Optimization and Approximation

About

This text was created as part of the project “Príprava štúdia matematiky a informatiky na FMFI UK v anglickom jazyku”, ITMS: 26140230008, funded by the EU.

In the text, the following graphics were used:

- unit sprites from the project ”Battle for Wesnoth” `http://www.wesnoth.org`
- binder graphics from the project digstud `https://github.com/bitfragment/digstud.git`
- images of printers from manufacturers' web presetnations

Contents

1 Linear programming

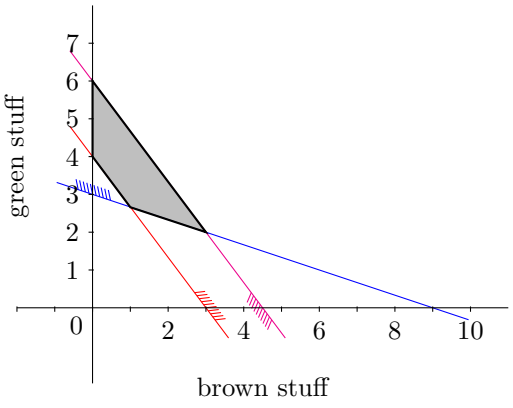
1.1 A gentle introduction

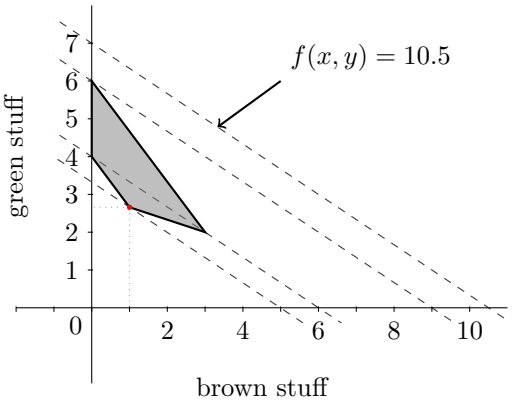
Let us start with an example that is, in some of its many forms, a must in any text about linear programming. Let us have a student going to pull an all-nighter in preparation for some exam. In order to stay awake he needs to get at least 270 mg of caffeine, and at least 120 g of sugar. At the same time, a dose of more than 180 mg of aspartame may be hazardous to his health. He can choose between two drinks: brown stuff for €1/dl, and green stuff for €1,50/dl. 1 dl of brown stuff contains 30 mg of caffeine, 40 g of sugar, and 40 mg of aspartame, while the same amount of green stuff contains 90 mg of caffeine, 30 g of sugar, and 30 mg of aspartame. How much of which stuff should the student buy in order to satisfy all his needs at the smallest possible cost? If he buys x dl of brown stuff, and y dl of green stuff, the problem (called a *linear program*) can be formulated as follows:

	dl of stuff brown		green			
minimize	x	$+$	$1.5y$	$=:$	$f(x,y)$	*price*
subject to	$30x$	$+$	$90y$	$\geq$	270	*caffeine*
	$40x$	$+$	$30y$	$\geq$	120	*sugar*
	$40x$	$+$	$30y$	$\leq$	180	*aspartame*
			x, y	$\geq$	0	

(1)

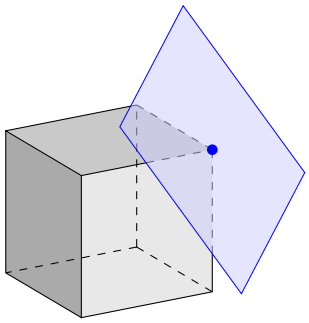
It is easy to see that, e.g., 4 dl of green stuff satisfy all the requirements at the cost of €6, while providing even more caffeine than necessary, and less aspartame than allowed. On the other hand, if the student would be buying only brown stuff, in order to satisfy his needs for caffeine he would have to buy 9 dl of it, which would overdose him with aspartame (and it would be too costly anyway). When looking for the optimal optimal solution the following geometric interpretation comes handy: to each solution with x dl of brown stuff, and y dl of green stuff assign a point (x, y) in the plane. Each of the constraints then defines a half-plane where a feasible solution can be located. Hence, all feasible solutions lie in the quadrilateral on the left:





At the same time we know that $f(x, y) = x + 1.5y$ is a linear function, so points with the same value of f form a straight line (see the right figure). Now it should be clear that in order to find the optimal solution it is sufficient to to check the four corners of the quadrilateral and conclude that the best thing to do is to buy 1 dl of brown stuff, and $2\frac{2}{3}$ dl of green stuff for €5.

The previous ideas lead directly to a simple algorithm for solving the problem with two variables, and a number of inequality constraints: for each constraint consider the corresponding half-plane, construct the (convex) polygon that is the intersection of all these halfplanes, and choose the optimal solution from among all its vertices. With three variables, the constraints are of the form $a_1x + a_2y + a_3z \geq b$, and they define half-spaces in 3D, whose intersection is a polyhedron. Points with the same value of the function $f(x, y, z) = c_1x + c_2y + c_3z$ form a plane, and so it is sufficient to check all the vertices of the polyhedron to find the optimal solution.



With increasing the number of variables, however, many problems arise, and it soon becomes clear that a straightforward generalization is a road to hell. Let us have a different look at the geometry of the linear program. First, let us rewrite program (1) to an equivalent form as follows: multiply the function $f(x, y)$ by -1, and from minimization problem we get a maximization. Then also premultiply by -1 the first and second constraint to get all the constraints (except those $x, y \geq 0$) in the "$\leq$" form. Finally, since in each constraint the left hand side is smaller than the right hand side, we can add a new non-negative variable to represent the difference between the right hand side and the left hand side. These transformations yield the following linear program which is obviously equivalent to the original one:

$$\begin{array}{rrrrrrrrr} \text{maximmize} & -x & -1.5y & & & & & =: & f(x,y,s_1,s_2,s_3) \\ \text{subject to} & -30x & -90y & +s_1 & & & & = & -270 \\ & -40x & -30y & & +s_2 & & & = & -120 \\ & 40x & +30y & & & +s_3 & & = & 180 \\ & & & & & & x,y,s_1,s_2,s_3 & \geq & 0 \end{array} \tag{2}$$

If we denote

$$\mathbf{c} = \begin{pmatrix} -1 \\ -1.5 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad A = \begin{pmatrix} -30 & -90 & 1 & 0 & 0 \\ -40 & -30 & 0 & 1 & 0 \\ 30 & 40 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} -270 \\ -120 \\ 200 \end{pmatrix}$$

then the program (2) can be written concisely

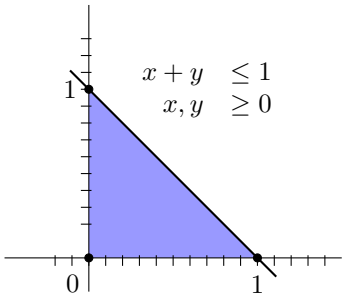
$$\max_{\mathbf{x} \in \mathbb{R}^5} \left\{ \mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq 0 \right\}.$$

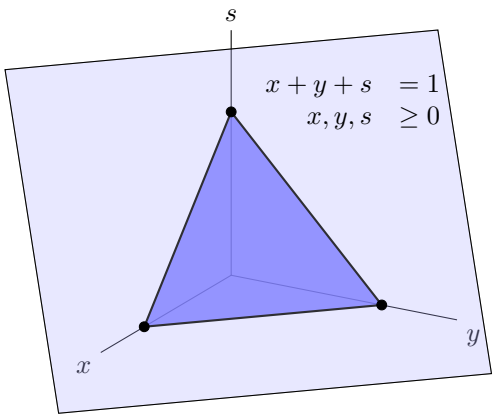
The transformation increased the dimension of the problem from 2 to 5 (and we lost the possibility of a graphical solution), but the gain is a nicer set of feasible solutions: feasible solutions are non-negative solutions of a system of linear equations. In our case the matrix A has rank 3 (the rows are linearly independent) and the solutions of the system $A\mathbf{x} = \mathbf{b}$ form a two-dimensional subspace $\mathbf{o} + c_1\boldsymbol{\alpha} + c_2\boldsymbol{\beta}$ where

$$\mathbf{o} = \begin{pmatrix} 0 \\ 0 \\ -270 \\ -120 \\ 180 \end{pmatrix} \qquad \boldsymbol{\alpha} = \begin{pmatrix} 1 \\ 0 \\ 30 \\ 40 \\ -40 \end{pmatrix} \qquad \boldsymbol{\beta} = \begin{pmatrix} 0 \\ 1 \\ 90 \\ 30 \\ -30 \end{pmatrix}.$$

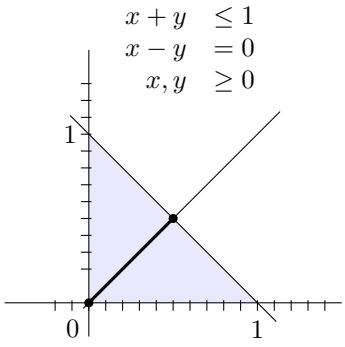
In other words, the feasible solutions of program (1) form a 2-dimensional polygon (quadrilateral) in a 2-dimensional subspace (a plane), while the feasible solutions of program (2) form a

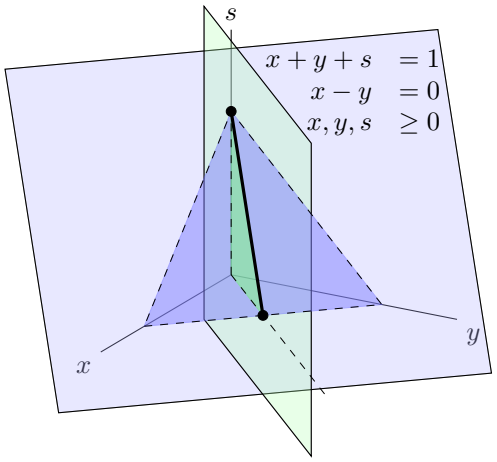
2-dimensional polygon $\mathcal{D}$ in a 5-dimensional space, where $\mathcal{D}$ is the intersection of a (2-dimensional) plane with the positive orthant[1]. To illustrate this, consider the following simple programs (we show only the constraints, the utility function does not matter in this case):





On the left there is a program with two variables and a 2-dimensional solution space. After the transformation into three dimensions the solution space remained 2-dimensional, and it is the intersection of a plane with the positive octant. In the next example the original problem has a 1-dimensional solution space. After the transformation into three dimensions it is still 1-dimensional, and it is an intersection of a line with the positive octant.





The advantage of this form of the linear program is that it is easier to identify the vertices of the polygon of feasible solutions: they lie in some face of the form $x_i = 0$.
Before continuing in our musings it is good to note that we may, without loss of generality, suppose that the rows of the matrix A are linearly independent: if there is a row that is a linear combination of some other rows, then either there is no feasible solution at all, or by leaving out that row the set of feasible solutions remains the same. We can summarize our ideas, and define a *normal form* of a linear program as follows:

Definition 1.1. A linear program is in the **normal form**, if it is written as

$$\max_{\mathbf{x} \in \mathbb{R}^n} \left\{ \mathbf{c}^\mathsf{T} \mathbf{x} \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0} \right\}, \tag{3}$$

where $A \in \mathbb{R}^{m \times n}$ has rank m. The set of feasible solutions $\mathcal{D} = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$.

- **The goal is maximization.** If the original goal was to minimize a linear function $f(\mathbf{x})$,

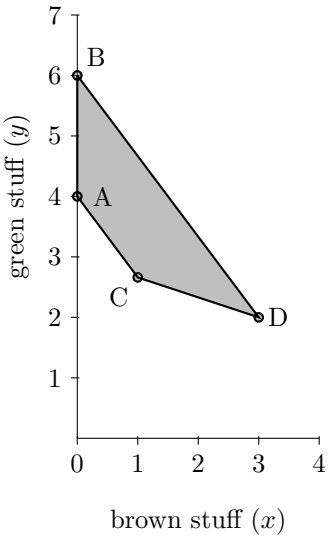
[1] in the same spirit as "quadrant" in 2D, and "octant" in 3D, we use the term "orthant" in a general n-dimensional space

the new goal is to maximize the (linear) function $-f(\mathbf{x})$.

