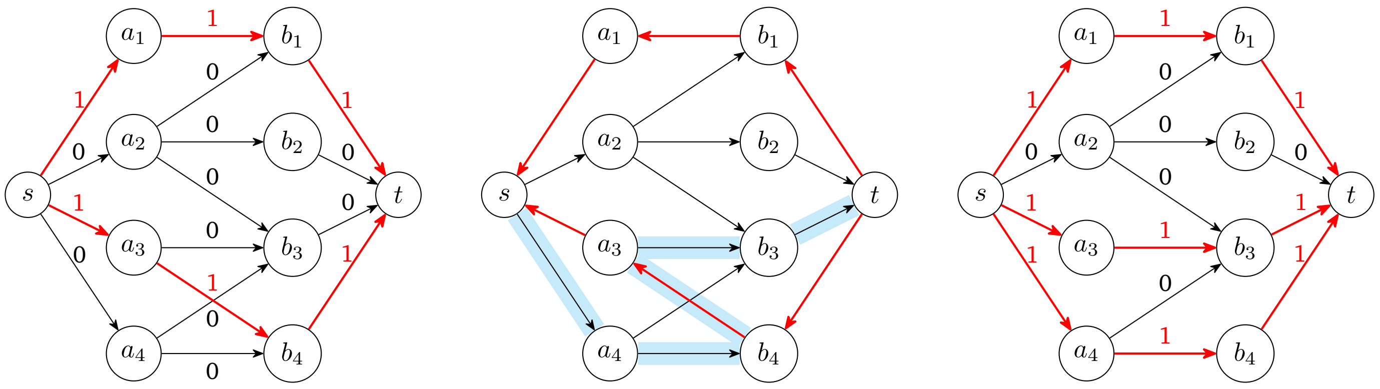
$$\min \sum_{(a,b)} c(a,b)T(a,b) \qquad \text{subject to} \qquad \begin{cases}\sum_j T(a,b_j) = 1 & \text{for each } a \in A\\ \sum_i T(a_i,b) = 1 & \text{for each } b \in B\end{cases} \tag{Modified OT Primal}$$

and the dual is therefore

$$\max\left[\sum_i \varphi(a_i) + \sum_j \varphi(b_j)\right] \qquad \text{subject to} \qquad \begin{cases}\varphi(a_i) + \varphi(b_i) \leqslant c(a_i,b_j) & \forall (a_i,b_j) \in E\\ \varphi(x) \in \mathbb{R}.\end{cases} \tag{Modified OT Dual}$$

We will in HW2 prove that there exists an integer solution for this LP. In other words, **this problem reduces to a minimum-weight bipartite matching problem**.

To solve the LP, we will once again resort to computing a residual graph and iteratively augmenting along the cheapest residual path. The numerical labels in the following illustration (see left) indicate the amount of flow we send along each edge.



We state, without a formal proof, the following fact. Given this extended bipartite graph $s \to A \to B \to t$, we can naturally extract a matching out of it. Every time we augment along a residual graph (see middle), the cardinality of the matching increases by $1$. To see this, we discard the first edge leaving $s$ and the last edge entering $t$. Everything in the middle is zigzagging, alternating between forward and backward edges. The caridnality of the original matching is the number of backward edges, whereas the new matching's carinality matches the number of forwad edges, which is $1$ more.

Therefore, the algorithm terminates after $n$ iterations of augmentation. A natural attempt to minimize the cost is by augmenting along the min-cost path in each iteration. Since negative edge costs are allowed, the naïve approach is to use Bellman-Ford. Each iteration would therefore take $\mathcal{O}(mn)$ time, and the total runtime is $\mathcal{O}(mn^2)$.

### 3.2.1 Deriving the Hungarian Algorithm

How can we improve this? If we can make the edge weights nonnegative, we can use Dijkstra's algorithm instead, achieving a better logarithmic runtime. This is possible by using the price function and reduced cost, concepts we've encountered in the Minimum Cost Circulation (MCC) problem. This approach leads to what is known as the **Hungarian algorithm**.

The nonnegativity condition is given by the dual constraint: for any pair of nodes $a$ and $b$, we know that $c(a,b) - \varphi(a) - \varphi(b) \geq 0$. We denote this as the **reduced cost** $\tilde{c}(a,b)$. Like other primal-dual algorithms, we keep track of the dual constraints that become tight—in this case, the set of edges whose reduced cost equals zero, making them tight. At any given time, the set of tight edges forms a bipartite graph between sets $A$ and $B$. Therefore, if $M$ is a partial matching on this bipartite graph, then

$$\sum_{(a,b)\in M} c(a,b) = \sum_{(a,b)\in M} [\varphi(a) + \varphi(b)]$$

Thus, if $M$ is a perfect matching (i.e., $|M| = n$), we have naturally found a primal solution and a dual solution with equal objective values. By strong duality, this would be the optimal solution.

But how do we find appropriate augmenting paths, and when none exist, how do we create one? Let's step back and enumerate a few goals we aim to achieve during the iterations:

(1) Loop invariant: nonnegative reduced costs. Recall that our initial motivation was to improve upon Bellman-Ford's runtime by transforming the edge costs into nonnegative values. *If we add edges via the dual LP, this is automatically guaranteed, since $\tilde{c} \geq 0$ iff the dual constraint holds. This ensures that every edge in the partial bipartite graph is tight.*

(2) Strictly increasing matching cardinality: if, at the start of an iteration, our tight edges admit a matching of

cardinality $k$, we want to increase it to (at least) $k+1$ by the end of the iteration.

If we augment along the residual graph as described, the loop invariant is trivially preserved: we never modify the reduced cost of any edge. Furthermore, by counting the number of forward and backward edges (excluding the first edge leaving $s$ and the last edge entering $t$), the post-augmentation matching has a larger cardinality.

So, the easy case is done. Now we ask, what if we cannot find such a zigzag path? If no such path exists, we must modify the values of $\varphi$ on certain vertices to saturate some edges and add them to $\tilde{E}$, hoping that the now-larger graph admits a larger matching. But before that, we need a new definition.

Define $\tilde{E}$ to be the set of tight edges after a certain iteration, and consider a matching $\tilde{M}$ defined on $\tilde{E}$. A node $x$ is called a **free vertex** if $\tilde{M}$ has no edge incident on $x$. Based on our previous observation, if $a \in A$ and $b \in B$ are free vertices and a residual path exists from $a$ to $b$, then augmenting along this path increases the cardinality of an existing matching by 1. Assuming integral variable values, the amount of flow along any edge will remain integral as long as the updates are. In this problem, we can even assume they are binary. Therefore, for any adjacent $u, v$, only edges of one direction can appear at a time. This is crucial, since it implies that the second node in any $s-t$ residual path *must* be free (and so is the second last node). Thus, detecting the existence of zigzagging paths in the bipartite graph amounts to finding whether there is an $s-t$ path.

We want to introduce new edges, and of course, these edges should be natural extensions of the existing (unidirectional) zigzag paths. Define $X = \{x \mid x \text{ is reachable from } s \text{ in the residual graph}\}$. It is clear that if a zigzag path starts from a free vertex, then any vertex it passes through is contained in $X$, and vice versa. Therefore, to extend the possibility of constructing zigzagging paths with a new edge, it suffices to ensure that the new addition to the residual graph is a directed edge starting from a node in $X$.

We've been treating our artificial matchings as directed edges from $A$ to $B$, so let us once again consider candidate edges of the form $(a, b)$ with $\tilde{c}(a, b) > 0$, where $a \in A \cap X$. Since all edges in $\tilde{E}$ are tight, this implies that $b \in B \backslash X$. To introduce a new edge to $\tilde{E}$, we need to tighten some edges. The straightforward approach is to increase $\varphi(a)$ by $\tilde{c}(a, b)$ so that the new reduced cost of this edge becomes 0, making the edge tight. However, since there may be multiple edges with $a \in A \cap X$ and $b \in B \backslash X$, we increase each of them by the minimum reduced cost of these edges to avoid violating any dual constraint:

$$\delta = \min_{\substack{a \in A \cap X \\ b \in B \backslash X}} \tilde{c}(a, b).$$

Our reasoning indicated that it is safe to let $\varphi(a) \leftarrow \varphi(a) + \delta$ for each $a \in A \cap X$ without directly violating the constraint $(a, b)$ where $a \in A \cap X, b \in B \backslash X$. But that's not all. What about those edges $(a, b)$ where $a \in A \cap X$ and $b \in B \cap X$ also? These are the edges where originally $\tilde{c}(a, b) = 0$. Therefore, for them, as we increase $\varphi(a)$ by $\delta$, we must correspondingly decrease $\varphi(b)$ by $\delta$.

The following sums up the process. In the last line, we use $\tilde{E}$ to define the set of admissible edges. This set is not used in the algorithm but helpful for a proof later.

$$\begin{aligned}
&\text{let } S \leftarrow \{x \text{ reachable fromsome free } a \in A \text{ in the residual graph}\} \\
&\Longrightarrow \text{ let } \delta \leftarrow \min_{\substack{a \in X \cap A \\ b \in B \backslash X}} \tilde{c}(a,b), \quad \tilde{e} \leftarrow \operatorname*{arg\,min}_{\substack{(a,b) \in E \\ a \in X \cap A \\ b \in B \backslash X}} \tilde{c}(e) \\
&\Longrightarrow \text{ update } \varphi : \begin{cases} \varphi(a) \leftarrow \varphi(a) + \delta & \forall a \in A \cap X \\ \varphi(b) \leftarrow \varphi(b) - \delta & \forall b \in B \cap X \\ \varphi(x) \text{ unchanged otherwise} \end{cases} \\
&\Longrightarrow \tilde{E} \leftarrow \tilde{E} \cup \{\tilde{e}\} \quad \text{(add all if multiple)}.
\end{aligned}$$

It is not at all clear whether our dual objective is increasing, since we are increasing $\varphi$ for some nodes and decreasing for others. This is fortunately true, based on the following observation.

$$|A \cap X| - |B \cap X| = \underbrace{|A \cap X| + |A \backslash X|}_{=|A|=n} - |A \backslash X| - |B \cap X|. \tag{3}$$

What does $|A \backslash X| + |B \cap X|$ represent? Referring to the diagram again, we see that:

- No node $a \in A \backslash X$ is free — otherwise we would have started DFS on $a$, directly adding it to $X$.
- No node $b \in B \cap X$ is free either. Otherwise, consider the DFS path from some free $a \in A$ to $b$. This *is* a path that we can directly augment on, contradicting the assumption that we need to expand $\tilde{E}$.

By definition, each $a \in A\backslash$ or $b \in B \cap X$ are incident to some edges in $\tilde{E}$. However, there is one more thing to notice:

- If $M^*$ is the maximum bipartite matching on $\tilde{E}$, then for each $a \in A \backslash X$ and $b \in B \cap X$, the edge $(a,b) \notin M^*$. Otherwise, because $b$ is reachable from some free node $a_0 \in A$ and there is a backward edge $(b,a)$ in the residual graph, so is $a$, contradicting $a \in A \backslash X$.

Therefore, every node in $(A \backslash X) \cup (B \cap X)$ is a vertex of some matching, and different nodes in this union corresponds to different pairs. Since $M^*$ is maximal on $\tilde{E}$,

$$|A \backslash X| + |B \cap X| \leqslant |M^*|.$$

It remains to notice that $|M^*| < n$ since our algorithm would have terminated had a perfect matching been found. Therefore, (3) is positive, and in each iteration, the value of the dual objective strictly increases.