- **All variables are non-negative.** Each variable x that is not bound by a constraint $x \geq 0$ will be replaced by two variables $p_x, q_x \geq 0$, and each occurrence of x in will be replaced by $p_x 0 q_x$.
- **Each constraint, except for those $x \geq 0$ has the form of equality**. A constraint of the form $\sum_i a_i x_i \geq b$ is first multiplied by -1 on both sides, turning it into $\sum_i -a_i x_i \leq -b$. After all constraints have the inequalities facing the same direction, a new variable s (*slack*) is introduced for each constraint $\sum_i a_i x_i \leq b$; this variable represents the extent to which the equality is violated, i.e., $s \geq 0$, and $s + \sum_i a_i x_i = b$.
- **The matrix A has full rank, i.e., has m linearly independent rows.**

Now let us have a program in the normal form. In a way similar to the opening example, we want to find the set of vertices (corners) of the polyhedron $\mathcal{D}$ of the feasible solutions, so it is sufficient to check those vertices in order to find the optimal solution. Since $\mathcal{D}$ is formed by the intersection of the solution space of the system $A\mathbf{x} = \mathbf{b}$ with the positive orthant, its corners have some (at least one) coordinates $x_{i_1}, \ldots, x_{i_k}$ zero (the hyperplane $x_i = 0$ is on the border of the orthant). Moreover, the corners are "pointy" (unlike the points in some face $x_i = 0$ where, say, a whole line segment lies). The "pointyness" of a vertex can be formulated the following way: if the constraints $A\mathbf{x} = \mathbf{b}$ are augmented by the set of equations $x_{i_1} = 0, \ldots, x_{i_k} = 0$, the resulting system has a unique solution (i.e., a "corner" lies in the intersection of some of the boundary faces of the orthant, and no other point lies in the same intersection). Since A has full rank m, in order to have a unique solution of the system, we need $k = n - m$.
Now let us get back to program (2). The matrix A has rank 3, so the vertices of $\mathcal{D}$ are obtained by augmenting the system $A\mathbf{x} = \mathbf{b}$ by two constraints $x_i = 0$, and $x_j = 0$. In this way, 10 different equation systems can be obtained, with the following solutions:

constraints	x	y	s_1	s_2	s_3	
$x = y = 0$	0	0	-270	-120	180	
$x = s_1 = 0$	0	3	0	-30	90	
$x = s_2 = 0$	0	4	90	0	60	A
$x = s_3 = 0$	0	6	270	60	0	B
$y = s_1 = 0$	9	0	0	240	-180	
$y = s_2 = 0$	3	0	-180	0	60	
$y = s_3 = 0$	4.5	0	-135	60	0	
$\mathbf{s_1 = s_2 = 0}$	**1**	$\frac{8}{3}$	**0**	**0**	**60**	C
$s_1 = s_3 = 0$	3	2	0	60	0	D
$s_2 = s_3 = 0$	no solution					



The solutions of the system $A\mathbf{x} = \mathbf{b}$ form a (2-dimensional) plane in a 5-dimensional space. The variables x, y represent the amounts of brown and green stuff, respectively, the variables s_1, s_2, s_3 give the slack of the corresponding constraint. So, for example, the fourth row of the table with constraints $x = s_3 = 0$ tells us that if the student doesn't buy any brown stuff, and at the same time he wants to reach the exact allowed intake of aspartame, he has to buy 6 dl of the green stuff, while he gets more caffeine and sugar than needs. The last row shows that the constraints can not be added arbitrarily, but care must be taken not to introduce any linearly dependent rows (in our case the red and purple lines from the first figure are parallel, so there is no point in their intersection). In the language of linear programs the rows of this table are called *basic solutions*

and those among them that are also feasible (the highlighted rows) are *feasible basic solutions*, and correspond to the vertices of $\mathcal{D}$; in our case the feasible solutions of the program (2) form a (2-dimensional) quadrilateral in a 5-dimensional space.
It is worth noting that the feasible basic solutions of the program (2), i.e., the vertices of the quadrilateral of feasible solutions in the 5-dimensional space correspond in a straightforward way to the vertices of the quadrilateral of the feasible solutions of the program (1): each vertex of the quadrilateral on the right is an intersection of two lines corresponding to a constraint $x_i = 0$ (and the respective basic solution is feasible). On the other hand, every feasible basic solution is the intersection of two such lines.

Noting that the introduction of a constraint $x = 0$ actually deletes a column corresponding to the variable x from the respective equation system, we arrive to the following definition:

Notation. Let $A \in \mathbb{R}^{m\times n}$ be a matrix with m rows and n columns. Given a set $B \subseteq \{1, 2, \ldots, n\}$, the symbol A_B denotes the submatrix of A consisting of columns indexed by the set B. The same notation $\mathbf{x}_B$ will be used for vectors.

For example,

$$A = \begin{pmatrix} -30 & -90 & 1 & 0 & 0 \\ -40 & -30 & 0 & 1 & 0 \\ 30 & 40 & 0 & 0 & 1 \end{pmatrix} \quad A_{\{1,2\}} = \begin{pmatrix} -30 & -90 \\ -40 & -30 \\ 30 & 40 \end{pmatrix} \quad A_{\{3,4,5\}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Definition 1.2. Consider a linear program in the normal form, where $A \in \mathbb{R}^{m\times n}$. A **basic solution** is a vector $\mathbf{x} \in \mathbb{R}^n$, for which there is an m-element set $B \subseteq \{1, \ldots, n\}$ such that

1. the matrix $A_B \in \mathbb{R}^{m\times m}$ has full rank, m (i.e., is non-singular)
2. $x_j = 0$ for all $j \notin B$

Now let us show that our definition of a basic solution is good in the sense that in order to find the optimal solution it is sufficient to check the feasible basic solutions:

Theorem 1.3. *Consider a linear program in the normal form, such that the value of the utility function $\mathbf{c}^\mathsf{T}\mathbf{x}$ is bounded from above on the polyhedron $\mathcal{D}$. Then for every feasible solution $\mathbf{x_0}$ there is some feasible basic solution $\mathbf{x}$, for which $\mathbf{c}^\mathsf{T}\mathbf{x} \geq \mathbf{c}^\mathsf{T}\mathbf{x_0}$.*

Proof: Let us take any feasible solution $\mathbf{x_0}$, and consider all feasible solutions $\mathbf{x}$, for which $\mathbf{c}^\mathsf{T}\mathbf{x} \geq \mathbf{c}^\mathsf{T}\mathbf{x_0}$. Let $\mathbf{x}$ the one from them with the largest number of zero coordinates. We show that $\mathbf{x}$ is basic. If $\mathbf{x} = \mathbf{0}$, it is clearly basic. So let us suppose that $\mathbf{x}$ has at least one non-zero coordinate. Let us denote by $K = \{j \in \{1, \ldots, n\} \mid \tilde{x}_j > 0\}$ the set of all positive (a feasible solution has no negative coordinates) coordinates of the vector $\mathbf{x}$. Subsequently, let us distinguish two cases:

• *The columns of the matrix A_K are linearly independent.* Clearly $|K| \leq m$ (the matrix A has m rows). If $|K| = m$, according to the Definition 1.2, $\mathbf{x}$ has $\tilde{x_j} = 0$ for all $j \notin K$, and the matrix A_K is non-singular. If $|K| < m$, the $|K|$ columns of A_K can be augmented with $m - k$ columns from A in such a way to be linearly independent[2]. Hence, we arrive at a set K', so that $|K'| = m$, $A_{K'}$ is non-singular, and $\tilde{x_j} = 0$ for all $j \notin K' \supseteq K$.

• *The columns of the matrix A_K are linearly dependent,* which means that there is a vector $\boldsymbol{\vartheta} \in \mathbb{R}^{|K|}$ such that $A_K\boldsymbol{\vartheta} = \mathbf{0}$ ($\boldsymbol{\vartheta}$ determines the linear combination of the columns of A_K that yields the zero vector). Let us extend $\boldsymbol{\vartheta}$ to n-dimensional vector $\mathbf{w}$ by letting all coordinates not in K be zero, so that $\mathbf{w}_K = \boldsymbol{\vartheta}$, and $A\mathbf{w} = 0$. For any real $t \geq 0$, let us denote $\mathbf{x}(t) = \mathbf{x} + t\mathbf{w}$. Since $\mathbf{x}$ is a feasible solution, it holds $A\mathbf{x} = \mathbf{b}$. At the same time we have $A\mathbf{w} = \mathbf{0}$, and so also $A\mathbf{x}(t) = \mathbf{b}$. Before concluding the proof, let us modify the vector $\mathbf{w}$ so that $\mathbf{c}^\mathsf{T}\mathbf{w} \geq 0$, and at the same time $w_j < 0$ for some $j \in K$: If $\mathbf{c}^\mathsf{T}\mathbf{w} = 0$, and for all $j \in K$ it holds $w_j > 0$, it is sufficient to multiply

[2]This claim is covered in the introductory algebra course.

$\mathbf{w}$ by -1, and we have the required form. So let it hold $\mathbf{c}^\mathsf{T}\mathbf{w} \neq 0$. If $\mathbf{c}^\mathsf{T}\mathbf{w} < 0$, the $\mathbf{w}$ can be again multiplied by -1, so without loss of generality we can suppose that $\mathbf{c}^\mathsf{T}\mathbf{w} > 0$. Let us show that now there must exist some $j \in K$, for which $w_j < 0$. If it is not the case, i.e., if for all $j \in K$ it holds $w_j > 0$, clearly it must be that $\mathbf{w} \geq \mathbf{0}$ (the coordinates $i \notin K$ have been padded by zeroes). But then $\mathbf{x}(t) = \mathbf{x} + t\mathbf{w} \geq 0$ for all $t \geq 0$, so that $\mathbf{x}(t)$ is a feasible solution. The utility function has value $\mathbf{c}^\mathsf{T}\mathbf{x}(t) = \mathbf{c}^\mathsf{T}\mathbf{x} + t\mathbf{c}^\mathsf{T}\mathbf{w}$. Since $\mathbf{c}^\mathsf{T}\mathbf{w} > 0$, for $t \mapsto \infty$ is $\mathbf{c}^\mathsf{T}\mathbf{x}(t) \mapsto \infty$, and hence the original linear program was not bounded.

Now let us have the vector $\mathbf{w}$ in the form where $\mathbf{c}^\mathsf{T}\mathbf{w} \geq 0$, and at the same time $w_j < 0$ for some $j \in K$. We show that for some $t_1 > 0$, the vector $\mathbf{x}(t_1)$ is a feasible solution with more zero coordinates than $\mathbf{x}$. However, this contradicts the fact that $\mathbf{x}$ has the most zero coordinates from among all feasible solutions $\mathbf{x}$ for which $\mathbf{c}^\mathsf{T}\mathbf{x} \geq \mathbf{c}^\mathsf{T}\mathbf{x_0}$, since $\mathbf{c}^\mathsf{T}\mathbf{x}(t_1) = \mathbf{c}^\mathsf{T}\mathbf{x} + t_1\mathbf{c}^\mathsf{T}\mathbf{w} \geq \mathbf{c}^\mathsf{T}\mathbf{x_0}$ (because $\mathbf{c}^\mathsf{T}\mathbf{x} \geq \mathbf{c}^\mathsf{T}\mathbf{x_0}$ a $\mathbf{c}^\mathsf{T}\mathbf{w} \geq 0$).

The vector $\mathbf{x}(t_0) = \mathbf{x}$ is feasible, and has coordinates $j \in K$ (strictly) positive, ant the remaining ones zero. At the same time we know that there is at least one $j \in K$ for which $w_j < 0$. Since the j-th coordinate of $\mathbf{x}(t)$ is $x(t)_j = \tilde{x}_j + tw_j$, with increasing t the values $x(t)_j$ are decreasing for all j for which $w_j < 0$. Denote t_1 the value t when the first of the values $x(t)_j$ reaches zero. Clearly, $\mathbf{x}(t_1)$ is feasible, and has more zero coordinates than $\mathbf{x}$. □

The consequence of this theorem is that in order to find the optimal solution of a linear program with finite optimum, it is sufficient to check all feasible basic solutions. This is a generalization of the approach from the introductory example with two dimensions, where it was sufficient to check the vertices of a suitable polygon. How to find the basic solutions? Just note that for a given set $B \subseteq \{1, \ldots, n\}$ there is at most one basic solution [3]: if there were two basic solutions $\mathbf{y}, \mathbf{z}$ with the same set B, it must hold $A\mathbf{y} = A\mathbf{z} = \mathbf{b}$ and so $A_B\mathbf{y} = A_B\mathbf{z} = \mathbf{b}$. Since A is a non-singular square matrix, the equation system $A_B\mathbf{x} = \mathbf{b}$ has a unique solution, and hence $\mathbf{y} = \mathbf{z}$. It means that it is sufficient to try all sets $B<$ check it the corresponding A_B is non-singular (e.g., by Gaussian elimination), check if the solution $A_B\mathbf{x} = \mathbf{b}$ is feasible, and choose the best of such feasible $\mathbf{x}$. The problem, of course, is that with n variables, and m constraints, one has potentially $\binom{n}{m}$ distinct basis solutions, so the algorithm that checks all of them is not polynomial [4]. In the next chapter we show how to solve the linear programming problem in a more efficient way using the simplex method.

1.2 The simplex method

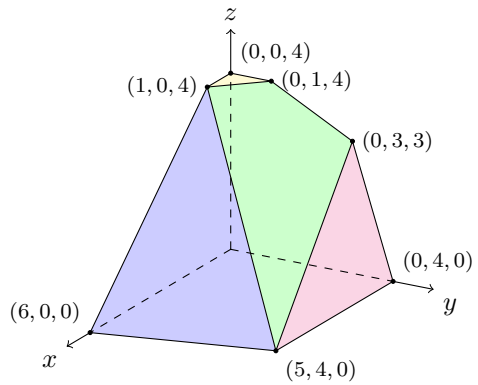
The term *simplex algorithm* denotes any greedy algorithm that traverses the basic solutions (the vertices of $\mathcal{D}$) and always moves to a neighboring solution (along an edge of $\mathcal{D}$) that increases (for maximization problem) the value of the utility function. Again, let us start with an example. Consider the following linear program:

$$\begin{array}{rrrrrl} \text{maximize} & x & +y & +z & =: f(x,y,z) \\ \text{subject toh} & x & +y & +2z & \leq 9 \\ & 4x & +y & +5z & \leq 24 \\ & & 3y & +z & \leq 12 \\ & & & z & \leq 4 \\ & & & x,y,z & \geq 0 \end{array} \tag{4}$$

[3] The other direction does not hold, the same basic solution $\mathbf{x}$ can be possibly obtained from different sets B, B'. If, e.g., the vector $\mathbf{0}$ is feasible, it is also a feasible solution for any basis B.

[4] For, say, $m = n/2$, we get from Stirling approximation $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + o(n))$ that $\binom{n}{\frac{n}{2}} = \frac{n!}{\left[\left(\frac{n}{2}\right)!\right]^2} \geq 2^n/n^2$.