**Algorithm 6:** Hungarian Algorithm

```
1  Inputs: $s-t$ augmented bipartite graph $G$
2  Inputs: edge costs $c : E \to \mathbb{R}$, potential $\varphi : A \cup B \to \mathbb{R}$
3  Initialization: $\varphi(a_i) = 0$ for $a_i \in A$; $\varphi(b_j) = \min_i c(a_i, b_j)$ for $b_j \in B$
4  Initialization: $\tilde{c}(a_i, b_j) \leftarrow c(a_i, b_j) - \varphi(a_i) - \varphi(b_j)$ # nonnegative by construction
5  Initialization: $\tilde{E} = \varnothing$, the set of tight bipartite edges; $M = \varnothing$, empty matching
6  while $M$ is not a perfect matching do
7      compute residual graph $G_f$
8      check if an $s-t$ residual path exists
9      let $X$ be the set of non-$\{s,t\}$ visited vertices in this search process
10     if such path $P$ exists then
11         $M \leftarrow (M \backslash P) \cup (P \backslash M)$ # replace with forward edges. New $|M|$ increases by 1.
12     else
13         let $\delta \leftarrow \min_{\substack{a \in X \cap A \\ b \in B \backslash X}} \tilde{c}(a, b)$ # maximum possible dual increment along qualifying edges
14         update $\varphi$ by $\begin{cases} \varphi(a) \leftarrow \varphi(a) + \delta & \text{for all } a \in A \cap X \\ \varphi(b) \leftarrow \varphi(b) - \delta & \text{for all } b \in B \cap X \\ \text{no change otherwise.} \end{cases}$
15         update $\tilde{E}$ by $\tilde{E} \leftarrow \tilde{E} \cup \{e : \tilde{c}(e) = \delta\}$
16         update $M$ by $M \leftarrow$ maximum bipartite matching on $\tilde{E}$
17 return (maximal) bipartite matching $M$
```

**Remark.** Since the maximum bipartite matching $M$ contains $n$ edges, the `while` loop repeats $n$ times. Each iteration's runtime is dominated by the perfect matching procedure (updates are linear time), which can easily be done in $\mathcal{O}(mn)$. So overall the Hungarian algorithm can be run in $\mathcal{O}(mn^2)$. With a more efficient implementation, this bound can be further reduced to $\mathcal{O}(mn \log n)$.

Beginning of 09/23/2024

# 4 Approximation Algorithms

Many optimization problems that we encounter in practice are NP-hard and unlikely to be efficiently solvable. This has given rise to the area of approximation algorithms where the goal is to obtain approximately optimal solutions in polynomial time. In the following section, we will first consider two well-known NP-complete problems, VERTEX COVER and SET COVER, and show that some efficient polynomial algorithms, albeit not optimal, are "not too far" from optimality.
In general, for problems that we cannot provide efficiently provide an optimal solution, we resort to designing an **approximation algorithm** as a workaround. Instead of demanding optimality, we require a worst-case quality assurance:

**Definition**

For a minimization [resp. maximization] problem, an algorithm ALG is called an **$\alpha$-approximation**, $\alpha \geqslant 1$ [resp. $\alpha \leqslant 1$], if for all input instances $I$, $\text{ALG}(I) \leqslant \alpha \cdot \text{OPT}(I)$ [resp. $\geqslant$].

For example, a 2-approximation algorithm for a minimization problem would output a value that is no larger than twice the optimal value. The closer our approximation factor is to 1, the closer to optimal it is. With these definitions, we briefly categorize algorithms to the following classes:

- Exact algorithms. These are algorithms that solve optimization problems to optimality (obtains exact maximizer / minimizer).
- **PTAS** (*polynomial time approximation schemes*). An algorithm that provides a solution within a factor $1 \pm \epsilon$ of being optimal given any $\epsilon > 0$, but as $\epsilon \to 0$ the running time blows up.
- Constant approximation $\mathcal{O}(1)$. An algorithm that does not yield optimal solution, but the quality of approximation does not degrade as input size increases. Value of $\alpha$ bounded from above.
- Superconstant, e.g. $\mathcal{O}(\log N)$, $\mathcal{O}(N^c)$, etc. The quality of approximation decays as input size $N$ increases.

## 4.1 Introduction: SET COVER and VERTEX COVER

### SET COVER

First consider the SET COVER problem:

Given $G = (V, E)$, find a $S \subset V$ of minimal cardinality such that for all $e = (u, v) \in E$, $|\{u, v\} \cap S| \geqslant 1$.

To solve this problem, we can model it using LP:

$$\min_{v \in V} x_v \qquad \text{subject to} \qquad \begin{cases} x_u + x_v \geqslant 1 \text{ for each edge } (u,v) \in E \\ x_v \in \{0,1\} \text{ for each } v \in V. \end{cases}$$

However, a problem immediately arises: the feasible set is not convex! Consider the simplest graph of one edge and two vertices $u, v$. We can set either one to be $1$ and the other to be $0$, but taking the average, $x_u = x_v = 0.5$, no longer a feasible solution. The fix? We deviate from the original problem, relaxing it into *fractional* values for $x_v$:

$$\min_{v\in V} x_v \qquad \text{subject to} \qquad \begin{cases} x_u + x_v \geqslant 1 \text{ for each edge } (u,v) \in E \\ x_v \in [0,1] \text{ for each } v \in V. \end{cases} \tag{4}$$

Notice that we can still assume $x_v \in [0,1]$ since if $x_v > 1$, we can simply set $x_v$ to be $1$ to further decrease the objective function. We will assume the fact that there are polynomial-time algorithms that solve these type of LP.

A few problems nevertheless persist. *Firstly, how do we define an approximation factor?* Now we have three quantities: the output of our algorithm, the output of the optimal solution to VERTEX COVER, and the output of the optimal solution to LP. Let us call these $\mathrm{ALG}, \mathrm{OPT}(\mathrm{VC})$, and $\mathrm{OPT}(\mathrm{LP})$, respectively. Immediately we have the following inequality:

$$\mathrm{ALG} \geqslant \mathrm{OPT}(\mathrm{VC}) \geqslant \mathrm{OPT}(\mathrm{LP}) \tag{5}$$

where the first $\geqslant$ follows from the optimality of $\mathrm{OPT}(\mathrm{VC})$ on VERTEX COVER, and the second $\geqslant$ follows from the fact that the LP has less constraints than VERTEX COVER. Therefore we can consider the ratio between $\mathrm{ALG}$ and $\mathrm{OPT}(\mathrm{LP})$ and define an approximation factor out of it.

*Second problem: how do we recover an* $\mathrm{ALG}$ *from* $\mathrm{OPT}(\mathrm{LP})$ *that makes sense?* The answer here is via **rounding**: given our optimal fractional solution, we want to recover an integer solution from which we can extract a valid vertex cover. And most naturally we round our fractional values to $0$ and $1$ with a threshold of $0.5$: if $x_v \geqslant 0.5$ we round it to $1$, and $0$ otherwise.

***Proof that rounding is a 2-approximation***. We first observe that rounding yields a valid vertex cover: for any edge $e = (u,v)$, if $x_u + x_v = 0$ post-rounding, then it means $x_u, x_v$ are both $< 0.5$ pre-rounding, contradicting the constraints in (4). Therefore each edge has at least one value of $1$ after rounding, i.e., this is a valid vertex cover. To see this algorithm has an approximation factor of $2$, we note that the values assigned to each edge is at most doubled ($0.5$ to $1$). Therefore the objective is at most doubled. □

### 4.1.1 Integrality Gaps

One may naturally wonder if we can achieve a better approximation factor on SET COVER. Here, we want to compare $\mathrm{ALG}$ against $\mathrm{OPT}(\mathrm{LP})$. But notice that we have a chain of inequalities, where the second one $\mathrm{OPT}(\mathrm{VC}) \geqslant \mathrm{OPT}(\mathrm{LP})$ *has nothing to do with what algorithm we choose* — it is a property inherent to the nature of the VERTEX COVER problem itself. Imagine a problem whose optimal integer LP is doing significantly worse than fractional LP. Then, regardless of how good our proposed $\mathrm{ALG}$ is, the approximation factor cannot surpass the ratio $\mathrm{OPT}(\text{integer LP})/\mathrm{OPT}(\text{fractional LP})$. This quantity is called the **integrality gap** of the problem.

So what is the integrality gap of VERTEX COVER?
Recall that our objective is to minimize $\sum_v x_v$ subject to $x_u + x_v \geqslant 1$ for each edge $(u,v)$. An easy example to show the discrepancy between fractional and rounded solution is by considering a triangle. Clearly, an integer solution would need to pick $2$ out of the $3$ vertices, hence achieving an objective value of $2$. On the other hand, the optimal fractional solution assigns $1/2$ to each vertex, resulting in an objective value of $3/2$. Therefore, for this specific graph, the gap is $2/(3/2) = 4/3$.
More generally, consider a complete graph $K_n$ (i.e. vertices are pairwise directly connected). The integral solution needs to choose $n-1$ vertices, but the fractional solution can again assign $1/2$ to each vertex. Here, the gap becomes

$(n-1)/(n/2) = 2(1-1/n)$. Since $n$ is arbitrary, we see that the integrality gap for the problem must be at least 2. In other words,

> It is impossible for any rounding algorithm on VERTEX COVER to achieve an approximation factor better than 2.

And since we showed that threshold rounding indeed achieves an approximation of 2, we conclude that **the integrality gap of VERTEX COVER is 2**.

## SET COVER[10]

> Given a universal set $U$ of $n$ elements and $m$ subsets $S_1, \cdots, S_m \subset U$ with $\bigcup_i S_i = U$, find a minimum number of subsets that cover $U$.

Many approximation algorithms exist for SET COVER, but we approach this problem using a naïve greedy algorithm: always pick the set that covers as many remaining elements as possible. We show that this algorithm has an approximation factor of $\log n$.

***Greedy* SET COVER *algorithm is* $\log n$*-approximate*.** Let $S_1, S_2, \cdots$ be the sequence of sets picked by our greedy algorithm, ALG. Say $S_1$ covers $x_1$ elements, and for each $i > 1$, $S_i$ covers $i$ *additional* (uncovered) elements. Consider the optimal solution OPT which consists of, with the abuse of notations, OPT subsets of $U$. The key observation is that:

- $\underline{x_1 \geqslant n/\text{OPT}.}$ To see this, note that $|U| = n$, so by pigeonhole the largest set has $\geqslant n/\text{OPT}$ elements.
- By the same token, $x_2 \geqslant (n - x_1)/\text{OPT}$ since all sets in OPT must still collectively cover the remaining $n - x_1$ elements. Thus, the best remaining set performs at least as good as the average.
- Iteratively, $x_i \geqslant (n - \sum_{j<i} x_j)/\text{OPT}$.

Rearranging the last inequality, we obtain (abusing the notation that ALG also means the cardinality of the outputted set)

$$1 \leqslant \text{OPT} \cdot \frac{x_i}{\sum_{j \geqslant i} x_j} \qquad \text{for each } i \in [\text{ALG}]$$

[10]The lecture used weighted SET COVER where each $S_i$ also has a weight $w(S_i)$. If this is the case, simply greedily pick the set with the smallest value of $w(S_i)/|S_i \backslash \{\text{elements already covered}\}|$ and add new elements covered by $S_i$ to the set of covered elements. The proof is similar, and the resulting approximation factor is identical.

Summing over $i$ we see

$$\begin{aligned}\text{ALG} &\leqslant \text{OPT} \cdot \sum_{i=1}^{\text{ALG}} \frac{x_i}{\sum_{j \geqslant i} x_j} \\ &= \text{OPT} \cdot \left[ \frac{x_1}{n} + \frac{x_2}{n - x_1} + \frac{x_3}{n - (x_1 + x_2)} + \cdots \right] \\ &\leqslant \text{OPT} \cdot \left[ \underbrace{\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{n - x_1 + 1}}_{n \text{ total}} \right. \\ &\qquad + \underbrace{\frac{1}{n - x_1} + \frac{1}{n - x_1 - 1} + \cdots + \frac{1}{n - (x_1 + x_2) + 1}}_{x_2 \text{ total}} \\ &\qquad \left. + (x_3) \text{ total decomposed terms for } \frac{x_3}{n - (x_1 + x_2)} + \cdots \right] \\ &\leqslant \text{OPT} \cdot (1 + 1/2 + \cdots + 1/n) = \text{OPT} \cdot \Theta(\log n).\end{aligned}$$

□

**Integrality Gap of Set Cover**[11]

Recall that our objective is to minimize $\sum_i x_i$ such that $\sum_{i: j \in S_i} x_i \geqslant 1$ for each $j$ in the universal set $U$. Consider the following instance of set cover. Say we have $m$ sets, $S_1, \cdots, S_m$. For every collection $X$ of $m/2$ sets, define an element $x$ that only belongs to these $m/2$ sets. By Sterling's formula, there are roughly

$$\binom{m}{m/2} \approx \frac{2^m}{\sqrt{\pi m}}$$

elements. By the pigeonhole principle, the optimal integral solution must contain $m/2 + 1$ elements, but it suffices to assign $2/m$ to each element in a fractional solution, resulting in a total weight of $2/m \cdot m = 2$. Therefore, the gap is $m/4$ which, by taking the log of $\binom{m}{m/2}$, becomes $\Theta(\log n)$. Therefore, **rounding-based Set Cover LP algorithms with $n$ elements cannot get better than a factor of $\Theta(\log n)$.**

Beginning of 09/25/2024

## 4.2 Clustering

In this section, we consider a few dry but useful clustering algorithms, in particular, center-based clustering. The problem setup is as follows: given a finite metric space $(X, d)$ consisting of points $p_1, \cdots, p_n$, a **clustering** is a partition $S_1, \cdots, S_k$ of the points. For each $S_i$, we also want to assign the **center** to be $c_i$ that minimizes *some* objective function w.r.t. $S$, the set of clusters, and $C$, the set of centers. Below are some common objective functions.

$$\varphi(S, C) = \max_{i \in [k]} \max_{p \in S_i} d(p, c_i) \qquad (\textbf{\textit{k}-center})$$

$$\varphi(S, C) = \frac{1}{n} \sum_{i=1}^{k} \sum_{p \in S_i} d(p, c_i) \qquad (\textbf{\textit{k}-median})$$

$$\varphi(S, C) = \left( \frac{1}{n} \sum_{i=1}^{k} \sum_{p \in S_i} d(p, c_i)^2 \right)^{1/2} \quad \text{and} \quad \varphi_q(S, C) = \left( \frac{1}{n} \sum_{i=1}^{k} \sum_{p \in S_i} d(p, c_i)^q \right)^{1/q} \qquad (\text{(generalized) } \textbf{\textit{k}-mean})$$

[11] Not covered, but included for the sake of completeness.

**Remark.** Before going anywhere further, let us state an almost trivial but important fact that will appear many times in the following proofs: given **fixed** set of centers $c_1, \cdots, c_k$, the assignment that sends each point to its nearest $c_i$ minimizes all objectives.

Back to greedy algorithms we go. Let us consider the $k$-center problem first: we need $k$ centers by the end, but initially we have none. So we perform $k$ iterations, adding one cluster at a time. The objective of $k$-center is to minimize $\varphi(S, C) = \max_{i\in[k]} \max_{p\in S_i} d(p, c_i)$, so the intuition is that if we have yet to introduce a new center or cluster, we better assign the currently most isolated point its own cluster. This gives rise to the following algorithm:

**Algorithm 7:** Greedy $k$-center

```
1 Initialization: $C = \varnothing$, set of centers
2 Initialization: distance function $\mathrm{dist}[p] = \infty$ for all points $p$
3 for iteration $1, \cdots, k$ do
4     $p^* \leftarrow \mathrm{argmax}_p \mathrm{dist}[p]$, break ties arbitrarily # furthest point
5     update $C \leftarrow C \cup \{p^*\}$
6     for each point $p$ do
7         $\mathrm{dist}[p] \leftarrow \min(\mathrm{dist}[p], d(p, p^*))$ # update distance if needed
8         if distance updated: associate $p$ with the new cluster
9 return centers $C$, and (optionally) clusters
```

As discussed above, this algorithm intuitively makes sense. The following proof establishes that it is indeed 2-approximate.

***Proof that greedy k-center is 2-approximate.*** Suppose $c_1^*, \cdots, c_k^*$ is a set of optimal centers for the $k$-center problem, and the objective value is $r^*$. That is, the union of balls $B(c_i^*, r^*)$ [centered at $c_i^*$, radius $r_i^*$] cover all points. Now consider any point $p$. Clearly, $p$ is contained in one such balls. Call this ball $B_p$ and call its center $c_p^*$.
Let $c_1, \cdots, c_k$ be the set of centers outputted by the greedy $k$-center. Two cases:

(1) $B_p$ contains a greedy center. According to the first remark, we may WLOG assume that this greedy center, say $c_p$, is also the closest one to $p$, and that in the greedy assignment, $p$ belongs to the cluster represented by $c_p$. It follows by triangle inequality that

$$d(p, c_p) \leqslant d(p, c_p^*) + d(c_p^*, c_p) \leqslant 2r^*.$$

(2) What if $B_p$ does not contain a greedy center? By pigeonhole, since we have $k$ greedy centers and $k$ optimal balls, one ball $B'$ with center contains at least two greedy centers. One of them, call it $c'$, was added first. Call the other one $c''$. What caused the algorithm to choose $c''$ over $p$? Well, it means in that iteration, $\mathrm{dist}[c''] > \mathrm{dist}[p]$. On the other hand, we know that since both greedy centers $c', c''$ are within $r^*$ from the center of $B'$, so $d(c', c'') \leqslant 2r^*$. Therefore, either $\mathrm{dist}[p] \leqslant 2r^*$ or the algorithm would have chosen $p$ over $c''$. This concludes the proof that greedy $k$-center is 2-approximate. □

Next up, we consider $k$-means clustering. In addition to the remark above, we also note that if we have partitioned the points into $S_1, \cdots, S_k$, then choosing each $c_i$ to be the centroid of $S_i$ would ensure that the objective is minimized.

**Lloyd's algorithm** [2006] simply abuses these two observation, as shown below.

**Algorithm 8:** Lloyd's Algorithm

```
1 Initialization: randomly assign all points to a centroid ($k$ total)
2 Initialization: calculate the centroid $c_i$ of each cluster
3 while not converged do
4     reassign each point to their closest centroid $c_i$
5     recalculate and update centroids
6 return centers, paritition
```

The convergence criterion for Lloyd's algorithm (also known as the vanilla $k$-means algorithm) is that no centroid got updated — the partition of points into clusters is now stable, so there is no need to update centroids. Unfortunately, Lloyd's algorithm is heuristic and greatly depends on the random initialiation, where a bad initialization may fail to escape a local optimum. To this end, we consider a probabilistic approach called $k$-means++ [2007] to improve the algorithm, which was proposed one year after Lloyd's algorithm.

**Algorithm 9:** $k$-means++

```
1 Initialization: choose $p_1$ uniformly at random from points
2 $C \leftarrow \{p_1\}$, and $\text{dist}[p] = d(p, p_1)$ for all $p$
3 for remaining centers in iteration $2, \cdots, k$ do
4     sample $p^*$ from $S$ from distribution $\mathbb{P}(p \text{ chosen}) = \text{dist}[p]^2 / \sum_{\tilde{p}} \text{dist}[\tilde{p}]^2$
5     $C \leftarrow C \cup \{p^*\}$
6     for each point $p$ do
7         $\text{dist}[p] = \min(\text{dist}[p], d(p, p^*))$, update cluster assignment if needed
8 return centers, partition
```

**Remark.** This algorithm is $\mathcal{O}(\log k)$-approximate, a significant improvement over the vanilla $k$-means, which does not provide any guarantee.

A variant [2019] of $k$-means++ manages to theoretically reduce the $\mathcal{O}(\log k)$ approximation factor of $k$-means++ to constant via **local search**. The following is the idea for local search:

- Draw point $p$ among all points with probability $\text{dist}[p]^2 / \sum_{\tilde{p}} \text{dist}[\tilde{p}]^2$ (just like in $k$-means++),
- Let $q = \arg\min_{q'} \varphi(C \cup \{p\} \backslash \{q'\})$,
- If $\varphi(C \cup \{p\} \backslash \{q\}) < \varphi(C)$, update $C$.

Basically, we are interested in potentially swapping some current center for another point that reduces the objective value. The authors showed that $k$-means++ followed by $\epsilon \cdot k$ calls of local search will, in expectation, bound the objective by $\mathcal{O}(\epsilon^{-3}) \cdot \text{OPT}$. A constant approximation factor for sure, but very weak.

## 4.3 Scheduling

In this section, we consider the scheduling of jobs. Suppose we have $n$ jobs, each having a processing time $p_i$ to be processed. Suppose we have $m < n$ machines that we can run the jobs on. Finally, use $\Sigma$ to denote a schedule, and with respect to this $\Sigma$, each job has a completion time $c_i$ at which it is finished. The **makespan** of our schedule $\Sigma$ is the longest completion time $C_{\max}(\Sigma) = \max_i c_i$. Our goal is to compute a schedule $\Sigma^*$ with the minimal makespan.

This naturally gives rise to a greedy algorithm:

**Algorithm 10:** Greedy Makespan

```
1 Initialization: $S_i$ = subset of jobs assigned to machine $i$, initially empty, $i \in [m]$
2 Initialization: $L_i$ = load of machine $i$, initially $= 0$
3 for $j = 1, \cdots, n$ do
4     $k = \arg\min_{1 \leqslant i \leqslant m} L_i$
5     $S_K \leftarrow S_k \cup \{j\}$, and $L_k \leftarrow L_k - p_j$
6 return assignment $\Sigma = (S_1, \cdots, S_m)$
```

Intuitively, for each job, we simply assign it to the machine with the current lowest load, hoping to minimize the end time of this job. But we have a problem: we never looked at the actual duration of the job, which could certain be a setback in some situations.
Let $\ell$ be the last job to be schedule by this algorithm, then it is immediately clear that $C_{\max} = S_\ell + p_\ell$, where $S_\ell$ is the starting time of job $\ell$. The following claim helps us to establish that this greedy algorithm is 2-approximate:

**Claim**. All machines are busy at time $S_\ell$, i.e., when job $\ell$ starts, i.e., $S_\ell \leqslant L_i$ for all machine $i$.
Next, observe that the sum of processing times $P$ satisfies

$$P = \sum_{j=1}^{n} p_j = \sum_{i=1}^{m} L_i \geqslant m \cdot S_\ell.$$

In the optimal schedule, each job apparently has to be processed by some machine, so on the other hand we have

$$C^*_{\max} \geqslant \frac{P}{m} \implies P \leqslant m \cdot C^*_{\max}.$$

Combining the two inequalities, we see $S_\ell \leqslant C^*_{\max}$, and so for our greedy solution satisfies

$$C_{\max} = S_\ell + p_\ell \leqslant C^*_{\max} + \max_j p_j \leqslant C^*_{\max} + C^*_{\max} = 2C^*_{\max}.$$

This goes to show that greedy makespan achieves an approximation factor of 2.