The constraints define half-spaces in a three-dimensional space. Their boundary planes (green, blue, red, and yellow, respectively) define the polyhedron $\mathcal{D}$. By examining all vertices of $\mathcal{D}$ we can conclude that the maximum of the utility function is attained in the point $(5, 4, 0)$.



In accord with the previous part we introduce slack variables $s_1, \ldots, s_4$; if the variables are ordered $x, y, z, s_1, \ldots, s_4$ we get an equivalent program in the normal form

$$\max_{\mathbf{x} \in \mathbb{R}^7} \left\{ \mathbf{c}^\mathsf{T} \mathbf{x} \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq 0 \right\}$$

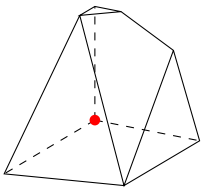
where

$$\mathbf{c} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad A = \begin{pmatrix} 1 & 1 & 2 & 1 & 0 & 0 & 0 \\ 4 & 1 & 5 & 0 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 9 \\ 24 \\ 12 \\ 4 \end{pmatrix}$$

There are 7 indeterminates and the matrix A has rank 4, so the solutions of the system $A\mathbf{x} = \mathbf{b}$ form a three-dimensional subspace in a 7-dimensional space. In particular, we can select any three indeterminates as parameters, and the remaining ones (including the utility function) write as their linear combinations. In our case the matrix $A_{4,5,6,7}$ is diagonal, so it is easy to see that the program (4) can be equivalently written as:

$f =$		x	$+ y$	$+ z$
$s_1 =$	9	$- x$	$- y$	$- 2z$
$s_2 =$	24	$- 4x$	$- y$	$- 5z$
$s_3 =$	12		$- 3y$	$- z$
$s_4 =$	4			$- z$

(5)



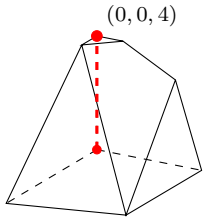
The table (5) is called a *tableaux*, and its meaning is the following: we are looking for values of parameters x, y, z such that the value f is maximized, and at the same time the variables $s_1, \ldots, s_4$ are non-negative. In our case the choice $x = y = z = 0$ makes $s_1, \ldots, s_4$ non-negative. Hence, the tableaux (5) represents the basic solution $(0, 0, 0, 9, 24, 12, 4)$ with the base $\{s_1, s_2, s_3, s_4\}$, and the value of the utility function $f = 0$. Basic variables are in the rows, and the non-basic zero variables are the parameters in the columns.

The picture on the right represents the basic solution from the tableaux (5): Alhough the program we are solving has 7 dimensions, it is easy to see in the same way as in the previous part, that each vertex of the polyhedron in three dimensions (x, y, z) can be uniquely assigned to a basic solution of the program (5).

Our goal is to find the best feasible solution; in particular, its basis. So we want to find some three non-basic variables (parameters) that will be set to zero, and from the expression of the remaining ones (and the utility function) as linear combinations of these parameters we can compute their

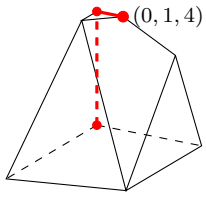
values in this basic solution. Again, if we try all triples of parameters, we scan through all vertices of the polyhedron, and find the best solution. However, we want to avoid this.
How can we locally increase the value of the utility function f? Note that, e.g. the variable z is in the first line multiplied with a positive coefficient $+1$, so if we increase the value of z, also the value of f will be increased. How much can we increase z? Surely, all variables $s_1, \ldots, s_4$ must remain non-negative. This means that every row of the tableaux limits the maximal value of z; the limits are, respectively, $\frac{9}{2}$, $\frac{24}{5}$, 12, 4. So we can set $z := 4$, from wchich we get that $s_4 = 0$. This way we got a new basic solution, in which the non-basic variables will be x, y, s_4 instead of x, y, z. We change our representation accordingly: we express z from the equation for s_4, and substitute into the remaining equations. We get a new tableaux:

$$\begin{array}{rrrrr} f = & 4 & +x & +y & -s_4 \\ \hline z = & 4 & & & -s_4 \\ s_1 = & 1 & -x & -y & +2s_4 \\ s_2 = & 4 & -4x & -y & +5s_4 \\ s_3 = & 8 & & -3y & +s_4 \end{array} \qquad (6)$$



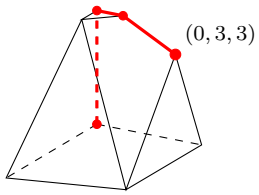
The tableaxu (6) represents a basic solution $(0, 0, 4, 1, 4, 8, 0)$ with basis $\{z, s_1, s_2, s_3\}$ and the value of the utility function $f = 4$. We again have a representation where the rows correspond to basic variables, and the parametres in the columns correspond to non-basic variables. Our goal is to find values for parameters x, y, s_4 in such a way as to maximize the value of f, and at the same time to keep z, s_1, s_2, s_3 non-negative. It is crucial to observe that we performed only an equivalent transformation of the system of linear equations, and so (5) and (6) have the same solutions. A step from (5) to (6) corresponds to a traversal of one edge of the polyhedron of feasible solutions, and we will call it a *pivoting step*: one non-basic variable, *pivot*, in our case z, enters the base, and one basic variable (in our case s_4) becomes non-basic. This step can be iterated. Note, e.g., that in the first row, the variable y occurs with a positive coefficient, an so by increasing y, the value of f is increased. Particular rows yield bounds on the maximum value of y, respectively $1, 4, \frac{8}{3}$; the last equation does not contain y, and hence this row poses no restriction on the value of y. Let us choose $y = 1$ and perform a pivoting step with the pivot y, where y replaces s_1 in the basis. Express $y = 1 - x - s_1 + 2s_4$, and after substitution we get:

$$\begin{array}{rrrrr} f = & 5 & & -s_1 & +s_4 \\ \hline y = & 1 & -x & -s_1 & +2s_4 \\ z = & 4 & & & -s_4 \\ s_2 = & 3 & -3x & +s_1 & +3s_4 \\ s_3 = & 5 & +3x & +3s_1 & -5s_4 \end{array} \qquad (7)$$



Now we have a tableaux for a basic solution $(0, 1, 4, 0, 3, 5, 0)$ with basis $\{y, z, s_2, s_3\}$ and the value $f = 5$. Let us continue a couple of steps further. The only way how to increase f is to choose a pivot s_4, and remove s_3 from the basis, obtaining:

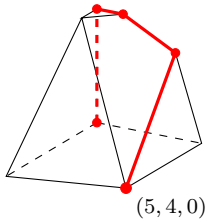
$$\begin{array}{rrrrr} f = & 6 & +\frac{3}{5}x & -\frac{2}{5}s_1 & -\frac{1}{5}s_3 \\ \hline y = & 3 & +\frac{1}{5}x & +\frac{1}{5}s_1 & -\frac{2}{5}s_3 \\ z = & 3 & -\frac{3}{5}x & -\frac{3}{5}s_1 & +\frac{1}{5}s_3 \\ s_2 = & 6 & -\frac{6}{5}x & +\frac{14}{5}s_1 & -\frac{3}{5}s_3 \\ s_4 = & 1 & +\frac{3}{5}x & +\frac{3}{5}s_1 & -\frac{1}{5}s_3 \end{array} \qquad (8)$$



Again, the only possibility how to perform a pivoting step is to include x in the basis. However,

setting $x = 5$ zeroes out both z and s_2, so we can choose which of them will leave the basis. Let us decide that x replaces z in the basis, thus obtaining

$$\begin{array}{rrrrr} f = & 9 & - z & - s_1 & \\ \hline x = & 5 & - \frac{5}{3}z & - s_1 & + \frac{1}{3}s_3 \\ y = & 4 & - \frac{1}{3}z & & - \frac{1}{3}s_3 \\ s_2 = & & 2z & + 4s_1 & - s_3 \\ s_4 = & 4 & - z & & \end{array} \tag{9}$$



Now we got stuck in a situation when no further pivoting step can be performed. However, we only performed equivalent transformations so far, and hence the solutions of (4) and (9) are the same. Moreover, from (9) can easily be seen that for any non-negative z and s_1 the value of f is at most 9, meaning that the solution we found is optimal.

The previous example can be generalized. Let us formally define the tableaux corresponding to a basic solution as follows:

Definition 1.4. Let us have a program in the normal form

$$\max_{\mathbf{x}\in\mathbb{R}^n} \left\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq 0\right\}$$

where $A \in \mathbb{R}^{m\times n}$, and a feasible basic solution corresponding to a basis B. The **tableaux** $\mathcal{T}(B)$ corresponding to the basis B is the system of $m+1$ linear equations with indeterminates $x_1, \ldots, x_n, f$, having the same set of solutions as the system $A\mathbf{x} = \mathbf{b}, f = \mathbf{c}^\mathsf{T}\mathbf{x}$, and written as

$$\begin{array}{rcccc} f & = & f_0 & + & \mathbf{r}^\mathsf{T}\mathbf{x}_N \\ \hline \mathbf{x}_B & = & \mathbf{p} & + & Q\,\mathbf{x}_N \end{array}$$

where $\mathbf{x}_B$ is the vector of basis variables, $N = \{1, \ldots, n\} - B$, $\mathbf{x}_N$ is the vector of non-basis variables, $\mathbf{r} \in \mathbb{R}^{n-m}$, $\mathbf{p} \in \mathbb{R}^m$, and $Q \in \mathbb{R}^{m\times n-m}$.

Since we started from a feasible basis solution B in our definition of a tableaux, clearly $\mathbf{p} \geq \mathbf{0}$. In order to verify that this definition is sound, it suffices to note that $\mathbf{b} = A\mathbf{x} = A_B\mathbf{x}_B + A_N\mathbf{x}_N$, and A_B is non-singular, so there exists an inverse matrix A_B^{-1}, implying that $\mathbf{x}_B = A_B^{-1}\mathbf{b} - A_B^{-1}A_N\mathbf{x}_N$. We can formulate this musings in the following lemma, which can be proven in detail by a careful reader.

Lemma 1.5. *To every feasible basic solution B of the program from the definition 1.4 there is exactly one tableaux $\mathcal{T}(B)$ satisfying*

$$\mathbf{p} = A_B^{-1}\mathbf{b} \qquad Q = -A_B^{-1}A_N \qquad f_0 = \mathbf{c}_B^\mathsf{T}A_B^{-1}\mathbf{b} \qquad \mathbf{r} = \mathbf{c}_N - (\mathbf{c}_B^\mathsf{T}A_B^{-1}A_N)^\mathsf{T}.$$

The ideas we introduced in our example lead us to

Claim 1.6. Let B be the basis of a feasible solution and let $\mathbf{r} \leq \mathbf{0}$ in $\mathcal{T}(B)$. Then f_0 is the maximal value of the given prolgram.

In order to conclude the description of the simplex algorithm we need to define the pivoting step: it consists of selecting some non-basic variable that occurs in $\mathbf{r}^\mathsf{T}\mathbf{x}_N$ with positive coefficient, increasing it as much as possible while keeping the basis variables non-negative, and changing the basis appropriately. Let us denote $B = \{\beta_1, \ldots, \beta_m\}$ so that $\beta_1 < \beta_2 < \cdots < \beta_m$, and, similarly, $N = \{\mu_1, \ldots, \mu_{n-m}\}$ where $\mu_1 < \mu_2 < \cdots < \mu_{n-m}$. Using this notation, the tableaux can be written as

$$\begin{array}{lcll} f & = & f_0 & + \sum_{j=1}^{n-m} r_j x_{\mu_j} \\ \hline x_{\beta_1} & = & p_1 & + \sum_{j=1}^{n-m} q_{1,j} x_{\mu_j} \\ \vdots & & \vdots & \\ x_{\beta_m} & = & p_m & + \sum_{j=1}^{n-m} q_{m,j} x_{\mu_j} \end{array} \tag{10}$$

Any variable x_{μ_j} with $r_j > 0$ can be chosen as pivot. If $q_{i,j} > 0$, the i-th row poses no restrictions on x_{μ_j}; oterwise, it must hold $p_i + q_{i,j} x_{\mu_j} > 0$. This leads us to the following definition:

Definition 1.7. Let us have a linear program in the normal form, and a basis B corresponding to some feasible solution. Let $\mathcal{T}(B)$ be written as in (10), and let $r_e > 0$ for some e. Denote

$$s := \min_{i=1,\ldots,m} \left\{ -\frac{p_i}{q_{i,e}} \mid q_{i,e} < 0 \right\}.$$

The **pivoting step** on the variable x_{μ_e} changes the basis B to a basis

$$B' := (B - \{\beta_\ell\}) \cup \{\mu_e\},$$

where β_ℓ is an arbitrary index for which $q_{\ell,e} < 0$, and $-\frac{p_\ell}{q_{\ell,e}} = s$.

The reader can easily check that B' is again a feasible basis. The simplex algorithm starts from some feasible solution with basis B_0 and applies the pivoting steps until possible. According to the claim 1.6, when a basis B for which $\mathbf{r} \leq \mathbf{0}$ is reached, the algorithm has found an optimal solution. In order to argue the correctness of the simplex method, we need to address three issues: how to find B_0, what to do when there is no pivoting step available, and finally show that the algorithm always stops in finite time.

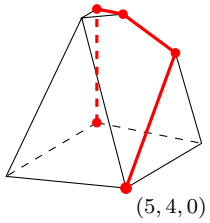
What if no pivoting step is available

The definition of a pivoting step (Definitiona 1.7) requires that for a pivot x_{μ_e} it holds $r_e > 0$. If there is no such x_{μ_e}, following Claim 1.6 the solution is optimal. Further, a valid pivoting step requires that the pivot replaces in the basis a variable x_{β_ℓ}, for qhich $q_{\ell,e} < 0$. If there is no such ℓ, i.e., if $q_{\ell,e} \geq 0$ for all ℓ, it means that no row poses any restriction on the increasing of the variable x_{μ_e}. Increasing x_{μ_e} increases also the value of the utility function, and hence the program has no finite maximum.