We now consider another greedy algorithm, *longest job first*. This aims to address the problem of not considering job length when scheduling in the greedy algorithm. To sum up:

```
Algorithm 11: Makespan: Longest Job First
1 Initialization: sort jobs in non-decreasing order of processing time $p_i$
2 Run greedy list scheduling
```

*Proof that longest-job-first is* $4/3$*-approximate.* As usual, let $\ell$ be the last jobs scheduled by longest-job-first. First, it is immediately clear that if $p_\ell \leqslant C^*_{\max}/3$ then $C_{\max} = p_\ell + S_\ell \leqslant C^*_{\max}/3 + C^*_{\max} \leqslant 4/3 \cdot C^*_{\max}$.
So let us assume $p_\ell > C^*_{\max}/3$. Since $p_\ell$ is the last job to finish, jobs with larger indices (after sorting) $p_{\ell+1}, \cdots, p_n$ do not affect the value of $C_{\max}$ (they end earlier). Therefore we can assume that $p_\ell = p_n$. But then each $p_i > C^*_{\max}/3$, so each machine handles at most 2 jobs, and the greedy algorithm finds an optimal schedule, implying $C_{\max} = C^*_{\max}$. Either way, the algorithm is bounded by a $4/3$ approximation factor. □

### 4.3.1 A $(1+\epsilon)$-Approximation Algorithm

The next step is to upgrade our approximation: for any $\epsilon > 0$, we want to find a schedule with $C_{\max}(\Sigma) \leqslant (1+\epsilon)C^*_{\max}$. To simplify notations, we use $k = 1/\epsilon$ and $P = \sum_j p_j$ the total processing time, like above. We say a job $j$ is **long** if its processing time $p_j > p/(km)$ and **short** otherwise. This implies that the number of long jobs is less than $km$.

Our first step is to find an optimal schedule for the long jobs using an exhaustive search. Then, we run the list scheduling algorithm for short jobs and obtain the resulting $\hat{\Sigma}$ overal schedule.

Let $\ell$ once gain be the last job to be processed by the overall schedule $\hat{\Sigma}$. Then:

- if $\ell$ is a long job, $C_{\max}(\hat{\Sigma}) = C^*_{\max}$, because all long jobs are processed, and
- if $\ell$ is a short job, then

$$C_{\max}(\hat{\Sigma}) \leqslant S_\ell + p_\ell \leqslant C^*_{\max} + \frac{1}{k}\frac{p}{m} \leqslant C^*_{\max} + \epsilon C^*_{\max} = (1+\epsilon)C^*_{\max}.$$

This shows that we have an $(1+\epsilon)$-approximate algorithm. But what about runtime? The total runtime would be $\mathcal{O}(m^{km})$. If $m$ is $\mathcal{O}(1)$, then this reduces to $\mathcal{O}(2^{\epsilon^{-1} m \log m})$. But what if $m$ is large (i.e. not $\mathcal{O}(1)$)?

- Use greedy list scheduling to find a paretmer $T_0 \in [C^*_{\max}, 2C^*_{\max}]$ as a starting point. We iteratively maintain an interval $[L, U]$ to perform binary search on, subject to $L \leqslant C^*_{\max}$. Initially let $L = T_0/2$ and $U = T_0$.
- Given a threshold $T$, design an algorithm that returns a schedule $\Sigma$ with $C_{\max}(\Sigma) \leqslant (1+\epsilon)T$, if any.

A high-level pseudocode is provided below:

```
Algorithm 12: $(1+\epsilon)$-Approximate Scheduling
1 Initialization: run greedy scheduling to find $T_0 \in [C^*_{\max}, 2C^*_{\max}]$
2 Initialization: binary search bounds $L = T_0/2$, $U = T_0$
3 while $L < U$ do
4     assume $T \geqslant P/m$
5     for each $j$, define $j$ long if $p_j > T/k$ (where $k = \epsilon^{-1}$) and short otherwise
6     round each long job down to be a multiple of $T/k^2$: $\tilde{p}_j \leftarrow \left\lfloor \frac{p_j}{T/k^2} \right\rfloor T/k^2$
7     find a schedule makespan $\leqslant T$ for rounded long jobs
8     use greedy scheduling to assign short jobs
```

Time to analyze this algorithm. First, let $S_i$ be the set of rounded *long* jobs assigned to machien $i$. Immediately we see $|S_i| \leqslant k$, since each $p_j > T/k$. Therefore, for each machine $j$,

$$\sum_{j \in S_i} p_j \leqslant \sum_{j \in S_i} (\tilde{p}_j + Tk^{-2}) \leqslant \sum_{j \in S_i} \tilde{p}_j + \frac{T}{k} \leqslant T + \frac{T}{k} = (1 + k^{-1})T.$$

This explains the seeming arbitrary round-down factor $T/k^2$ in the algorithm. Therefore, if the last job $p_\ell$ is a long job, we are done with the proof.

Now what if job $\ell$ is short? As above, $S_\ell \leqslant p/m \leqslant T$, so

$$C_{\max}(\Sigma) = S_\ell + p_\ell \leqslant T + \frac{T}{k} = (1 + k^{-1})T$$

completing the proof of $(1 + \epsilon)$-approximation once again.

## 4.4 Multiplicative Weight Method

In this section, we consider two **multiplicative weight method** (MWU) based solutions to two problems: *hitting set* and *approximate linear programming*. Let us begin with hitting set, whose algorithm is simpler but requires many definitions from the Vapnik–Chervonenkis (VC) theory, which is crucial in computation learning theory.

### The Hitting Set Problem

Consider $\Sigma = (X, \mathcal{R})$, a *finite range space*, where $X$ is a (finite) set, and $\mathcal{R} \subset 2^X$ is a collection of subsets of $X$. We call $\mathcal{R}$ **ranges** or **hyperedges**. A subset $H \subset X$ is called a **hitting set** if $H \cap R \neq \varnothing$ for all $R \in \mathcal{R}$: think of this as $H$ does not miss a single range $R \in \mathcal{R}$, essentially "hitting" all of them.

A subset $Y \subset X$ is is said to be **shattered** by $\mathcal{R}$ if $\{R \cap Y : R \in \mathcal{R}\}$ gets all subsets of $Y$. Finally, the **VC dimension** of $\Sigma$, written $\mathrm{VC}\dim(\Sigma)$, is the size of the largest subset that can be shattered. If we can find arbitrarily large subsets that can be shttered, then $\mathrm{VC}\dim(\Sigma) = \infty$.

Let us first consider a quick example. Consider $\Sigma = (X, \mathcal{R})$ where $X = \mathbb{R}$ and $\mathcal{R} = \{X \cap I : I \text{ is an interval}\}$, i.e., the collection of real-valued intervals. What is the VC dimension of $\mathbb{R}$? Well, for any set $Y = \{a, b\}$ consisting of two points, it is shattered by $\mathcal{R}$, since $\{a, b\} \cap [a + \epsilon, b - \epsilon] = \varnothing$, $\{a, b\} \cap [a - \epsilon, a + \epsilon] = \{a\}$, $\{a, b\} \cap [b - \epsilon, b + \epsilon] = \{b\}$, and finally, $\{a, b\} \cap [a - \epsilon, b + \epsilon] = \{a, b\}$. In other words, for any subset $Y' \subset Y$, we can find $R \in \mathcal{R}$ such that $Y \cap R = Y'$. So any set of size 2 can be shattered. On the other hand, if $Y$ consists of three elements $\{a, b, c\}$ with $a < b < c$, there is no interval whose intersection with $Y$ is precisely $\{a, c\}$ without including $b$. Hence $\mathrm{VC}\dim(\Sigma) = 2$.

Immediately, we see that if $\mathrm{VC}\dim(\Sigma) = d$, then $|\mathcal{R}| = \mathcal{O}(n^d)$ where $n$ is the size of $X$. This is particularly useful for boolean functions: classifiers are learnable if the VC dimension is bounded. But we will not go into depth.

A weaker condition than being a hitting set is defined the notion of $\epsilon$-nets. In addition to the set system $\Sigma = (X, \mathcal{R})$, we introduce a weight function defined on elements in $X$, and also on ranges: $w(R) = \sum_{x \in R} w(x)$ for each $R \in \mathcal{R}$.

A subset $N \subset X$ is called an **$\epsilon$-net** if $N \cap R \neq \varnothing$ for all sets $R$ with weight $w(R) \geqslant \epsilon w(X)$. Essentially, this hitting set demands the intersection to be nonempty for any range $R$, whereas an $\epsilon$-net only demands such nonempty intersection from **heavy** sets (the ones with $w(R) \geqslant \epsilon w(X)$). It is known that

**Theorem**

Given a set system $\Sigma = (X, \mathcal{R})$ with $\mathrm{VC}\dim(\Sigma) = d$, a random subset of size $\mathcal{O}(d\epsilon^{-1}\log(\epsilon^{-1}))$ will be a $\epsilon$-net with probability $\geqslant 1/2$.

Below, we consider a MWU-based algorithm for computing an $\epsilon$-net / hitting set of $\Sigma$. We say a range/hyperedge in $\Sigma$ is **$\epsilon$-light** if it's not $\epsilon$-heavy. The weight update is simple — we just double the weight of some elements.
Let us assume that $k$, the size of the optimal hitting set of $\Sigma$, is given. (Else we can do a binary search to find $k$.) We initialize $\epsilon = \log\sqrt{2}/k$ and give all elements a weight of $w(x) = 1$. Then, every time we find an $\epsilon$-light set, we just double the weight of everything inside. This is summarized in the algorithm below.

**Algorithm 13:** MWU Hitting Set

```
1 Initialization: $w(x) = 1$ for all $x \in X$; $\epsilon = \log\sqrt{2}/k$.
2 while there exists an $\epsilon$-light hyperedge/range do
3     let $R$ be one such range
4     $w(x) \leftarrow 2w(x)$ for all $x \in R$
```

It is not clear whether this algorithm would terminate, and if it does, after how many iterations. To analyze this, let $w_i$ be the weight of $w(X)$ (entire set) after $i$ iterations. Clearly, $w_0 = |X| = n$ initially. A light range has weight at most $\epsilon^{-1}$ of total weight, so

$$w_{i+1} \leqslant w_i + \epsilon w_i = (1+\epsilon)w_i,$$

which implies

$$w_i \leqslant (1+\epsilon)^i w_0 \sim \exp(i\epsilon)n.$$

So that is an upper bound. How about a lower bound? We look at a hitting set $H$ of size $k$. $H \cap R$ is nonempty, so in each iteration, everything in $H \cap R$ is doubled. The worst case is round robin, meaning that in each iteration, one different element gets doubled, where it takes $k$ iterations to get everything to 2, and another $k$ iterations to 4, and so on. Therefore, a crude lower bound can be provided by $w(H) \geqslant k2^{(i/k)}$, which implies

$$ik^{-1}\log 2 - ik^{-1}\log\sqrt{2} \leqslant \log(n/k)$$

and so $i = \mathcal{O}(k\log(n/k))$. That is, after these many iterations, we obtain an $\epsilon$-net. Overall, $|H| = \mathcal{O}(\mathrm{OPT}\log\mathrm{OPT})$.

## Approximate Linear Programming

In this section, we consider **approximate linear programming**. Instead of solving $Ax \geqslant b$, we ask the following relaxed LP:

(Approximate LP) Given $\epsilon$, does there exist a solution satisfying $Ax \geqslant b - \epsilon$?

(Note that if $x$ is feasible for the original LP, then it is certainly feasible for this approximate LP.)
To approach this, we will use an **oracle** $O$, that applies a probability distribution $p$ to constraints. In more details, the oracle transforms the approximate LP into a probabilistic inequality and answers the following question:

$$\boxed{\text{(Oracle) Given a distribution } p\text{, does there exist a solution satisfying } p^T Ax \geqslant p^T b\text{?}}$$

In addition, if the distribution returns a YES, we are also interested in the "width" of the oracle, or a bound on the error of $Ax - b$: find $\rho > 0$ such that $Ax - b \in [-\rho, \rho]^n$.

The following briefly describes an algorithm that uses the oracle to solve the approximate LP.

**Algorithm 14:** MWU Approximate LP

```
1 Initialization: $\eta = \epsilon/(4\rho), T = 8\rho^2\epsilon^{-2}\log m$
2 Initialization: $w_i^{(t)}$ = weight of $i^{\text{th}}$ constraint after iteration $t$; $w_i^{(0)} = 1$
3 Initialization: $\Phi^{(0)} = \sum_{i=1}^m w_i^{(0)}$, $p_i^{(0)} = w_i^{(0)}/\Phi^{(0)}$ (normalized probabilities)
4 for $t = 1, \cdots, T$ do
5     $x^{(t)} = \text{Oracle}(p^{(t-1)T}Ax \geqslant p^{(t-1)T}b)$
6     if Oracle returns infeasible, return infeasible
7     $m_i^{(t)} \leftarrow (A_i x^{(t)} - b_i)/\rho$
8     $w_i^{(t+1)} \leftarrow w_i^{(t)}(1 - \eta m_i^{(t)})$
9 return $\overline{x} = \sum_{t=1}^T x^{(t)}$
```

**Proposition**

If the algorithm above doesn't return infeasible, then the output $\overline{x}$ satisfies approximate LP, i.e., $A_i\overline{x} \geqslant b_i - \epsilon$ for each coordinate $i$.

In addition, we can prove a stronger claim: the "slcakness" of constraints are not too far from their expectation. Put in formulas, this means for any $i \leqslant m$,

$$\sum_{t=1}^T \underbrace{m^{(t)} \cdot p^{(t)}}_{\text{expected slackness}} \leqslant \sum_{t=1}^T \underbrace{m_i^{(t)}}_{i^{\text{th}}\text{ slackness}} + \eta \sum_{t=1}^T |m_i^{(t)}| + \frac{\log m}{\eta}.$$

Using this additional claim, by the Oracle feasibility, for each $t$, $p^{(t)T}Ax^{(t)} = p^{(t)T}b \geqslant 0$. Therefore,

$$\begin{aligned}
0 &\leqslant \text{LHS}/T \leqslant \text{RHS}/T \\
&\leqslant \frac{1}{T}\sum_{t=1}^T (A_i x^{(t)} - b_i) + \frac{\eta}{T}\sum_{i=1}^T |A_i x^{(t)} - b_i| + \frac{\eta}{T}\frac{\log m}{\eta} \\
&\leqslant A_i\overline{x} - b_i + \frac{\eta}{T}\eta T + \frac{\eta}{T}\frac{\log m}{n} \\
&\leqslant A_i\overline{x} - b_i + \epsilon + \frac{\epsilon}{2} = A_i\overline{x} - b_i + 3\epsilon/4
\end{aligned}$$

Which goes to show $A_i x \geqslant b_i - \epsilon$, completing the claim.

## 5 The Primal Dual Method

In this section we consider two applications of the primal dual method. We have previously (implicitly) used it in deriving the Hungarian algorithm. In essence, for many LP problems, it is helpful to keep track of its dual. Instead of modifying the primal LP variables, we initialize the variables in the dual LP, then slowly modify (usually increase) the values until some dual constraint becomes tight. We then look back at the primal and interpret what it means for the dual constraint to become tight. Since dual constraints correspond to primal variables, usually when a dual constraint becomes tight, we know that the primal variable has been taken care of. This procedure is called **dual fitting**.

## 5.1 Feedback Vertex Set

Given an undirected graph $G = (V, E)$ with nonnegative weights $w_e$, a vertex set $F \subset V$ is said to be a **feedback vertex set, FVS**, if every single cycle contains a vertex in $F$. An equivalent definition is that $G[V\backslash F]$ (the subgraph induced by $V\backslash F$) is acyclic. Our goal is to find the minimum weight FVS.
Let is first formulate the primal and dual LP:

$$\min_v w_v x_v \qquad \text{subject to} \qquad \begin{cases} \sum_{i\in C} x_i \geqslant 1 & \text{for all cycle } C \\ x_i \geqslant 0. \end{cases} \tag{FVS Primal}$$

$$\max \sum_{\text{cycles } C} \pi_C \qquad \text{subject to} \qquad \begin{cases} \sum_{C: v\in C} \pi_C \leqslant w_v & \text{for all vertices } v \\ \pi_C \geqslant 0. \end{cases} \tag{FVS Dual}$$

Now let us apply the general principle of dual fitting to the FVS problem. Our dual variables correspond to cycles, but notice we need not (and cannot, since there can be exponentially many cycles) maintain a variable for each cycle. Recall that we usually initialize every variable to $0$, so in our case, we can "cheat" by defining cycle variables $\pi_C$ only when they become nonzero. This bypasses the problem of having exponentially many cycles.
Each dual constraint is defined on a vertex, so correspondingly we can define the dual slackness $s_i = w_i - \sum_{i\in C} \pi_C$, which we will use to guide our dual fitting process.
If the complement graph $G[V\backslash F]$ is free of cycles, then the algorithms should terminate. Otherwise, there exists a cycle, and we need to add another vertex to $F$. Therefore we want to break a cycle, introduce a vertex to $F$, and update the dual variables and slackness accordingly, so that the newly introduced vertex now is not slack.
Summarizing everything into a high-level algorithm:

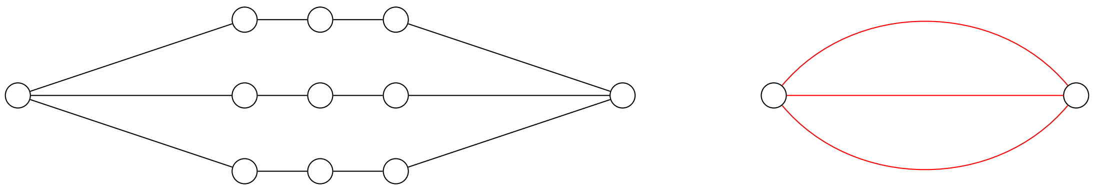
**Algorithm 15:** Feedback Vertex Set

```
1 Inputs: $G = (V, E)$ with weights $w_e \geqslant 0$
2 Initialization: $F = \varnothing$, $H = G[V\backslash F]$
3 Initialization: slackness $s_v = w_v - \sum_{v\in C} \pi_C$ for each vertex $v$
4 while there is a cycle $C$ in $H$ do
5     let $C$ be one such cycle
6     find vertex $j \in C$ with smallest slackness $s_j = \Delta$
7     update variables: $F \leftarrow F \cup \{j\}$, $\pi_C = \Delta, s_v \leftarrow s_v - \Delta$ for all $v \in C$
8 Return: $F$
```

Observe that every vertex $v \in F$ corresponds to a tight dual constraint, $\sum_{v\in C} \pi_C \leqslant w_v$. Therefore, using an exchange sum argument,

$$\text{total cost of } F = \sum_{v\in F} w_v = \sum_{v\in F} \sum_{v\in C} \pi_C = \sum_C |F \cap C| \pi_C.$$

If we can show that $|F \cap C| \leqslant \alpha$ for all $F$, then $F$ is $\alpha$-approximate. Unfortunately we cannot assert anything exciting about $|F \cap C|$. So we need some modifications to guide us to a stronger result.
Observe that if a node has degree $1$, then it is certainly not in any cycle, so we can simply remove these nodes. On the other hand, if a node has degree $2$ (or if a chain of nodes all have degree $2$), then they belong to the exact same cycle (if there is any). Therefore, in the perspective of cycles, all of them can be compressed into one single *edge*.



After this **contraction**, every node has degree $\geqslant 3$. It can be shown that such a graph will have cycles of length $\leqslant 2\log_2 n$. Therefore, $|F \cap C| = \mathcal{O}(\log n)$. **We just need to compute the contracted graph $H^*$ and replace "there is a cycle $C$ in $H$" with "in $H^*$."** The analysis above shows that we obtain a $\mathcal{O}(\log n)$ approximate algorithm.

## 5.2 The Facility Location Problem

Perhaps a more involved example of primal dual is the facility location problem: we have

- $D$, a set of demand points (customers),
- $F$, a set of facility that can be opened, and
- $f_i$, the cost of opening each facility $i$, $i \in F$, and
- $c_{i,j}$, the cost of serving customer $j$ via facility $i$.

The key distinction of this problem is that now the cost involves more terms. With these defined, the priaal and dual can be expressed as the following:

$$\min \left[ \sum_{i\in F} f_i y_i + \sum_{i\in F, j\in D} c_{i,j} x_{i,j} \right] \text{ subject to } \begin{cases} \sum_{i\in F} x_{i,j} \geqslant 1 & \text{for all customer } j \\ x_{i,j} \leqslant y_i & \text{for all } i,j \\ x_{i,j}, y_i \geqslant 0. \end{cases} \tag{Facility Primal}$$

(To interpret the second constraint: viewing as an integer LP instance, if $y_i = 0$, then $x_{i,j} = 0$, for if a facility is not opened, it cannot be used to serve customeres.) The dual is

$$\max \sum_{j\in D} v_j \left( + \sum_{i\in F, j\in D} 0 \cdot w_{i,j} \right) \text{ subject to } \begin{cases} \sum_{j\in D} w_{i,j} \leqslant f_i & \text{for each facility } i \\ v_i - w_{i,j} \leqslant c_{i,j} & \text{for all } i,j. \end{cases} \tag{Facility Dual}$$

Let us first describe the process of primal fitting for this problem. The dual variables are $v_j$ for each customer $j$, as well as $w_{i,j}$ for each (facility, customer) pair. Whenever a customer is not yet assigned a facility, we need to do something, namely, grow the $v_j$ as well as all $w_{i,j}$ that can still be grown. We can view $w_{i,j}$ as the "popularity" of facility $i$ coming from customer $j$, and $v_j$ is customer $j$'s "demand." The growing must stop when a constraint becomes tight. We have two cases: the first one is $v_i - w_{i,j} = c_{i,j}$, where we know that an assignment can now be made, since we have reached customer $j$'s limit. The second case is $\sum_j w_{i,j} = f_i$, where there is now enough popularity to overcome the facility opening cost, so we open facility $i$. Now, we summarize the intuition into a high-level algorithm. First, some notations:

- For $j \in D$, let $N(j)$ denote $\{\text{facilities} : v_i \geqslant c_{i,j}\}$, the set of facilities that can be used to server customer $j$;
- For $i \in F$, let $N(i)$ denote $\{\text{customer } j : i \in N(j)\}$, the potential customers;
- We say customer $j$ **contributes** to facility $i$ if $w_{i,j} \geqslant 0$;
- $T = \{i \in F : \sum_{j \in D} w_{i,j} = f_i\}$, the set of facilities that could be opened; and
- $X = \{j \in D : N(j) \cap T = \varnothing\}$, the set of customers not yet connected to any opened facility.

The algorithm goes as follows. There are a lot of symbols but the idea is not much more than what is described above.