How to avoid infinite loops

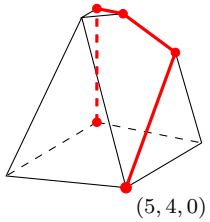
In the introductory example every pivoting step increased the value of the utility function. If this could be guaranteed in every step it is easy to see that the algorithm will terminate in finite time: There are only finitely many basic solutions, and due to the increasing of the value of f they can not repeat. However, reconsider the example, and suppose that in step (8), instead of replacing z, the pivot x replaces s_2 in the basis. Instead of the tableaux (9) we get

$$
\begin{array}{rrrrr}
f = & 9 & + s_1 & - \frac{1}{2}s_2 & - \frac{1}{2}s_3 \\
\hline
x = & 5 & + \frac{7}{3}s_1 & - \frac{5}{6}s_2 & - \frac{1}{2}s_3 \\
y = & 4 & + \frac{2}{3}s_1 & - \frac{1}{6}s_2 & - \frac{1}{2}s_3 \\
z = & & - 2s_1 & + \frac{1}{2}s_2 & + \frac{1}{2}s_3 \\
s_4 = & 4 & + 2s_1 & - \frac{1}{2}s_2 & - \frac{1}{2}s_3
\end{array}
\qquad (11)
$$



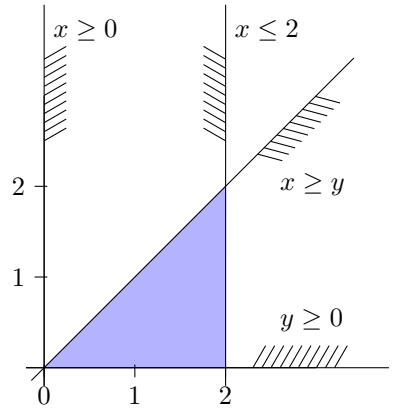
The tableaux ((9) represents the basis $\{x, y, s_2, s_4\}$, and the tbleaux (11) the basis $\{x, y, z, s_4\}$; both of them have the same basic solution $(5, 4, 0, 0, 0, 0, 4)$. However, from (11) it is not evident that the optimal solution has been already found, and an additional pivoting step with the pivot s_1 is needed. In this step the variable z does not allow to increase s_1, so this pivoting step is "void": it only replaces s_1 with z in the basis without changing the value of the function f. After this step, we obtain a tableaux

$$
\begin{array}{rrrrr}
f = & 9 & - \frac{1}{2}z & - \frac{1}{4}s_2 & - \frac{1}{4}s_3 \\
\hline
x = & 5 & - \frac{7}{6}z & - \frac{1}{4}s_2 & + \frac{1}{12}s_3 \\
y = & 4 & - \frac{1}{3}z & & - \frac{1}{3}s_3 \\
s_1 = & & - \frac{1}{2}z & + \frac{1}{4}s_2 & + \frac{1}{4}s_3 \\
s_4 = & 4 & - z & &
\end{array}
\qquad (12)
$$



from which the optimality can be inferred. The degenerate step destroyed our original halting argument: there is no way to ensure that the value of f would increase in every step. Worse even, the neccesity to perform a void step may not arise only as the last step of the algorithm, when the solution is already optimal, as can be seen from the following simple example (due to [11]):

$$
\begin{array}{lrrl}
\text{maximize} & & y & =: f(x, y) \\
\text{subject to} & - x & + y & \leq 0 \\
& x & & \leq 2 \\
& & x, y & \geq 0
\end{array}
$$



After introducing slackness variables s_1, s_2, we obtain a tableaux

$$
\begin{array}{rrrr}
f = & & & y \\
\hline
s_1 = & & x & -y \\
s_2 = & 2 & - x &
\end{array}
$$

for the basis $\{s_1, s_2\}$, with the utility value $f = 0$. The only way to continue (and reach the optimal solution with utility value 2), is to perofrm a void pivoting step, in which y replaces s_1 in

the basis:

$$\begin{array}{rrl} f = & & x \quad -s_1 \\ \hline y = & & x \quad -s_1 \\ s_2 = & 2 & -x \end{array}$$

A similar situation occurs quite often.

Definition 1.8. A *void pivoting step* of a simplex algorithm is a step, in which the basis B is transformed to B' without changing the corresponding basic solution.

Void steps cannot be avoided, and as the following example from [4] shows, the algorithm may loop infinitely if we are not careful enough. Consider the following tableaux:

$$\begin{array}{rrlll} f = & 10x_1 & -57x_2 & -9x_3 & -24x_4 \\ \hline x_5 = & -0.5x_1 & +5.5x_2 & +2.5x_3 & -9x_4 \\ x_6 = & -0.5x_1 & +1.5x_2 & +0.5x_3 & -x_4 \\ x_7 = 1 & -x_1 & & & \end{array} \tag{13}$$

for the basis $\{x_5, x_6, x_7\}$. Suppose that this particular simplex algorithm alway selects as pivot the variable x_{μ_e} with the maximum value r_e. If the pivoting step zeroes out several basic variables, the algorithm selects the one with the minimal index. We leave as an exercise to verify that the algorithm successively visits bases $\{x_1, x_6, x_7\}$, $\{x_1, x_2, x_7\}$, $\{x_2, x_3, x_7\}$, $\{x_3, x_4, x_7\}$, $\{x_4, x_5, x_7\}$, finally reaching the basis $\{x_5, x_6, x_7\}$ again. Since we have a cycle consistnig of void steps, the algorithm will loop forever.

Since we cannot hope to prove the convergence of the simplex method for an arbitrary choice of the pivot, we have to fix some algorithm to select the pivot, and try to prove convergence of this algorithm. There are many rules for the pivot selection that address the convergence issue from different approaches. Here we shall use the so called *Bland's anticycleing rule*, invented in [3]

Definition 1.9. The *Bland's anticycling rule* selects the pivot variable x_{μ_e}, such that μ_e is the smallest one for which $r_e > 0$. If the pivoting step zeroes out several variables, the variable x_{β_ℓ} with the snmallest index β_ℓ leaves the basis.

Theorem 1.10. *The simplex algorithm with Bland's anticycling rule always terminates and finds an optimal solution.*

Proof: We already know that if a simplex algorithm terminates the obtained solution is optimal. First note that the only reason that may prevent a simplex algorithm from terminating is a cycle consisting solely of void steps. Indeed, if the algorithm never terminates, it must enter the same base B infinitely many times. Since any non-void step increases the value of f, and the void steps don't alter it, all steps between two visits of B must be void. Hence, to prove the theorem it is sufficient to show that a cycle of void steps cannot occur.

Let us suppose, for the sake of contradiction, that the algorithm has a tableaux for a base B_0, and succesivelly enters tableaux with bases $B_1, B_2, \ldots, B_k = B_0$, where all pivoting steps are void (i.e., all bases $B_1, \ldots, B_k$ have the same basic solution). A variable x_i will be called *volatile* if it appears in some basis B_j, but doesn't appear in another basis $B_{j'}$. Let t be the maximal index such that x_t is volatile. Since x_t is volatile, there is a pivoting step where x_t leaves the basis, i.e., for some j it holds $x_t \in B_j$ and $x_t \notin B_{j+1}$. So there mus exist another volatile variable x_s that replaces x_t in the basis: $x_s \notin B_j$ and $x_s \in B_{j+1}$. At the same time, however, x_t must return to the basis during the sequence $B_j, \ldots, B_k, B_1, B_2, \ldots B_{j+1}$, so there is a basis $B_{j^\star}$ such that $x_t \notin B_{j^\star}$ and $x_t \in B_{j^\star+1}$. Let $\mathcal{T}(B_j)$ is as follows:

$$\begin{array}{lcll} f & = & f_0 & + \sum\limits_{k \notin B_j} r_k x_k \\ \hline x_{\beta_1} & = & p_{\beta_1} & + \sum\limits_{k \in B_j} q_{\beta_1,k} x_k \\ \vdots & & \vdots & \\ x_{\beta_m} & = & p_{\beta_m} & + \sum\limits_{k \in B_j} q_{\beta_m,k} x_k \end{array} \tag{14}$$

where $B_j = \{\beta_1, \ldots, \beta_m\}$. Since all pivonting steps of the cycle are void by assumption, the two basis B_j, and $B_{j^\star}$ have the same basic solution (i.e., the values of all variables are the same). So we can write

$$f = f_0 + \sum_{k=1}^{n} r_k^\star x_k \tag{15}$$

where

$$r_k^\star = \begin{cases} 0 & \text{if } k \in B_{j^\star} \\ \text{coefficient } r \text{ by } x_k \text{ in the tableaux } \mathcal{T}(B_{j^\star}) & \text{otherwise} \end{cases}$$

Since $\mathcal{T}(B_{j^\star})$, and in particular the equation (15) was derived by equivalent transformations from the system (14), all solutions of the system (14) fullfil (15). Let us now consider a solution (not necessarily basic or even feasible) of the system (14): choose an arbitrary y and set

$$x_i = \begin{cases} y & \text{if } i = s \\ 0 & \text{if } i \neq s \text{ and } i \notin B_j \\ p_i + q_{i,s} y & \text{if } i \in B_j \end{cases}$$

It is easy to see that thus constructed $\mathbf{x}$ is a solution of the system (14)[5] Since our $\mathbf{x}$ satisfies both (14) and (15), be expressing f in both of them we get

$$f_0 + r_s y = f_0 + \sum_{k=1}^{n} r_k^\star x_k = f_0 + r_s^\star y + \sum_{k \in B_j} r_k^\star (p_k + q_{k,s} y)$$

which yields

$$\left(r_s - r_s^\star - \sum_{k \in B_j} r_k^\star q_{k,s} \right) y = \sum_{k \in B_j} r_k^\star b_k.$$

Since this equality holds for any y, and the right-hand side does not depend on y, it follows that

$$r_s - r_s^\star - \sum_{k \in B_j} r_k^\star q_{k,s} = 0.$$

Since x_s was selected pivot in the basis B_j, itmust hold $r_s > 0$. In $B_{j^\star}$ the pivoting variable was x_t, and since $t > s$, it is $r_s^\star \leq 0$. Further, from $r_s - r_s^\star > 0$ it follows that there is some $z \in B_j$ for which

$$r_z^\star q_{z,s} > 0.$$

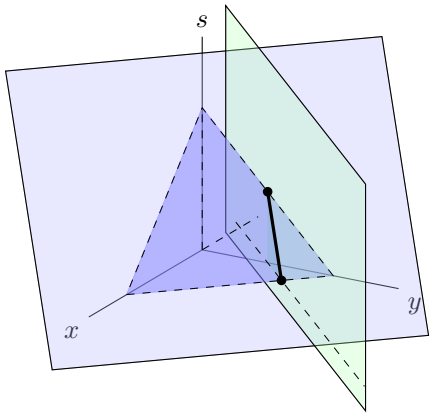
The variable x_z is basic in B_j, and at the same time $z \notin B_{j^\star}$, because $r_z^\star \neq 0$. Hence x_z is volatile and by the thefinition of t it holds $z \leq t$. Moreover, $z \neq t$: since x_t left the basis B_j during a pivoting step, it holds $q_{t,s} < 0$ and $r_t^\star q_{t,s} < 0$ (since x_t was pivot in $B_{j^\star}$). Now we see that $z < t$. But x_z was not pivot in $B_{j^\star}$, so $r_z^\star \leq 0$. Since $r_z^\star q_z, s > 0$, it must be $q_{z,s} < 0$.
From the fact that all basic solutions in the void cycle are the same, and $z \notin B_{j^\star}$ it folllows that $x_z = 0$ in both basic solutions B_j and $B_{j^\star}$. Since $z \in B_j$, it holds $p_z = 0$. However, that means the x_z could have left the basis B_j, but x_t was selected instead, contradicting the Bland's anticcling rule. □

[5]It is similar to performing a pivoting step with variable x_s by the amount of y without caring about keeping the basic variables nonnegative.