**Algorithm 16:** Facility Opening

```
1 Inputs: facilities $F$, demands $D$, facility opening costs $f_i$, serving costs $w_{i,j}$
2 Initialization: $w_{i,j}, v_j = 0$
  $T = \varnothing, X = D, N(x) = \varnothing$ for all $x \in F$ and $x \in D$
  while $X \neq \varnothing$ do
      increase $v_j \in X$ and $w_{i,j} \in N(j)$ uniformly, for all $j \in X$
      stop until some new dual constraint becomes tight
      if CASE 1:$v_i - w_{i,j} = c_i, j$ for some $i, j$ then
          # $v_j$ cannot be increased anymore
          $N(j) \leftarrow N(j) \cup \{i\}$
          $N(i) \leftarrow N(i) \cup \{j\}$
          if $i \in T$ then $X \leftarrow X - \{j\}$ # $j$ now assigned
      else
          # CASE 2: $\sum_j w_{i,j} = f_i$ for some $i$
          open facility $i$, i.e., $T \leftarrow T \cup \{i\}$
          $X \leftarrow X \backslash N(i)$
  Post-processing: $T' = \varnothing$
  while $T \neq \varnothing$ do
      pick $i \in T$ and update $T' \leftarrow T' \cup \{i\}$
      remove dependencies:
      $T \leftarrow T \backslash \{n \in T :$ there exists $j \in D$ with $w_{i,j} > 0, w_{n,j} > 0\}$
  Return: $T'$ (instead of $T$)
```

Again, when the algorithm terminates, the corresponding constraints for are right, so exchanging sums, we obtain

$$\sum_{i \in T} f_i + \sum_{j \in D} \min_{i \in T} c_{i,j} = \sum_{i \in T} \sum_{j \in N(i)} w_{i,j} + \sum_{j \in D} \min_{i \in T} c_{i,j} = \sum_{i \in T} \sum_{j \in N(i)} w_{i,j} + \sum_{i \in T} \sum_{j \in N(i)} c_{i,j}$$
$$= \sum_{i \in T} j \in N(i)(w_{i,j} + c_{i,j}) = \sum_{i \in T} \sum_{j \in N(i)} v_j = \sum_{j \in D} v_j \cdot |N(j) \cap T|.$$

Once again, we fall into the deadend of bounding $|N(j) \cap T|$. The workaround here is to further modify $T$ into a $T'$ so that each $|N(j) \cap T'|$ are disjoint. This is easy, and can be fixed by appending the red section in the algorithm above. Then, with some notation pushing, one can show that this gives a 3-approximation, the detail of which has been omitted here.

# 6 Randomized Algorithms

## 6.1 Global Min-Cut

In this problem, let $G = (V, E)$ be undirected and unweighted. A global minimum cut is a direct generalization of a minimum $s - t$ cut but taken across all vertex pairs. Therefore in order to find a global min cut, one way is to brute force iterate through vertex pairs and find the min cut separate them.

In this section, we proposed another edge-contraction based randomized algorithm for a more efficient alternative. Let $e \in E$ be given and define $G_e$ to be the graph with $e$ contracted, i.e., both endpoints of $e$ collapsing into one — note that the resulting $G_e$ may be a multi-graph, with more than one distinct edge between certain pairs of nodes. The key insight is $C$ is a cut of $G$ if ($e \notin C \Rightarrow C$ is also a cut in $G_e$).

If we pick an edge at random, and if a cut has $|C| \leqslant k$, then it is clear that $\mathbb{P}(C \in G_e) \geqslant 1 - k/m$. Therefore, this gives an iterative randomized algorithm, where we can repeatedly (i) pick a random edge, (ii) contract along that edge, and (iii) stop when there are two vertices left. Then we return the set of edges that did not collapse (in the end, the graph is a multi-graph with two vertices and many edges in between them). A probabilistic bound shows that

$$\mathbb{P}(C \text{ survives}) \geqslant \binom{n}{2}^{-1}.$$

This is a *very* weak result, but what if we run this algorithm many times and return the best result? By repeating the above contraction algorithm $t$ times,

$$\mathbb{P}(\text{success}) = 1 - \mathbb{P}(\text{failure}) = 1 - \left(1 - \binom{n}{2}^{-1}\right)^t \geqslant 1 - n^\alpha$$

for some $\alpha > 0$ with $t \sim \alpha\binom{n}{2}\log n$. This is better, but the runtime of $\mathcal{O}(n^4)$ isn't ideal. How do we improve? Observe that we are contracting the same graph from scratch many times, which leads to many excess work. The idea lies in observing that our cut $C$ is more likely to survive in a larger graph, so it is in general a better idea to save the earlier phases of the contraction and focus more on contracting smaller graphs. This gives rise to the following recursive algorithm:

**Algorithm 17:** Recursive Graph Contraction

```
1  Inputs: graph $G = (V, E)$, undirected and unweighted
2  if $V$ is sufficiently small then brute force return min cut
3  $t = n - n/\sqrt{2}$
4  for $i = 1, \cdots, t$ do
5      randomly contract, like in the vanilla algorithm
6  # now $t = n\ \sqrt{2}$ and we have $n - n/\sqrt{2}$ edges left
7  fork two sub-processes:
8     $C_1 \leftarrow$ recursive contraction on $G$
9     $C_2 \leftarrow$ another instance of recursive contraction on $G$
10 return the better (cheaper) of the two cuts
```

Its runtime is given by $T(n) \leqslant 2T(n/\sqrt{2}) + \mathcal{O}(n)$ which yields $T(n) = \mathcal{O}(n^2 \log n)$. Here, the chance of success is $\mathcal{O}(1/\log n)$, so after repeating $\mathcal{O}(\log^2 n)$ times we can boost it into $1 - n^{-\alpha}$ like before. The runtime indeed improves from the previous $\mathcal{O}(n^4)$ to now $\mathcal{O}(n^2 \log^3 n)$.

## 6.2 LP-Based Randomized Rounding

Beginning of 10/21/2024

In this section we consider the MAX-SAT problem. Let us consider a **conjunctive normal form** (CNF), i.e., a formula of clauses, such that each clause consists of a series of ORs of literals, and that all clauses are connected via ANDs. One example could be (and this happens to be a 3-CNF since all clauses have length 3)

$$(x_1 \vee \overline{x}_2 \vee \overline{x}_4) \wedge (x_2 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3).$$

It is well-known that SAT, the satisfiability problem, which asks if there exists an assignment of truth values to the literals such that the overall CNF evaluates to true, is in general NP-hard.

Here, we consider a slight variant, the MAX-SAT, or the **max satisfiability** problem. Suppose there are $m$ clauses in total, where each clause $c_j$ is assigned weight $w_j$. The objective of MAX-SAT is to find a truth assignment so that the total weight of satisfied clauses is maximized. Immediately, one can see that SAT reduces to MAX-SAT, so unless P = NP we cannot expect to solve this problem in polynomial time, either.

In what follows, we consider two approaches to solving MAX-SAT.

### A Randomized Approach

Arguably the most naïve way to propose a solution is to randomly assign true/false to each variable. That is, for each variable $x_i$, we set $x_i$ to be true with probability $1/2$ and false otherwise. Thanks to the structure of CNF, even if we start with a random guess, because a clause is satisfied as long as at least one of the literals evaluates to true, when clauses are long, even random guess performs reasonably well. Let us define $\ell_i$ to be the length of clause $c_j$. Then

$$\mathbb{P}(c_j \text{ satisfied}) = 1 - \mathbb{P}(c_j \text{ not satisfied}) = 1 - 2^{-\ell_i}.$$

This goes to imply that if we define $\ell = \min_j \ell_j$ to be the length of the shortest clause, then

$$\mathbb{E}(\text{total weight of satisfied clauses}) = \sum_j (1-2^{-\ell_j})w_j \geqslant (1-2^{-\ell})\sum_j w_j \geqslant (1-2^{-\ell})\mathrm{OPT}, \tag{6}$$

so in other words we obtain a $1-2^{-\ell}$ approximation. The larger the $\ell$, the better this approximation is. However, when $\ell = 1$ for example, this only gives a $1/2$-approximation.

Next up, we propose an LP-based algorithm that aims to improve the performance for instances where there exist shorter clauses.

### Another LP-Based Randomized Rounding

Next up, we consider an LP formulation for MAX-SAT. For each clause $c_i$, define $P_i, N_i$ to be the set of indices of non-negated literals and negated variables, respectively. For example, $x_1 \vee \overline{x}_2 \vee x_3$ has $P = \{1,3\}$ and $N = \{2\}$. For the primal, we define variables $y_i$ corresponding to literals $x_i$, and $z_j$ corresponding to clauses $c_j$. Then the LP is

$$\max \sum_{j=1}^m w_j z_j \qquad \text{such that} \qquad \begin{cases} \sum_{i\in P_j} y_i + \sum_{i\in N_j}(1-y_i) \geqslant z_j & \text{for each clause } c_j \\ 0 \leqslant y_i, z_j \leqslant 1. \end{cases}$$

Let $(y^*, z^*)$ be an optimal fractional solution. For each variable $x_i$, we round $y_i^*$ via $\mathbb{P}(x_i = 1) = y_i^*$. Algebra plus the definition of violating a clause imply

$$\begin{aligned} \mathbb{P}(c_j \text{ not satisfied}) = \mathbb{P}(\text{everything wrong}) &= \prod_{i\in P_j}(1-y_i^*)\prod_{i\in N_j} y_i^* \\ &\leqslant \left[\frac{1}{\ell_j}\left(\sum_{P_i}(1-y_i^*) + \sum_{N_j} y_i^*\right)\right]^{\ell_j} && \text{(AM-GM, } (\textstyle\prod a_i)^{1/k} \leqslant \sum a_i/k) \\ &= \left[1 - \frac{1}{\ell_j}\left(\sum_{P_j} y_i^* + \sum_{N_j}(1-y_i^*)\right)\right]^{\ell_j} \leqslant (1 - z_j^*/\ell_j)^{\ell_j} \end{aligned}$$

where the last inequality is due to the primal constraint. This shows

$$\mathbb{P}(c_j \text{ satisfied}) = 1 - (1 - z_j/\ell_j)^{\ell_j}.$$

Viewing above RHS as the concave function $f(z_j) = 1-(1-z_j/\ell_j)^{\ell_j}$, using the fact that $f(0) = 0, f(1) = 1-(1-1/\ell_j)^{\ell_j}$, and the fact that concavity implies $f(x) \geqslant f(0) + x(f(1) - f(0))$ on $[0,1]$,

$$\mathbb{P}(c_j \text{ satisfied}) \geqslant 1 - (1 - z_j/\ell_j)^{\ell_j} \geqslant (1-(1-1/\ell_j)^{\ell_j})z_j^* \geqslant (1-1/e)z_j^*. \tag{7}$$

Therefore, the overall objective satisfies

$$\mathbb{E}\Big[\sum_{j=1}^m w_j z_j\Big] \geqslant (1-1/e)\sum_{j=1}^m w_j z_j^* \geqslant (1-1/e)\mathrm{OPT}.$$

### Combining Both Algorithms

Observe that in the first randomized algorithm, with respect to each clause $c_j$, the approximation ratio is $1-2^{-\ell_j}$, and that in the second algorithm, this ratio is $1-(1-1/\ell_j)^{\ell_j}$. For integer-valued $\ell_j \geqslant 1$, the average between them is $\geqslant 2$, so the larger one must be. This means **if we take the better of the two algorithms, we are guaranteed a 3/4-approximation.**

### 6.2.1 Detour: Facility Location with LP Rounding

Here we provide a high-level overview of two additional rounding-based methods to solve the facility location problem. Recall that the

$$\min\left[\sum_{i\in F} f_i y_i + \sum_{i\in F, j\in D} c_{i,j}x_{i,j}\right] \text{ subject to } \begin{cases} \sum_{i\in F} x_{i,j} \geqslant 1 & \text{for all customer } j \\ x_{i,j} \leqslant y_i & \text{for all } i,j \\ x_{i,j}, y_i \geqslant 0. \end{cases} \qquad \text{(Facility Primal)}$$

$$\max \sum_{j\in D} v_j \text{ subject to } \begin{cases} \sum_{j\in D} w_{i,j} \leqslant f_i & \text{for each facility } i \\ v_i - w_{i,j} \leqslant c_{i,j} & \text{for all } i,j. \end{cases} \qquad \text{(Facility Dual)}$$

Let $(x^*, y^*)$ be an optimal primal solution and $(v^*, w^*)$ optimal dual. For each customer/demand $j \in D$, we define $N(j)$ (with the abuse of notation, totally different from the previous $N(j)$) to be $N(j) = \{i \in F : x^*_{i,j} \geqslant 0\}$, i.e., the set of facilities such that there is a (fractional) potential of serving $j$ via facility $i$. We also define the second-order neighbor $N^2(j)$ to be $N^2(j) = \{\ell \in D : N(j) \cap N(\ell) \neq \varnothing\}$. This is the set of demand points that share a neighborhood, or the set of customers that can potentially share the same facility with $j$. For the rounding scheme, we want to always match customers to the cheapest qualifying facility. This gives rise to the following:

**Algorithm 18:** Facility Locations LP Rounding

```
1  Inputs: $x^*_{i,j}, y^*_i, v^*_j, w^*_{i,j}$ from optimal fractional primal/dual
2  Initialization: $C = \varnothing$ (unserved demands), $X = \varnothing$ (opened facilities), $k = 0$
3  while $C \neq \varnothing$ do
4      $k \leftarrow k + 1$
5      let $j = \arg\min_{j\in C} v^*_j$ # next customer to serve
6      find $i = \arg\min_{i\in N(j)} f_i$ # best facility to serve $j$
7      $X \leftarrow X \cup \{i\}$ # open facility $i$
8      assign all customers $j' \in N^2(j)$ to facility $i$ # so we are done with facility $i$
9      $C \leftarrow C \backslash N^2(j)$ # note this includes $j$ itself
10 return $X$
```

It can be shown that this algorithm is 4-approximate. The reason for this large approximation ratio is because in line 8, we are assigning all second-degree neighbors $j'$ to $i$, based solely on the fact that facility $i$ works best for $j$ — there is no guarantee that $i$ is also a good choice for other customers $j'$.

To combat this issue, for each customer $j$ we define a second metric $c^*_j = \sum_{i\in N(j)} x^*_{i,j}$, pick next customer based on the overall minimizer of $v^*_j + c^*_j$, and draw the best facility based on a soft probability distribution. It can be shown that the following algorithm achieves a better 3-approximation ratio.

**Algorithm 19:** Facility Locations LP Rounding

```
1  Inputs: $x^*_{i,j}, y^*_i, v^*_j, w^*_{i,j}$ from optimal fractional primal/dual
2  Initialization: $C = \varnothing$ (unserved demands), $X = \varnothing$ (opened facilities), $k = 0$
3  for $j \in D$ do
4      define $c^*_j = \sum_{i \in N(j)} x^*_{i,j}$
5  while $C \neq \varnothing$ do
6      $k \leftarrow k + 1$
7      let $j = \arg\min_{j \in C}[v^*_j + c^*_j]$ # next customer to serve
8      draw $i$ from $N(j)$ with distribution $\mathbb{P}(\text{open } i \in N(j)) = y^*_i$
9      assign all customers $j' \in N^2(j)$ to facility $i$ # so we are done with facility $i$
10     $C \leftarrow C \backslash N^2(j)$ # note this includes $j$ itself
11 return $X$
```

## 6.3 Semidefinite Programming (SDP)

In this section, we consider an alternative to linear programming LP, where we introduce a certain amount of geometric complexity by drawing connection to PSD (positive semidefinite) matrices in higher dimensions.

### Max-Cut

Firstly, we consider the Max-Cut problem, where given an undirected graph $G = (V, E)$ with edge weights $w_e \geqslant 0$, we want to find the cut $(S, S^c)$ that maximizes the weights of edges crossing the cut, i.e., $\max \sum_{e \in \partial S} w(e)$. It is known that Max-Cut is NP despite its counterpart, min-cut or max-flow, can be solved efficiently.

A baseline for Max-Cut is that any algorithm we propose must do better than "random guess." Specifically, if for each edge we independently include it with probability $1/2$ in $S$, then

$$\mathbb{E}(w(S)) := \mathbb{E} \sum_{e \in \partial S} w(e) = \frac{1}{2} \sum_{e \in E} w(e) \geqslant \frac{\text{OPT}}{2}$$

so the baseline is we can certainly achieve a $1/2$-approximation factor. A slightly more sophisticated threshold rounding scheme on the LP relaxation gives a $3/4$-approximation. But how can we do better?

The idea behind SDP is that instead of assigning each variable $y_i \in \{0, 1\}$, we let $y_i \in \{-1, 1\}$. Then, $(1 - y_i y_j)/2 = 0$ is a binary variable that equals 1 if and only if $y_i, y_j$ are assigned different values. Therefore, the SDP objective is

$$\max \frac{1}{2} \sum_{(i,j) \in E} (1 - y_i y_j) w_{i,j} \qquad \text{subject to} \qquad y_i \in \{-1, 1\}.$$

To fully generalize this problem into matrix algebraic notations and enjoy the benefits of geometry of PSD matrices, for vertex variable $y_i$, we define its **vertex relaxation** to be $v_i \in \mathbb{R}^n$, such that all $v_i$'s are unit vectors. The problem can now be written w.r.t. vector products

$$\max \frac{1}{2} \sum_{(i,j) \in E} (1 - v_i^T v_j) w_{i,j} \qquad \text{subject to} \qquad \|v_i\| = 1 \text{ for all } v_i \in \mathbb{R}^n. \tag{Max-Cut SDP}$$

First, we answer the question "**why SDP?**" Suppose we know how to solve the problemm, obtaining a set of vector

relaxation solution $\{v_i^*\}$. By definition, these points lie on the unit sphere in $\mathbb{R}^n$. Recall that our original objective is to partition $V$ into two sets of vertices, one for $S$ and another for $S^c$.
The most natural interpretation of $\{v_i^*\}$, therefore, is to partition these unit vectors into two sets as well. We consider an extremely simple approach: (i) pick a random unit vector $r$, and (ii) choose $S = \{i : r^T v_i^* > 0\}$, i.e., one of the hemispheres defined by the hyperplane to which $r$ is normal. Let $\theta_{i,j}$ denote the angle between $v_i$ and $v_j$, which can be calculated by $\theta_{i,j} = \cos^{-1}(v_i^{*T} v_j^* / (\|v_i^T\| \|v_j^*\|)) = \cos^{-1}(v_i^{*T} v_j^*)$. A simple geometric interpretation shows that $\mathbb{P}(i, j \text{ separated by cut}) = \mathbb{P}(v_i^*, v_j^* \text{ in different hemispheres}) = \theta_{i,j}/\pi = \pi^{-1} \cos^{-1}(v_i^{*T} v_j^*)$.
It can be shown that $\theta$, $\pi^{-1}\cos^{-1}(x) \geqslant 0.878(1-x)/2$, so this implies that the SDP randomized algorithm achieves an approximation factor of $0.878$. Two side notes:

- Unless P = NP, it can be shown that MAX-CUT cannot have an approximation factor better than $0.94$, and
- Under UGC (unique game conjecture), this factor of $0.878$ is already the best possible.

A side remark on how to solve the problem: the more general form of SDP (and also how it got this name) is

$$\max \sum_{i,j} c_{i,j} X_{i,j} \qquad \text{subject to} \qquad \begin{cases} \sum_{i,j} a_{i,j_k} X_{i,j} = b_k & \text{for each constraint } k \\ X \geq 0 & \text{(PSD).} \end{cases}$$

With these assumptions, the Ellipsoid method works. So in the future we will assume that we have the black box that solves the vector relaxations of SDP.

**Correlation Clustering**

Suppose we have a graph $G = (V, E)$, where for each edge $(i, j)$, we have a **similarity score** $w_{i,j}^+$ and a **dissimilarity score** $w_{i,j}^-$. The correlation clustering problem asks us to partition the vertices $V$ into $k$ clusters $S = \{S_1, \cdots, S_k\}$, while maximizing (i) the similarity scores of vertex pairs that are inside the same component, and (ii) the dissimilarity scores of vertex pairs that belong to different components.
In other words, if we define $E(S) = \{(i,j) : i, j \text{ both belongs to some } S_r\}$ the set of intra-cluster edges and $\delta(S) = \{(i,j) : i \in S_r, j \notin S_r\}$ the set of inter-cluster edges, then our objective is

$$\max w(S) = \max \left[ \sum_{(i,j) \in E(S)} w_{i,j}^+ + \sum_{(i,j) \in \delta(S)} w_{i,j}^- \right].$$

How do we translate the problem into a vector problem? We have $k$ clusters, so we define $\{e_1, \cdots, e_k\} \subset \mathbb{R}^k$ to be the set of one-hot vectors (e.g. $(0, \cdots, 0, 1, 0, \cdots, 0)$). These serve as indicators for each cluster. Then, for each vertex $i$, we let $y_i \in \{e_1, \cdots, e_k\}$ to represent its cluster membership. By doing so, $y_i^T y_j = 1$ if they belong to the same cluster and $0$ otherwise. Thus, the objective can now be written as

$$\max \sum_{(i,j) \in E} \left[ w_{i,j}^+ y_i^T y_j + w_{i,j}^- (1 - y_i^T y_j) \right]$$

Finally, just like how we always relax ILP into fractional LP, we now simply assume $y_i \in \mathbb{R}^k$ instead of being one-hot. Once we have an optimal relaxed solution $\{y_i^*\}$, we can pick two random vectors $r_1, r_2 \in \mathbb{R}^k$ which divide the space into $4$ quadrants. By assigning cluster membership based on which quadrant each $y_i^*$ lies in, we obtain a reasonably well approximation ratio using just $4$ clusters. With more random vectors of form $r$, we can create exponentially more clusters. The results will be slightly better but not much.

## 6.4 Random Sampling

In this section, we reconsider a set system $\Sigma = (X, \mathcal{R})$. For $Y \subset X$, define the subsystem $\Sigma_Y = (Y, \mathcal{R}_Y)$ where $\mathcal{R}_Y = \{R \cap Y \mid R \in \mathcal{R}\}$. In many applications, if $X = \mathbb{R}^d$, we assume $|\mathcal{R}| = \mathcal{O}(n^d)$ where $d$ is a constant.
The idea of **sampling** is we want to know more about ranges/sets $R \in \mathcal{R}$ by focusing on some local structures, especially when we do not have easy access to the entire space. More specifically, let $\mu$ be a probability distribution over $X$; we want to measure $\mu(R)$, the probability of certain events. For simplicity, let us first assume that $\mu$ is uniform, so $\mu(R) = |R|/|X|$, a quantity known as the **fractional size** of $R$. Our goal is answer the following question:

Is there a small **sample** $A \subset X$ such that

$$\frac{|R \cap A|}{|A|} = \frac{|R|}{|X|} \qquad \text{for all } R \in \mathcal{R}?$$

If so, we know that $A$ is "representative" of the structure of $X$.

Admittedly, demanding = is probably too harsh. Thus we also define a weaker notion:

Let $\epsilon \in (0,1)$. $A \times X$ is **$\epsilon$-approximate** if

$$\left|\frac{|R \cap A|}{|A|} - \frac{|R|}{|X|}\right| < \epsilon \text{ or equivalently } \left||R \cap A| \cdot \frac{|A|}{|X|} - |R|\right| \leqslant \epsilon|X| \text{ for all } R \in \mathcal{R}.$$

A natural question, that we ask, is about **sample complexity**: how large does our sample need to be in order to be representative or sufficiently good at approximating $X$? Certainly $X$ itself is good, but it may be too large.

**Theorem**

Assume $\mathcal{R}$ has size $\mathcal{O}(n^d)$. Then a random subset $A \subset X$ of size $\mathcal{O}(\log(1/\epsilon\delta)d/\epsilon^2)$ is an $\epsilon$-approximation of $\Sigma$ with probability $\geqslant 1 - \delta$.