How to start

The last detail that needs to be clarified is how the algorithm starts. So far we supposed that we start with a basis B_0 with a feasible basic solution. In the introductory example the starting basis B_0 in (5) was chosen to consist of the slackness variables s_1, s_2, s_3, s_4. This approach works for programs of the form $\max_{\mathbf{x}\in\mathbb{R}^7}\left\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \le \mathbf{b},\ \mathbf{x} \ge 0\right\}$ with slackness variables added, if $\mathbf{b} \ge \mathbf{0}$. What about the other programs? Consider the following program:

$$\begin{array}{lrrrrl} \text{maximize} & 4x & & - z & =: f(x,y,z) \\ \text{subject to} & x & + y & + z & = 4 \\ & x & - y & & = -2 \\ & & & x,y,z & \ge 0 \end{array} \tag{16}$$



The feasible solutions form the line segment $(1,3,0) - (0,2,2)$. In order to start the simplex algorithm we need some feasible solution. For a given point (x,y,z) denote $p_1 := 4 - x - y - z$; p_1 describes "how much" is the first equality[6] violated. Similarly, let $p_2 := 2 + x - y$ (note the equation has been rewritten in a way that the absolute term is non-negative). Finding a feasible solution means to find a point (x,y,z), such that $p_1 = p_2 = 0$, i.e., $p_1, p_2 \ge 0$, and $p_1 + p_2 = 0$. It is easy to see that the program (16) has a feasible solution exactly when the program

$$\begin{array}{lrrrrrrl} \text{maximize} & - p_1 & - p_2 & & & & & \\ \text{subject to} & p_1 & & + x & + y & + z & = 4 \\ & & p_2 & - x & + y & & = 2 \\ & & & & & x,y,z,p_1,p_2 & \ge 0 \end{array} \tag{17}$$

has a feasible solution with value 0. In this case it is easy to check that $\{p_1, p_2\}$ is the basis of the feasible solution. So we can use the simplex algorithm to find optimum of the second program, and use it as a starting point of the original program.
This approach can be used always. Let us consider a linear program in the normal form

$$\max_{\mathbf{x}\in\mathbb{R}^n}\left\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \ge \mathbf{0}\right\}$$

First make sure that $\mathbf{b} \ge 0$: if $b_i < 0$ for some i, premultiply the corresponding equation by -1. Introduce new variables $x_{n_1}, \ldots, x_{n+m}$ and set up the auxiliary program

$$\max_{\tilde{\mathbf{x}}\in\mathbb{R}^{n+m}}\left\{-x_{n+1} - \ldots - x_{n+m} \mid \tilde{A}\tilde{\mathbf{x}} = \mathbf{b},\ \tilde{\mathbf{x}} \ge \mathbf{0}\right\}$$

where $\tilde{A} = (A \mid I_m)$ is obtained from A by augmenting with identity matrix of dimensions $m \times m$. Since $\mathbf{b} \ge 0$, $\{x_{n+1}, \ldots, x_{n+m}\}$ form a basis of a feasible solution, and simplex algorithm can be used to compute the optimum. If the value optimum is 0, we got a feasible solution of the original program. On the other hand, for any feasible solution of the original program there is a solution of the auxiliary program with value 0. Hence, if the optimum of the auxiliary program is not 0, the original program had no feasible solution.

Exercise. Implement the simplex algorithm with Bland's anticycling rule.

[6] It is *not* the distance from (x,y,z) to the plane $x + y + z = 4$.

1.3 Complexity of the simplex algorithm

In the last chapter we introduced the simplex method for solving linear programs in a more efficient way than to check all the vertices of the polyhedron of feasible solutions. We showed that the simple method with the Bland's anticycling rule always terminates. Now the question is how efficient it is. Surprisingly enough, inspite of the fact that the simplex algorithm works very well in practice, its worst-case complexity is exponential as we show in a minute.

Before doing so, let us review some basic facts, as the subtle details will play a significant role here. When analyzing the complexity of an algorithm, we do so with respect to some parameter that is, in general, a part of the definition of the problem. When e.g. we say that the algorithm has complexity $O(n^2)$, we mean that there is a constant c and some n_0 such that for any input with the parameter $n > n_0$ the running time of the algorithm is at most cn^2. A natural and universally available parameter is the length of the input: the definition of the problem always contains the description of how the input is represented as a binary string and the length (number of bits) of this string is a good complexity parameter. Sometimes (and actually quite often), however, we use different parameters that are more natural for the problem: e.g. when analyzing sorting algorithms, the parameter is usually tne number of the numbers to be sorted, although the length of the input depends also on the sizes of the numbers. Similarly, in graph algorithms we sometimes use the number of vertices as the parameter, although $\Omega(n^2)$ bits are needed to represent a graph with n vertices [7]

The instance of a linear program with n variables and m constraints consists of two vectors of real numbers $\mathbf{c}$ and $\mathbf{b}$, and a matrix $A \in \mathbb{R}^{m \times n}$. Natural complexity parameters would be m, n, or the length of the input, where in the latter case we need to specify the encoding of the reals, and accept the fact that if we want to have finite inputs, we cannot encode all real numbers.

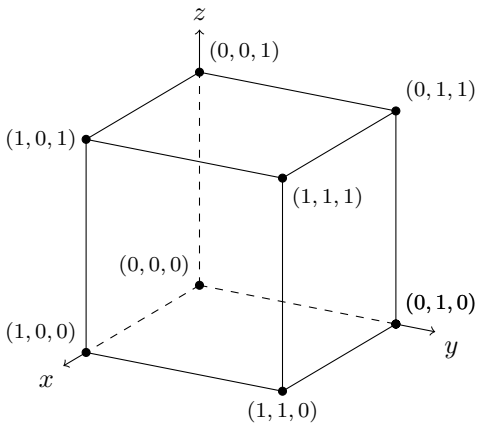
These details should be kept in mind, although we are not really worrying about them now: we shall construct an input with n variables and $2n$ constraints where the simplex algorithm performs $\Omega(2^n)$ iterations. Moreover, we shall use only numbers with short description (actually, only the numbers $\{\pm 1, \pm\frac{1}{4}, 0\}$), so we show exponential complexity in any of the abovementioned parameters.

Consider the simplex algorithm with the Bland's anticycleing rule (for a number of other pivot-choosing strategies there are similar proofs available). We construct, for each n, an input instance with n variables and $2n$ constraints in such a way that the polyhedron of feasible solutions has 2^n vertices and the simplex algorithm visits all of them. In our instance the goal is to maximize the variable x_n, and the constraints form a skewed cube. Let us begin by creating an n-dimensional cube using $2n$ constraints:

$$\begin{array}{rcl} 0 \leq & x_1 & \leq 1 \\ 0 \leq & x_2 & \leq 1 \\ & \cdots & \\ 0 \leq & x_n & \leq 1 \end{array}$$

In three dimensions the polyhedron of the feasible solutions is a cube:

[7] It is sufficient to note that a labeled graph may contain up to $\binom{n}{2}$, and each of then may be present in the graph or not, so there are $2^{\binom{n}{2}}$ (labelled) graphs, and so using the pidgeon-hole principle we need at least $\binom{n}{2}$ bits to identify them.



Now we want to shift the vertices such that there is a long increasing spiral. Choose some $\varepsilon < \frac{1}{2}$ and define constraints:

$$\begin{aligned} \varepsilon \leq \quad & x_1 \quad && \leq 1 \\ \varepsilon x_1 \leq \quad & x_2 \quad && \leq 1 - \varepsilon x_1 \\ & \cdots \\ \varepsilon x_{n-1} \leq \quad & x_n \quad && \leq 1 - \varepsilon x_{n-1} \end{aligned}$$

Transform the program into the normal form by introducing slackness variables r_i, s_i so that the constraints are expressed as equalities. This yields a program:

$$\begin{aligned} \text{maximize} \quad & x_n \\ \text{subject to} \quad & x_1 - r_1 && = \varepsilon \\ & x_1 + s_1 && = 1 \\ & x_2 - \varepsilon x_1 - r_2 && = 0 \\ & x_2 + \varepsilon x_1 + s_2 && = 1 \\ & \cdots && \cdots \\ & x_n - \varepsilon x_{n-1} - r_n && = 0 \\ & x_n + \varepsilon x_{n-1} + s_n && = 1 \end{aligned} \tag{18}$$

where all variables are non-negative. How do the feasible basic solutions look like? Thanks to the slackness variables, the constraints are linearly independent (each constraint contains a variable that does not appear anywhere else), so the basis has $2n$ elements. Moreover, $r_1 + s_1 = 1 - \varepsilon$, and for each pair of variables r_i, s_i where $i > 1$ it holds $r_i + s_i = 1 - 2\varepsilon x_{i-1} > 0$. Hence, it cannot hold $r_i = s_i = 0$, and so each basis must contain at least one of the variables r_i, s_i. Still more, all $x_i > 0$, and so they appear in every basis. Each basis B is thus uniquely characterized by a set $R_B \subseteq \{1, \ldots, d\}$: basic variables are

$$\{x_1, \ldots, x_n\} \cup \bigcup_{i \in R_B} \{r_i\} \cup \bigcup_{i \notin R_B} \{s_i\}$$

At the same time, the following claim is obvious:

Claim 1.11. Each pivoting step is uniquely characterized by an index i. The step changes the membership in the basis for variables r_i and s_i.

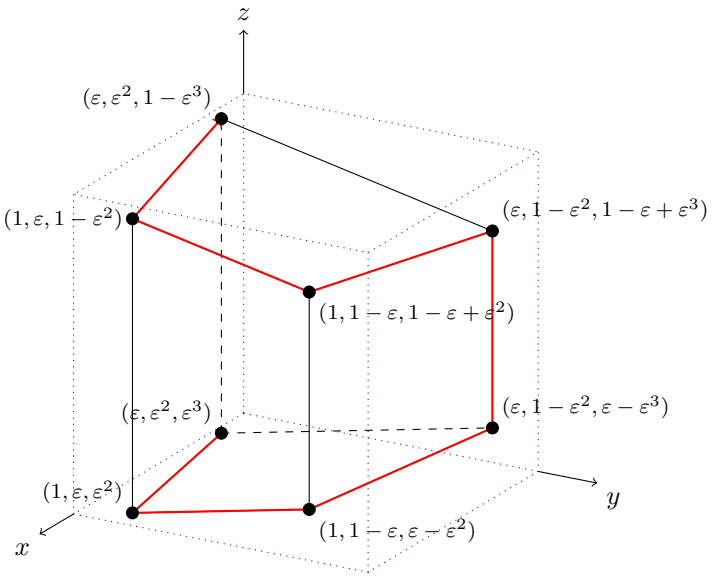
To illustrate the program, let $n = 3$. In the matrix notation, the program is

$$\max\{x_3 \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0\}$$

where

$$A = \begin{pmatrix} 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -\varepsilon & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ \varepsilon & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & -\varepsilon & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & \varepsilon & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ r_1 \\ s_1 \\ r_2 \\ s_2 \\ r_3 \\ s_3 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} \varepsilon \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

The matrix A has rank 6, and $R_B \subseteq \{1, 2, 3\}$. So there are 6 basic solutions that form a skewed cube:



The red edges correspond to an increasing path starting in the solution with the set $R_{B_0} = \emptyset$, and going through all vertices. The path always moves in the first dimension in which the utility function grows. In order to generalize this example to n dimensions we need to be able to reason about the pivoting steps of the algorithm with Bland's anticycling rule. This can be done using the claim 1.11 and the following lemma:

Lemma 1.12. *Given a basis B of the program (18), with the corresponding tableaux $\mathcal{T}(B)$, let the utility function in $\mathcal{T}(B)$ be expressed in terms of non-basic variables as $x_n = c_0 + c_1 v_1 + c_2 v_2 + \cdots + c_n v_n$, where v_i is either r_i or s_i, and c_i is the corresponding coefficient. Then c_i is positive exactly if the number of basic variables r_j for $j \geq i$ is even, i.e.,*

$$|\{j \mid j \in R_B,\ j \geq i\}| \equiv 0 \pmod 2$$

Proof: The proof is done by induction on the dimension of the problem n. For $n = 1$, if r_1 is in the basis we get $x_1 = 1 - s_1$ and c_1 is negative, and if r_1 is not in the basis, then $x_1 = \varepsilon + r_1$ and c_1 is positive.
Now suppose the statement holds for $n - 1$. If $n \in R_B$, the expression for x_n must contain s_n, and hence must be of the form $x_n = 1 - s_n - \varepsilon(c'_0 + c'_1 v'_1 + \cdots + c'_{n-1} v'_{n-1})$, where $x_{n-1} = c'_0 + c'_1 v'_1 + \cdots + c'_{n-1} v'_{n-1}$ is expressed in the non-basic variables $v_1, \ldots, v_{n-1}$. The result follows by expanding and applying the induction hypothesis. If $n \notin R_B$, the approach is similar using the equality $x_n = r_n + \varepsilon x_{n-1}$. □

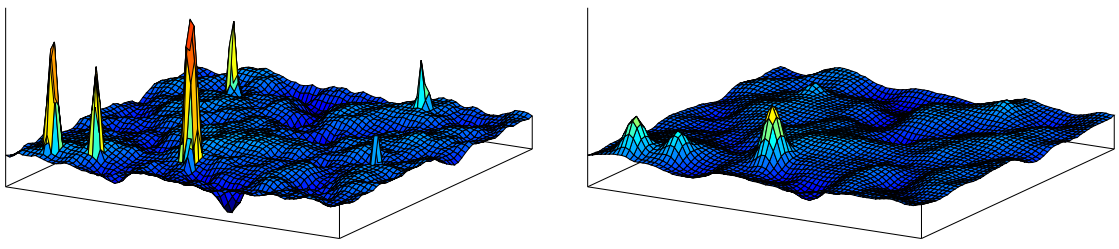
Now we can show that the simplex algorithm visits all vertices:

Theorem 1.13. *Let $i \in \{1, \ldots, n\}$ and $R \subseteq \{i+1, \ldots, n\}$. If the simplex algorithm with the Bland's anticycling rule starts in a basis B_0 with $R_{B_0} = R$ (resp. $R_{B_0} = \{i\} \cup R$) and $|R|$ is even (resp. $|R|$ is odd), it visits all the bases of the form $R' \cup R$ where $R' \subseteq \{1, \ldots, i\}$, and terminates in a basis B_1 with $R_{B_1} = \{i\} \cup R$ (resp. $R_{B_1} = R$).*

Proof: The proof is by induction on i. If $i = 1$ then in both cases ($R_{B_0} = R$, $|R|$ even, and $R_{B_0} = \{1\} \cup R$, $|R|$ odd) Lemma 1.12 asserts that the coefficient at v_1 is positive, and the algorithm performs a pivoting step with index 1.
Now let the statement hold for $i-1$. There are two cases. First, let $R_{B_0} = R$ with $|R|$ even. Since $R \subseteq \{i, \ldots, n\}$ we can use the induction hypothesis: the algorithm visits all bases of the form $R' \cup R$ where $R' \subseteq \{1, \ldots, i-1\}$, and terminates in $\{i-1\} \cup R$. Since $|R|$ is even, according to Lemma 1.12 the algorithm enters the basis $\{i-1, i\} \cup R$. The induction hypothesis for $i-1$ and an odd set $\{i\} \cup R$ yields the result.
The second case where $R_{B_0} = \{i\} \cup R$, and $|R|$ is odd can be taken care of in a similar fashion, and we leave it to the reader. □

Corollary 1.14. *The simplex algorithm with Bland's anticycling rule performs exponentially many iterarions on the program (18).*

Now we can see that the simplex algorithm is exponential in the worst case regardless of whether the complexity parameter is the number of variables, the number of constraints, or the length of the input. How come then it performs so well in practical situations? One possible attempt in explaining it, is to analyze the average case. However, there is an immediate problem – how to define the "average" case. Indeed, there are results stating that the simplex algorithm performs polynomially many iterations in the average case, where the "average case" is the expected number of iterations if both the matrix A, and the vectors $\mathbf{c}$, $\mathbf{b}$ are selected at random from a given probability distribution. However, this is not very satisfying answer: the average case defined this way is actually very far from a "typical" case that is solved in practice, where the (matrices of the) programs usually exhibit a great deal of structure. An interesting explanation was found using the notion of *smoothed complexity* that combines the worst case and the average case analysis. It considers all instances (i.e., worst-case), but for each instance we analyze the expected time over all instances that are "close" enough - i.e., those that can be obtained using some small perturbation. Spielman and Teng [13] showed that the smoothed complexity of the simplex algorithm is polynomial. If we imagine the space of all solutions as a plane, the comlexity of the simplex algorithm looks like the picture on the left – it is usually polynomial, with sparsely spaced "bad" instances.



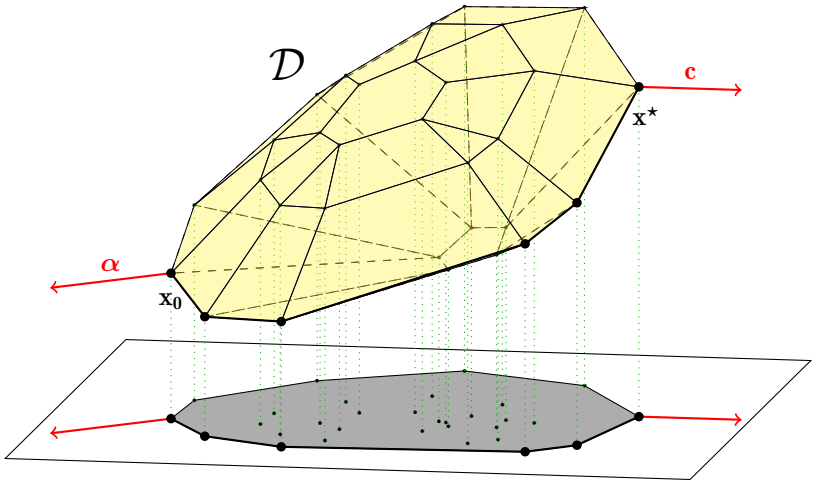
After averaging over small neighbourhoods, the bad instances are smoothed as on the right picture. This explains why the simplex method can be considered efficient even if it is not polynomial in the strict sense: in order to come up with an superpolynomial-time instance, the values must be

set very precisely; any small random perturbation of the values yields an instance with polynomial solution.

The result of Spielman and Teng is considered a breakthrough an on the first (and second, and third) sight may seem as magic. It is out of the scope of this text to present the complete proof, but we would like to finish this chapter with a simple visual reasoning why it might actually work. In the previous text we introduced the simplex algorithm with the Bland's anticycling rule. This time, however, the analysis is done using a different rule, called *shadow tracing*. Let's have a linear program in the form

$$\max_{\mathbf{x}\in\mathbb{R}^n}\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0\}$$

The feasible soluitons form a polyhedron $\mathcal{D}$ in an n-dimensional space, and to find an optimal basic solution means to find a vertex $\mathcal{D}$ that is furthest in the direction of the vector $\mathbf{c}$. Since we know that in two dimensions, this prolblem is easy, we can reason as follows: let's have a starting basic solution $\mathbf{x_0}$. Since it is a vertex of $\mathcal{D}$, there is a vector **alpha** such that the vertex $\mathbf{x_0}$ maximizes the value $\boldsymbol{\alpha}^\mathsf{T}\mathbf{x}$ ($\mathbf{x_0}$ is farthest in the direction of $\boldsymbol{\alpha}$). Take the (2-dimensional) plane spanned by the vectors $\boldsymbol{\alpha}$ and $\mathbf{c}$, and project every vertex of $\mathcal{D}$ into it. We get a set of points in the plane, and their convex hull is the *shadow* that $\mathcal{D}$ casts into the plane.



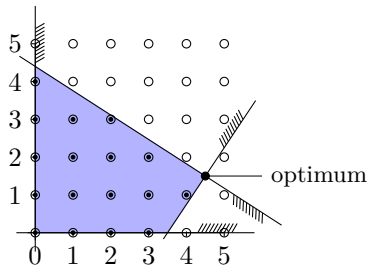
It is not difficult to see that the projections of both $\mathbf{x_0}$, and the optimal solution $\mathbf{x}^\star$ are located on the boundary of the shadow. A simplex algorithm with the shadow vertex rule always selects during a pivoting step a basic solution whose projection is on the boundary of the shadow. This can be tested e.g. by considering all possible pivoting steps, and comparing their projections (although more efficient ways exist). The number of iterations is clearly upper-bounded by the number of the number of points that project to the shadow boundary. An important step in the proof is to transform the program to the form where the constraints are of the form $\mathbf{a_i}^\mathsf{T}\mathbf{x} \leq 1$; in this case there is a polar description of $\mathcal{D}$, and it can be observed that the number of the vertives of the shadow is at most the number of vertices of a polynome $\mathcal{M}$, which is formed as an intersection of the conver hull of points $\mathbf{a_1}, \ldots, \mathbf{a_n}$ with the plane spanned by vectors $\boldsymbol{\alpha}$, $\mathbf{c}$. The core of the proof is a geometric statemet: given n points in a d-dimensional space, if each of them is translated by a random vector with normal distribution, then in the expected case the corresponding polymone $\mathcal{M}$ has at most $poly(n, d, \frac{1}{\sigma})$ points, where σ^2 is the standard deviation.

It is still a long way to the actual proof; we only wanted to hint on the direction to make the result more believable. Interested readers are invited to look into the paper [Spielman, Teng]. or the lecture notes `http://www.cs.yale.edu/homes/spielman/BAP/`

2 Rounding of linear programs

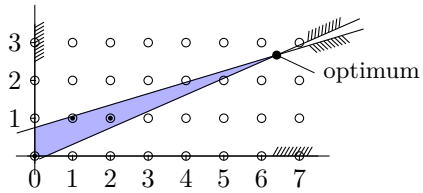
2.1 Integer programs and relaxation: the good, the bad, and the ugly

In the previous parts we showed how the linear programming problem can be solved efficiently. In solving real-life problems, we are often interested only in solutions where some of the variables involved are forced to be integers. In the first motivation example we found out that the optimal solution is to buy $2\frac{2}{3}$ dl of green stuff. This would probably not be feasible, since drinks are usually served in multiples of some fixed volume.



Feasible solutions of a linear program and its integral restriction.

The first intuition might suggest that this is just a small technical issue: after all, we can always find the optimal solution of the continuous version, round it in some way and get, if not the optimal solution, then surely something reasonably close to the optimum. On a second thought one starts wondering that it might not be so easy: if a linear program used to find out whether to travel by air or by train recommends to buy half air and half train ticket, the rounding is as complex as the entire solution. The following picture shows that the closest integer solution might be quite far from the optimum, and it is not obvious how to find it.



A linear program with some variables constrained to be integers is called an integral linear program (ILP):

Definition 2.1. *An integral linear program* (ILP) in a normal form is a linear program

$$\max_{\mathbf{x}\in\mathbb{R}^n} \left\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\right\}$$

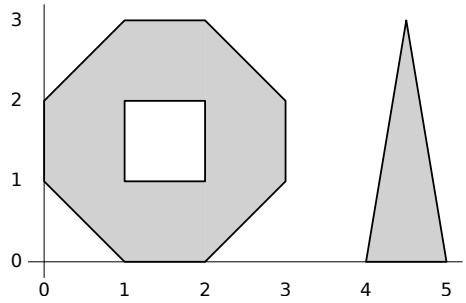
with additional constraints of the form $x_i \in \mathbb{Z}$.

A reader with some background in the complexity theory is surely not surprised by the fact that the integrality constraints significantly increase the expressive power of linear programs. When formulating problems in terms of linear programs important role is played by so called *indicator*

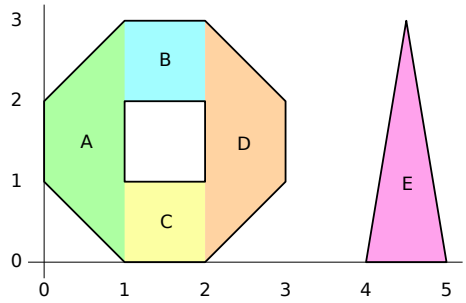
variables: a set of variables $\delta_1, \ldots, \delta_k \in \mathbb{Z}$ with constraints

$$\delta_1 + \delta_2 + \cdots + \delta_k = 1$$
$$\delta_i \geq 0 \text{ pre } i = 1 \ldots k$$

have the property that in any feasible solution is exactly one $\delta_i = 1$, and the rest of them is zero. Indicator variables allow to express the selection of one alternative among several possibilities, and thus e.g. describe complex non-convex domains. As a demonstration, let us suppose that we want to maximize the function $f(x, y) = x + y$ over the following domain $\mathcal{D}$:



The domain $\mathcal{D}$ can be decomposed into 5 parts:



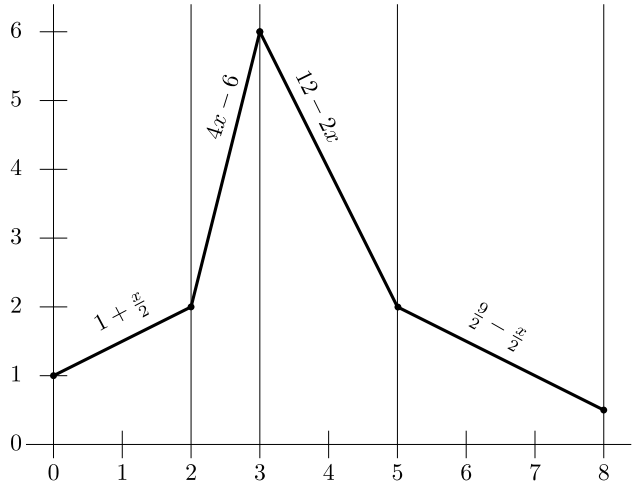
such that each of them can be represented by linear constraints:

$x \geq 0$	$x \geq 1$	$x \geq 1$	$x \geq 2$	$x \geq 4$
$x \leq 1$	$x \leq 2$	$x \leq 2$	$x \leq 3$	$x \leq 5$
$y \geq 1 - x$	$y \geq 2$	$y \geq 0$	$y \geq x - 2$	$y \leq 6x - 24$
$y \leq 2 + x$	$y \leq 3$	$y \leq 1$	$y \leq 5 - x$	$y \leq -6x + 30$
				$y \geq 0$

Let us introduce indicator variables $\delta_1, \ldots, \delta_5 \in \mathbb{Z}$ and rewrite each set of constraints describing one part, such that if the corresponding indicator is zero they yield trivial constraints, and if the indicator is one they retain their original values. In this way we obtain an ILP: the task is to maximize $x + y$, subject to the constraints:

$x \geq 0$	$x \geq \delta_2$	$x \geq \delta_3$	$x \geq 2\delta_4$	$x \geq 4\delta_5$
$x \leq 5 - 4\delta_1$	$x \leq 5 - 3\delta_2$	$x \leq 5 - 3\delta_3$	$x \leq 5 - 2\delta_4$	$x \leq 5$
$y \geq \delta_1 - x$	$y \geq 2\delta_2$	$y \geq 0$	$y \geq -5 + 3\delta_4 + x$	$y \leq 3 - 27\delta_5 + 6x$
$y \leq 3 - \delta_1 + x$	$y \leq 3$	$y \leq 3 - 4\delta_3$	$y \leq 8 - 3\delta_4 - x$	$y \leq 33 - 3\delta_5 - 6x$
				$y \geq 0$

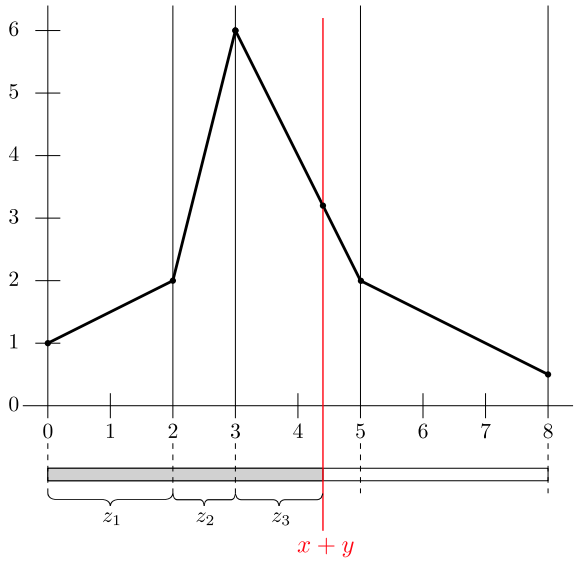
Another useful property of integer programs is the possibility to approximate the optimization of non-linear utility functions by piecewise-linear functions. Let us suppose that, in the previous example, instead of maximizing the function $x + y$, we want to maximize the function composed of linear pieces:



Let us introduce four variables z_1, z_2, z_2, z_3, z_4 that denote to what extent is the corresponding interval covered by the amount $x + y$:

$$x + y = z_1 + z_2 + z_3 + z_4$$

We need some additional constraints that ensure that the z_is will be "filled consecutively", i.e. if z_i is non-zero then z_{i-1} is at its maximum.