We also introduce the notion of **discrepancy**: let $\chi : X \to \{-1, +1\}$ be a coloring (partition into two groups). The discrepancy of a certain set is $\operatorname{disc}(\chi, R) = |\sum_{x \in \mathcal{R}} \chi(x)|$, which measures how unbalanced the coloring on that set is. In addition, the overall discrepancy is defined by $\operatorname{disc}(\chi, \Sigma) = \max_{R \in \mathcal{R}} \operatorname{disc}(\chi, R)$, the maximum discrepancy among all sets. The reason we introduce this notion is because the following theorem implies the one above:

**Theorem**

For a random coloring $\chi$, $\mathbb{P}(\operatorname{disc}(\chi, \Sigma) \geqslant \sqrt{2n\log(2m/\delta)}) \leqslant \delta$, where $n = |X|$ and $m = |\mathcal{R}|$. This bound is tight.

We will provide a high-level proof showing that the second theorem implies the first.

- Initially, define $A_0 = X$, and let $\chi_0$ be a random coloring.
- Next up, we let $A_1 = \{x \in A_0 : \chi_0(x) = 1\}$, and define another random coloring $\chi_1$ on $A_1$.
- Choose a parameter $k$ and repeat until we obtain $A_k$.

Suppose the coloring $\chi_i$ induces error $\epsilon_i$ on $A_i$. We want to choose a $k$ such that $\sum \epsilon_i \leqslant \epsilon$, the error bound in the first theorem. It can be shown that $|A_k| = \mathcal{O}(\log(1/\epsilon)d/\epsilon^2)$, as claimed.

To prove the second theorem, we need some results known as **tail bounds**. We first state the well-known **Markov inequality**, that $\mathbb{P}(X > k) \leqslant \mathbb{E}[X]/k$ if $X$ is a nonnegative random variable and $k > 0$. To see this, consider the indicator random variable $I = \mathbf{1}[X > k]$. Then $X \geqslant k \cdot I$, and

$$1 \cdot \mathbb{P}(X > k) = \mathbb{E}[I] \leqslant \frac{\mathbb{E}X}{k}.$$

What we will be deriving, instead, is (a special case of) the **Chernoff bound**.

**Theorem: Chernoff Bound**

Let $X_1, \cdots, X_n$ be independent random variables uniformly taking values in $\{-1, +1\}$ with probability $1/2$. Let $X = \sum X_i$. Then $\mathbb{P}(X > t) < \exp(-t^2/2n)$, or $\mathbb{P}(|X| > t) < 2\exp(-t^2/2n)$.

*Proof.* The proof of Chernoff bound is rather elegant — it introduces another parameter $c$, shows that $\mathbb{P}(X > t)$ is bounded by a function of $c$ for any $c$, then minimize this function and discard the dependency of this additional parameter.

More specifically, by monotonicity of exponential function, $\mathbb{P}(X > t) = \mathbb{P}(e^{cX} > e^{ct})$ for any nonzero $c$. By Markov inequality we know

$$\mathbb{P}(X > t) = \mathbb{P}(e^{cX} > e^{ct}) < \frac{\mathbb{E}e^{cX}}{e^{ct}} = \frac{e^{nc^2/2}}{e^{ct}} = \exp(nc^2/2 - ct),$$

where $\mathbb{E}e^{cX} = \prod_{i=1}^{n} \mathbb{E}e^{cX_i} = e^{c^2/2}$. The inequality above holds for any $c$, so we pick $c = t/n$ which minimizes $\exp(nc^2/2 - ct)$. This gives $\mathbb{P}(X > t) < \exp(-t^2/2n)$, as claimed. □

A more general corollary, known as the **Hoeffding inequality**, states the following:

$$\mathbb{P}\left(\sum_{i=1}^{n} a_i X_i > t\right) < \exp\left(-\frac{t^2}{2\sum_{i=1}^{n} a_i^2}\right).$$

The theorem on coloring is now obvious, once we apply the Chernoff bound: for a particular range $R$,

$$\mathbb{P}(\mathrm{disc}(\chi, R) > \sqrt{2n\log(2m/\delta)}) < 2\exp\left(-\frac{2n\log(2m/s)}{2n}\right) = 2\exp(-\log(2m/\delta)) = \frac{\delta}{m}.$$

Therefore, union bound implies

$$\mathbb{P}(\text{there exists } R \text{ with } \mathrm{disc}(\chi, R) > \sqrt{2n\log(2m/\delta)}) < m \cdot \delta/m = \delta,$$

which completes the proof, since the LHS is simply $\mathbb{P}(\mathrm{dist}(\chi, \mathcal{R}) > \sqrt{2n\log(2m/\delta)})$.

## 6.5 Tree Embeddings

In this section, we first throw a bunch of definitions on isometric embeddings, then consider a cool application that is reminiscent of binary search.

Let $(X, \rho)$, $(Y, \delta)$ be two metric spaces. We will view them as graphs (so for example, a metric could be the shortest distance). More specifically, assume $Y$ is a tree. A function $f(X \to Y)$ is said to be an **isometric tree embedding** if $\rho(u, v) = \delta(f(u), f(v))$. This is a very demanding property, and chances are most embedding cannot satisfy it — in fact, there are plenty of examples of metric spaces where an isometric embedding cannot even exist. Instead, we

take one step back and ask, *how to prevent distortion as much as possible*? To this end, we pursue **low distortion embeddings**.
We give a weaker definition: $f : X \to Y$ is said to be a ***D*-embedding** if there exists $r$ such that

$$r\rho(x,y) \leqslant \delta(f(x), f(y)) \leqslant rD\rho(x,y).$$

In other words, the stretch factor of $f$ is bounded by $[r, Dr]$. Our goal, therefore, is to find an embedding where this **distortion factor** $D$ can be minimized, so $f$ looks "almost" like an embedding that simply scales things.

Before considering any systematic methods to construct embedding, let us look at a simple example, a cyclic graph $C_n$ with $n$ nodes. What would the distortion be if we were to wrap it into a tree $Y$?
Well, two originally adjacent nodes may now end up having a distance $n-1$ for being on the opposite sides, which gives a distortion factor of $\mathcal{O}(n)$. We can do better, however, if we allow auxiliary nodes in $Y$: for example we can build a tree with $\log n$ layers, putting all $n$ nodes in $C_n$ as leaves in $Y$. This reduces the distortion factor to $2\log n = \mathcal{O}(\log n)$. (One needs to pay extra attention to arranging the nodes to preserve the lower bound $r\rho(x,y) \leqslant \delta(f(x), f(y))$ too, but the result is indeed $\mathcal{O}(\log n)$.)

A disadvantage of directly considering trees is this approach becomes hard for general graphs. Instead, we consider a **probabilistic embedding**. We want the distortion factor to hold in expectation: $\mathbb{E}[\delta(f(x), f(y))] \leqslant D\rho(x,y)$.

**Theorem**

Given $(X, \rho)$, there exists a probabilistic tree embedding $(Y, \delta)$ such that

(1) $\rho(X, y) \leqslant \delta(f(x), f(y))$, and

(2) $\mathbb{E}[\delta(f(x), f(y))] = \mathcal{O}(\log n)\rho(x,y)$.

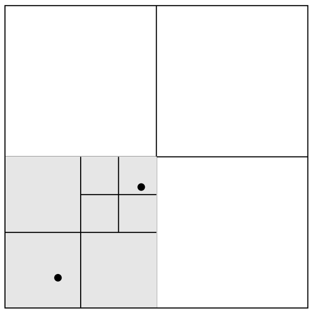
In other words, a distortion factor of $\mathcal{O}(\log n)$ can be guaranteed!

The proof is very lengthy. In the following space we will demonstrate one special case: $X = \mathbb{R}^2$, equipped with Euclidean distance as metric. The tree we will build is a **quad tree**. The intuition is extremely simple: given a square, divide it into four quadrants to split the points, essentially creating a four-way search.
In slightly more details, each node $u$ is associated with a square $B_u$. If $|B_u \cap X| = 1$, then this square uniquely identifies node $u$ so we are done and set $u$ as a leaf node. Otherwise, we need to perform another four-way split of $B_u$ and create four more children nodes. Repeat until all nodes/points in $X$ are uniquely identified by squares.
For practical purposes, we will assume that the set of points $X$ is well-behaved, i.e., the **spread** $\text{spread}(X) = \max_{p,q} \|p-q\| / \min_{p,q} \|p-q\|$ is $n^{\mathcal{O}(1)}$. (This means we exclude extreme cases where we only have two cluster of points at the opposite diagonal of a square.) So now let us build a tree out of these squares. For a node $u$, we define (i) $B_u$ to be the square that uniquely identifies it, (ii) $c_u$ the center of $B_u$, (iii) $\ell_u$ the side length of $B_u$, (iv) $p(u)$ the parent node, and (v) $u(a)$, the leaf that stores $a \in X$. Let the tree be $T = (Y, F)$ (where $F$ is the set of edges).

- We define $Y = X \cup \{c_u : u \in \text{ the quad tree}\}$.
- $F = \{(a, c_{u(a)}) : a \in X\} \cup \{(c_u, c_{p(u)}) : u \text{ is a node of } T\}$.
- For edges like $(a, u_a)$, we assign cost $\|a - c_{u(a)}\|$ to them, and for edges like $(u, p(u))$, we assign cost $\ell_u$.

Essentially, to reach a point $x$, we traverse along a series of squares (these are edges of form $(u, p(u))$), until we reach the smallest square containing $x$, at which point we move from $c_x$ to $x$ directly. The symbols are likely more complicated to understand than the actual concept itself.

Doing a quad-split aligns with our intuition, but observe we may have very bad examples with huge distortion: points may be really close in $\mathbb{R}^2$ but classified along drastically different quad-tree paths. consider, for example, an initial square $[-1,1] \times [-1,1]$. Though points like $(\epsilon, \epsilon)$ and $(-\epsilon, -\epsilon)$ can be very close to each other, they belong to different quadrants! If we run our algorithm, the path from $(0,0)$ to $(\epsilon,\epsilon)$ would be $(0,0) \to (1/2, 1/2) \to (1/4, 1/4) \to \cdots, \to (1/2^n, 1/2^n) \to (\epsilon, \epsilon)$ and likewise the other path to $(-\epsilon, -\epsilon)$ is also symmetric, starting from $(0,0) \to (-1/2, -1/2)$ and so on, a totally different path. The fix is to introduce a random offset into the positioning of the squares and argue that *in expectation* the distortion is good (i.e., such bad examples are unlikely to happen). To this end, we add some randomness to out initial square:



(1) Find the smallest square (or any small square) containing all points in $X$; call it $B$.

(2) Double the side length of $B$, keeping the NE corner anchored.

(3) Pick any point in the SW quadrant of $B$, and translate $B$ so that its SW corner is at this point.

(4) Start building the quad tree.

In other words, we make the initial square larger and still keep $X$ centered in expectation. Consider two points $a$ and $b$ and at what stage a partition separates them. It can be shown that $\mathbb{P}(a, b$ are separated vertically at $j^{\text{th}}$ split$) = |x_a - x_b|/2^j$ and likewise if they are split horizontally. To interpret this, essentially the initial random shift must take place in a very narrow interval range so that $a, b$ are separated precisely at the $j^{\text{th}}$ split. Conditioning on the location of $a$ and $b$'s lowest common ancestor, it can be shown that

$$\mathbb{E}(\delta(a,b)) = \sum_{k=0}^{\mathcal{O}(\log n)} \frac{\|a-b\|}{2^k} 2^{k+1} = \mathcal{O}(\log n)\|a-b\|,$$

completing the claim that the distortion is bounded by $\mathcal{O}(\log n)$.