In order to do so, we add three binary variables $\alpha_1, \alpha_2, \alpha_3$, where $\alpha_i \in \mathbb{Z}$, $0 \leq \alpha_i \leq 1$ with the meaning that that the variable $\alpha_i \in \{0, 1\}$ indicates whether z_{i+1} is non-zero. The constraints:

$$\begin{aligned} 2\alpha_1 &\leq z_1 \leq 2 \\ \alpha_2 &\leq z_2 \leq \alpha_1 \\ 2\alpha_3 &\leq z_3 \leq 2\alpha_2 \\ 0 &\leq z_4 \leq 3\alpha_3 \end{aligned}$$

then ensure the required properties, e.g. if $\alpha_1 = 0$, then $0 \leq z_1 \leq 2$, but $z_2 = z_3 = z_4 = 0$. On the other hand, if $\alpha_1 = 1$, then $z_1 = 2$ and $\alpha_2 \leq z_2 \leq 1$. Using these variables, the utility function is easily expressed as

$$f(x, y, z_1, \ldots, z_4, \alpha_1, \ldots, \alpha_3, \delta_1, \ldots, \delta_5) = 1 + \frac{1}{2}z_1 + 4z_2 - 2z_3 - \frac{1}{2}z_4$$

The previous examples have hopefully supported the reader's intuition that ILP can express much more complicated problems than LP, and so the solution of the ILP should be harder than that of

LP. We show that it is indeed so: even the problem of determining whether there is any feasible solution to a given ILP is NP-complete (and so the author of this text does not expect a polynomial time-algorithm for solving the ILP will be found). For our proof, we shall need the SAT problem, so let's review its definition:

Definition 2.2. Given n logical variables $x_1, \ldots, x_n \in \{true, false\}$, a *literal* is either a variable x_i or its negation $\overline{x_i}$, a *clause* is a disjunction of literals $C = l_1 \vee l_2 \vee \cdots \vee l_{k_C}$, and a formula in a conjunctive normal form (CNF) is a conjunction of clauses $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$. The problem SAT is a decision problem where for a given CNF formula F, the task is to decide whether there exists a satisfying truth assignment, i.e. an assignment of values $\{true, false\}$ to variables $x_1, \ldots, x_n$ such that F is true.

A fundamental result in the complexity theory is the Cook-Lewin theorem stating that SAT is NP-complete, i.e. the existence of a polynomial-time algorithm for SAT would imply $P = NP$. Now we show that if there would exist a polynomial-time algorithm `A` deciding, for any ILP, whether there exists some feasible solution, there would be a polynomial-time algorithm `A'` solving SAT. Let us suppose we have given the algorithm `A`, and we shall describe the algorithm `A'`. Let the input formula F consits of clauses $C_1, \ldots, C_m$. Denote by P_i the set of variables that appear in C_i as positive literals, and N_i the set of variables those variables that appear in C_i in negated literals. E.g. for the clause $C_i = (x_1 \vee \overline{x_2} \vee x_3)$ we have $P_i = \{x_1, x_3\}$ and $N_i = \{x_2\}$. Construct an ILP I such that to each logical variable x_i there will be an equally denoted variable $x_i \in \mathbb{Z}$ constrained to $0 \leq x_i \leq 1$. In a satisfying truth assignment, at least one literal must be true in any clause C_i, which can be expressed as a constraint

$$\sum_{x \in P_i} x + \sum_{x \in N_i} (1 - x) \geq 1$$

For example, for the formula

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

we get constraints

$$\begin{aligned}
x_1 + x_2 + x_3 &\geq 1 \\
(1 - x_1) + (1 - x_2) + (1 - x_3) &\geq 1 \\
x_1 + (1 - x_2) + (1 - x_3) &\geq 1 \\
(1 - x_1) + (1 - x_2) + x_3 &\geq 1 \\
(1 - x_1) + x_2 + x_3 &\geq 1 \\
x_1, x_2, x_3 &\geq 0 \\
x_1, x_2, x_3 &\leq 1
\end{aligned}$$

Clearly any satisfying assignment F is a feasible solution of I, and vice versa. Hence, the formula F is satisfiable if and only if there exists a feasible solution of I.

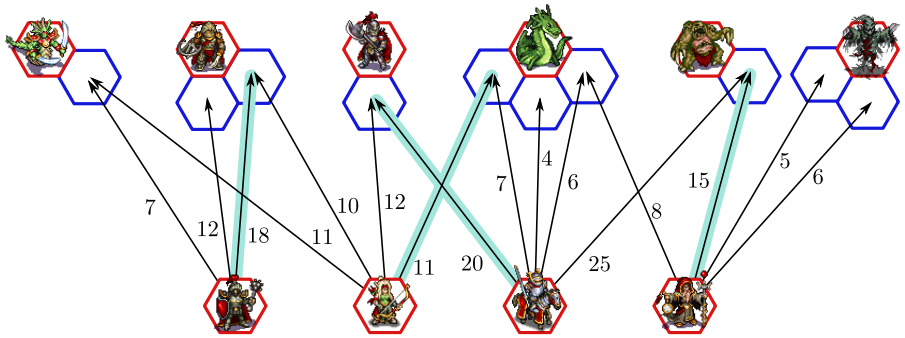
Exercise. Try to express other NP-hard problems (e.g. the Traveling Salesman Problem, Maximum Clique Problem, …) as ILP.

Our previous thoughts can be summarized as follows: ILP is a general framework that can express many optimization problems. This expressive power, however, comes at a cost, since we don't expect a polynomial-time algorithm for solving ILP to exist.

The good: MAX-WEIGHTED-BIPARTITE-MATCHING

Imagine you are writing a program to play a turn-based strategic game. In a turn, several units can be activated: each of them can move next to some enemy unit, and potentially attack it. After

an attack, the figure is not allowed to move anymore, and since any tile can be occupied by at most one unit, no two units cannot attack from the same tile. Every potential attack has a known damage, and the task is to move the units in such a way that maximizes the overall damage.[1]



Each unit in the bottom line can be moved to some of the indicated blue tiles and attack an enemy unit. The numbers represent the damage dealt by the corresponding attack. The optimal attack is highlighted.

The problem we are facing is known as the Maximum Bipartite Matching problem[2]:

Definition 2.3. Given a bipartite graph with edges labeled by non-negative weights, the MAX-WEIGHTED-BIPARTITE-MATCHING problem asks to select a set of edges with maximal overall weight in such a way that no two selected edges share a common vertex.

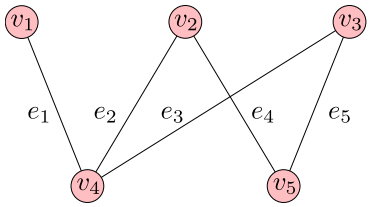
The problem MAX-WEIGHTED-BIPARTITE-MATCHING can be formulated as an ILP problem: given a bipartite graph $G = (V, E)$ with edge-weights $\omega : E \mapsto \mathbb{R}^+$, let us introduce a selector variable $x_e \in \{0, 1\}$ for each edge $e \in E$, that is selected if the corresponding edge has been included in the matching. The goal is to maximize the sum of the weights of the selected edges, i.e. $\max \sum_{e \in E} \omega_e x_e$ What remains is to design a set of linear constraints that ensure that the selected edges form a matching. The whole ILP then looks as follows (the constrains $x_e \leq 1$ don't have to be written explicitly, since they are implied by the rest of the program):

$$\begin{array}{lll} \text{maximize} & \sum_{e \in E} \omega_e x_e & \\ \text{subject to} & \sum_{\substack{e \in E \\ e=(v,w)}} x_e \leq 1 & \forall v \in V \\ & x_e \geq 0 & \forall e \in E \\ & x_e \in \mathbb{Z} & \end{array} \tag{19}$$

For example, for the graph on the left we get the constraints

[1] Admittedly, this is not the best possible overall strategy, but the task may be used as a subproblem in some specific cases.

[2] This problem is somewhat special: most of the problems presented in this course are NP-hard, but MAX-WEIGHTED-BIPARTITE-MATCHING is not. The reader may be familiar with an efficient algorithm to solve this problem. Even if it is the case, we ask the reader to follow our exposition, as it is meant to give an insight into the properties of ILPs.



$$\begin{aligned} x_{e_1} &\leq 1 \\ x_{e_2} + x_{e_4} &\leq 1 \\ x_{e_3} + x_{e_5} &\leq 1 \\ x_{e_1} + x_{e_2} + x_{e_3} &\leq 1 \\ x_{e_4} + x_{e_5} &\leq 1 \\ x_{e_1}, x_{e_2}, x_{e_3}, x_{e_4}, x_{e_5} &\geq 0 \end{aligned} \tag{20}$$

Since we have no efficient algorithm for solving ILP, we can try to solve the *relaxed* LP, where the constraints $x_e \in \mathbb{Z}$ are left out of the program (19). Clearly, all the feasible solutions of the ILP are also feasible solutions of the relaxed program, so the optimum of the relaxed program is an upper bound on the optimum of the ILP. As we have already seen, solving the relaxed program is in general a bad idea, since the optimum of the ILP can be really far away from the optimum of the relaxed program. However, we are not interested in arbitrary programs, but only in those that are obtained the way we described from some bipartite graph. This gives us hope that maybe due to the inherent structure of these programs some information about the solution of the ILP is contained in the optimal solution of the relaxed program. For example, let us use the constraints from (20), and let us set $\omega_{e_1} = 1$ and $\omega_{e_2} = \cdots = \omega_{e_5} = 10$. When we try to solve the relaxed linear program using the simplex method[3], to our pleasant surprise the found optimal solution will be integral, although there surely exist also non-integral optimal solutions (e.g. $x_{e_1} = 0$, $x_{e_2} = \cdots = x_{e_5} = \frac{1}{2}$). We can try different weights, and different graphs, and, surprisingly, the result is always the same: the LP solver for the relaxation always finds an integral optimal solution. Let's investigate this strange behavior further. We write our program in the matrix form

$$\max\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$$

where

$$\mathbf{c} = \begin{pmatrix} 1 \\ 10 \\ 10 \\ 10 \\ 10 \end{pmatrix} \qquad A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \qquad \mathbf{x} = \begin{pmatrix} x_{e_1} \\ x_{e_2} \\ x_{e_3} \\ x_{e_4} \\ x_{e_5} \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

The matrix A represents the bipartite graph in a way that there is one row for each vertex, and one column for each edge. In each column, there are ones in the rows corresponding to the (two) vertices incident with the given edge. Such a matrix is called the *incidence matrix* of a bipartite graph. In our particular case, we can further simplify the program by leaving the first row out: the constraint $x_{e_1} \leq 1$ is implied by $x_{e_1} + x_{e_2} + x_{e_3} \leq 1$, and the fact that the variables are non-negative. We introduce the slackness variables, and obtain a linear program in the normal form:

$$\max\{\mathfrak{c}^\mathsf{T}\mathfrak{x} \mid \tilde{A}\mathfrak{x} = \mathfrak{b},\ \mathfrak{x} \geq \mathbf{0}\} \tag{21}$$

where

$$\mathfrak{c} = \begin{pmatrix} 1 \\ 10 \\ 10 \\ 10 \\ 10 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad \tilde{A} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathfrak{x} = \begin{pmatrix} x_{e_1} \\ x_{e_2} \\ x_{e_3} \\ x_{e_4} \\ x_{e_5} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} \qquad \mathfrak{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

[3]For increased dramatic effect we encourage the reader to really try it.

Recall the definition of a basic solution. For each basis B, the matrix $\tilde{A}_B$ is a non-singular square matrix of dimensions 4×4; the non-basic variables are all zeroes, and the values of basic variables are given by the solution of the system

$$\tilde{A}_B \mathbf{x}_B = \mathfrak{b}.$$

Using the Cramer rule, the i-th component of the solution is given by

$$\frac{\det\left(\tilde{A}_B\langle i\rangle\right)}{\det\left(\tilde{A}_B\right)}$$

where $\tilde{A}_B\langle i\rangle$ is a matrix obtained from $\tilde{A}_B$ by replacing the i-th column by the right-hand side vector $\mathfrak{b}$. For example, the basis $B = (2, 4, 7, 8)$ yields a system

$$\tilde{A}_B \mathbf{x}_B = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_{e_2} \\ x_{e_4} \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

The following determinants are needed to solve the system using the Cramer rule:

$$\det\left(\tilde{A}_B\right) = \begin{vmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{vmatrix} = 1$$

$$\begin{vmatrix} \mathbf{1} & 1 & 0 & 0 \\ \mathbf{1} & 0 & 1 & 0 \\ \mathbf{1} & 0 & 0 & 1 \\ \mathbf{1} & 1 & 0 & 0 \end{vmatrix} = 0 \qquad \begin{vmatrix} 1 & \mathbf{1} & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 \\ 1 & \mathbf{1} & 0 & 1 \\ 0 & \mathbf{1} & 0 & 0 \end{vmatrix} = 1 \qquad \begin{vmatrix} 1 & 1 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 1 & \mathbf{1} & 0 \end{vmatrix} = 1 \qquad \begin{vmatrix} 1 & 1 & 0 & \mathbf{1} \\ 0 & 0 & 1 & \mathbf{1} \\ 1 & 0 & 0 & \mathbf{1} \\ 0 & 1 & 0 & \mathbf{1} \end{vmatrix} = 1$$

We obtained a solution $x_{e_2} = 0, x_{e_4} = 1, s_2 = 1, s_4 = 1$ that can be interpreted as a (non optimal) matching where only the edge e_4 is selected; the slackness variables $s_2 = s_3 = 1$ tell us that the vertices v_3, and v_4 have unused capacity. If we used this computation, and went through all of the $\binom{9}{4} = 126$ potential basic solutions, we would find out that all basic solutions are integral, and hence the optimum found by a simplex algorithm is also integral. Now let us try to generalize this finding.

Definition 2.4. A square matrix A with integral entries is called *inumodular*, if $det(A) = \pm 1$, where $det(\cdot)$ denotes the determinant of a matrix.
A matrix (not necessarily square) A with integral entries is called *totally unimodular* (TUM), if every non-singular square submatrix (i.e. a $k \times k$ matrix that is obtained from A by selecting some k rows and some k columns) is unimodular.

This definition in particular implies that the entries of a TUM matrix A are $0, \pm 1$: every entry is a 1×1 submatrix. Moreover for any square submatrix A', $det(A') \in \{0, \pm 1\}$.

Theorem 2.5. *Consider a linear program in normal form* $\max\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$, *where* A *is a TUM matrix, and* $\mathbf{b}$ *is a vector with integral entries. Then all basic solutions are integral.*

Proof: Let us consider a basis B, then A_B is a square non-singular submatrixof A. The values of the basic variables are given by the solution of the system $A_B\mathbf{x}_B = \mathbf{b}$, and can be expressed as

$$\frac{\det\left(A_B\langle i\rangle\right)}{\det(A_B)}.$$

Since A is TUM, $\det(A_B) = \pm 1$. Using the expansion along the i-th column to compute the determinant $\det(A_B\langle i\rangle)$, we get

$$\det(A_B\langle i\rangle) = (-1)^{i+1}b_1C_1 + \cdots + (-1)^{i+m}b_mC_m,$$

where C_j's are the determinants of the submatrices of A, and hence $C_j \in \{0, \pm 1\}$. By the hypothesis, $b_i \in \mathbb{Z}$, and the proof is concluded. □

In fact, we used a slightly weaker condition than the total unimodularity. It would be sufficient that the determinant of any non-singular $m \times m$ submatrix is ± 1, and the remaining determinants of square submatrices are integral. However, the class of the problems with TUM matrices is rich enough for our purposes, and in what follows we present some handy sufficient conditions.

Theorem 2.6. *Let A be an $n \times m$ TUM matrix, and let $k \leq m$. Then $B := \left(\begin{array}{c|c} A & \begin{array}{c} 0 \\ \hline I_k \end{array} \end{array}\right)$, where I_k is the $k \times k$ identity matrix, is TUM.*

Proof: Let C be an arbitrary non-singular square submatrix of B, then $C = (C_1 \mid C_2)$, where C_1 are columns from matrix A, and C_2 are columns from $\left(\begin{array}{c} 0 \\ \hline I_k \end{array}\right)$. Let us regroup the rows of C in such a way that the rows that are all zeroes in C_2 come first, yielding a non-singular matrix

$$\left(\begin{array}{c|c} A' & 0 \\ \hline X & I_\ell \end{array}\right)$$

where A' is a square submatrix of A. Since the matrix is non-singular, we have $\det(C) = \pm 1$. □

Theorem 2.7. *Consider a linear program* $\max\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$, *where A is TUM matrix, and $\mathbf{b}$ is a vector with integral coefficients. Then all basic solutions are integral.*

Proof: Following Theorem 2.6 a matrix $(A \mid I)$ is TUM, where I is the $m \times m$ identity matrix. The statement of the theorem then follows from the normalization of linear programs, where the matrix I represents the slackness variables. □

Theorem 2.8. *Le A be a matrix with entries $a_{ij} \in \{0, \pm 1\}$, such that the rows of A can be decomposed into two sets I_1, I_2 in such a way that*

1. *If there are two entries with the same sign in the same column, their rows belong to different sets I_1, I_2.*
2. *If there are two entries with different signs in the same column, their rows belong to the same set.*

Then A is TUM.

Proof: We have to show that every non-singular $k \times k$ submatrix of A has determinant ± 1. We prove this statement by induction on k. The induction basis is formed by the 1×1 submatrices, i.e. the entries of A. Let us argue by induction that the statement holds for $k-1 \times k-1$ submatrices. We are going to argue that then it holds for $k \times k$ submatrices. Consider an arbitrary $k \times k$ submatrix C. If all columns of C are zero, C is singular. If there is a column with exactly one non-zero entry (i.e. with value ± 1), we use the expansion of the determinant according to this column, and from the induction hypothesis it follows $det(C) = \pm 1$. Finally, suppose that every column contains at least two non-zero entries. From the statement of the theorem it follows that for every column j it holds

$$\sum_{i \in I_1} c_{ij} = \sum_{i \in I_2} c_{ij}$$

Hence, the sum (which is a linear combination) of the columns is 0, yielding $\det(C) = 0$. □

Corollary 2.9. *Every linear program of the from*

$$\max\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$$

or

$$\max\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\},$$

where $\mathbf{b}$ *has integral coefficients, and* A *is*

1. *incidence matrix of an undirected bipartite graph, or*
2. *incidence matrix of a directed graph*[4]

has only integral basic solutions.

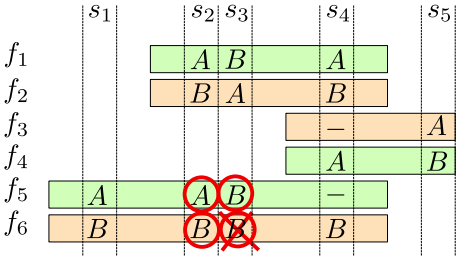
Proof: In the case 1) denote I_1 the rows corresponding to vertices in one bipartition, in the case 2) let I_1 be all rows. Subsequently, use Theorem 2.8. □

Now we can see that Max-Weighted-Bipartite-Matching is just one from a small class of problems where the simplex algorithm solving the relaxed program always returns integral optimum. Naturally, we won't be that lucky with the NP-hard problems we are investigating in this course. However, sometimes the optimal solution of the relaxed program can convey enough information that enables us to find at least an approximate solution of the integer program. We shall present several examples of this approach in the next parts.

Exercise. Express the problem of finding shortest path between two vertices in a graph as a linear program.

The bad: Min-Vertex-Cover

Problems where the relaxed linear program delivers the optimal solution are rare exceptions. However, even in some other cases, the solution of the relaxed LP can somewhat help. As an example, we introduce the following problem from computational biology (see [**?**]). Specimens of a given species share vast majority of the genome, with only few places where the genetic information can differ from specimen to specimen. The task at hand is to identify so called *snips* (*Single Nucleotide Polymorphism*) in the genome, which are places where different values can appear, surrounded by a context that is the same for all specimens. The genome is sequenced into a set of fragments, and the input is a matrix that specifies, for each snip and each fragment a value (A, or B, or $-$ if it is not detected). The goal is to split the segments into two parts (colors) corresponding to two alternative genomes.



A configuration of fragments f_4,f_5 and snips s_2,s_3 form a conflict: the snip s_2 asserts that fragments f_4, f_5 are in different parts, but the snip s_3 asserts that both f_4, and f_5 have the same value. If the value of the snip s_3 on the fragment f_6 were A, the green and red parts would comprise two alternative genomes.

[4] a matrix with a row for every vertex, and a column for each directed edge, such that in a column corresponding to a given directed edge (v_i, v_j), the i-th entry is 1, and the j-th entry is -1 (and the remaining entries are all zeroes).

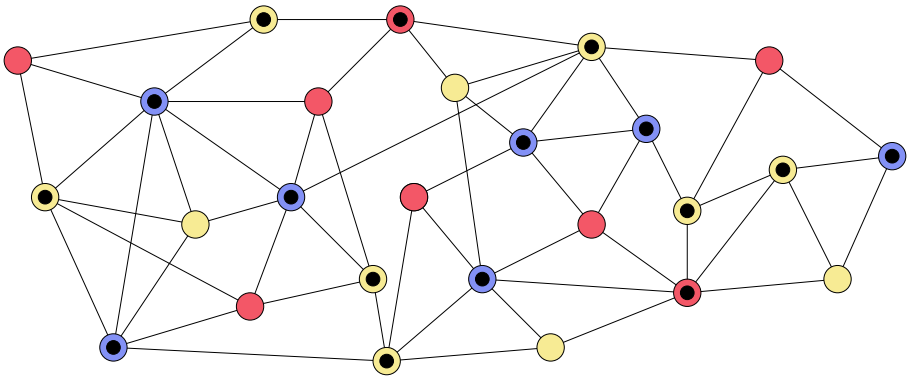
The input data always contain some number of errors. A configuration of two segments and two snips as on the above picture creates a conflict: one of the snips has to be read wrongly and must be ignored. If we have, for each snip, a numerical plausibility value, we can create a conflict graph of the snips: the vertices represent the snips, and an edge connects two snips if they are in conflict. The first step towards identifying the alternative genomes is to drop some snips to get a conflict-free assembly. This leads naturally to the following problem:

Definition 2.10. Given is a simple undirected graph $G = (V, E)$ with vertices labeled by non-negative weights $\omega : V \mapsto \mathbb{R}^+$. The goal of the problem Min-Vertex-Cover is to select a set of vertices such that every edge is incident with at least one selected vertex, and the overall weight of the selected vertices is minimized.

The problem is easily formulated as an ILP. For every vertex $v \in V$ we introduce a binary variable $x_v \in \{0, 1\}$ which indicates whether the vertex v is selected or not. The complete ILP then is as follows:

$$\begin{array}{lrcll} \text{minimize} & \sum\limits_{v \in V} \omega_v x_v & & & \\ \text{subject to} & x_u + x_v & \geq & 1 & \forall e = (u, v) \in E \\ & x_v & \geq & 0 & \forall v \in V \\ & x_v & \in & \mathbb{Z} & \end{array} \tag{22}$$

The utility function tells that we are minimizing the sum of weights of selected edges. For each edge $(u, v) \in E$ the constraint of the form $x_u + x_v \geq 1$ ensures that at least one of the endpoints is selected. Constraints $x_v \geq 0$ and $x_v \in \mathbb{Z}$ together ensure that $x_v \in \{0, 1\}$: if there is a feasible solution with some value $x_v > 1$, setting $x_v = 1$ keeps all the constraints satisfied and, since the weights are non-negative, the value of the utility function does not increase.



An example of a vertex cover. The blue vertices have weight 1, the yellow ones have weight 2, and the red ones have weight 13. The whole cover has the weight 47. Is it possible to find a cover with a smaller weight?

The problem Min-Vertex-Cover is NP-hard, so we don't expect to have a polynomial-time algorithm that would solve the program (22) in polynomial time. Therefore we settle with an approximation: our goal is to find a solution with a weight not exceeding twice the of the optimal solution. Such an algorithm is referred to as 2-approximative (in the following definition r is not necessarily constant, so we need to take the supremum over all inputs of given size):

Definition 2.11. Consider an optimization problem $\mathcal{P}$, and a polynomial-time algorithm $\mathcal{A}$ that solves it. Let $m^\star(\mathbf{x})$ be the value of the optimal solution of the input $\mathbf{x}$, and let $m_\mathcal{A}(\mathbf{x})$ be the value computed by the algorithm $\mathcal{A}$. An *approximation ratio* of the algorithm $\mathcal{A}$ on the input $\mathbf{x}$ is

$$R_\mathcal{A}(\mathbf{x}) = \begin{cases} \frac{m^\star(\mathbf{x})}{m_\mathcal{A}(\mathbf{x})} & \text{if the goal of } \mathcal{P} \text{ is minimization} \\ \\ \frac{m_\mathcal{A}(\mathbf{x})}{m^\star(\mathbf{x})} & \text{if the goal of } \mathcal{P} \text{ is maximization} \end{cases}$$

An algorithm $\mathcal{A}$ is $r(n)$-*approximative*, if

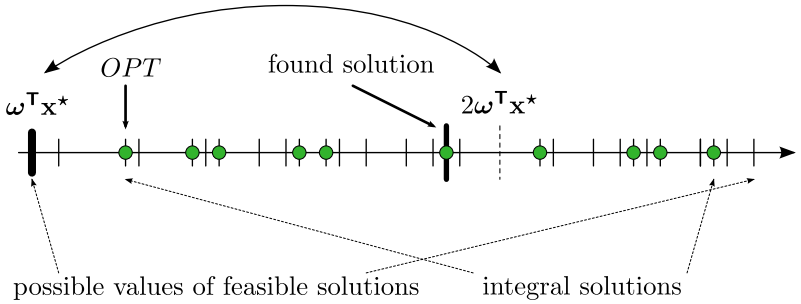
$$\forall n : \sup_{|\mathbf{x}|=n} R_\mathcal{A}(\mathbf{x}) \leq r(n)$$

where the supremum is taken over all inputs of length n.

To begin with, let us relax the ILP (22) by leaving out the constraints $x_v \in \mathbb{Z}$ obtaining an LP

$$\begin{array}{lrcll} \text{minimize} & \sum_{v \in V} \omega_v x_v & & & \\ \text{subject to} & x_u + x_v & \geq & 1 & \forall e = (u,v) \in E \\ & x_v & \geq & 0 & \forall v \in V \end{array} \tag{23}$$

that can be efficiently solved. Let OPT be the optimal (integral) solution of the program (22) and let $\mathbf{x}^\star$ be the optimal solution of the program (23). Clearly $\boldsymbol{\omega}^\mathsf{T}\mathbf{x}^\star \leq OPT$ (the relaxation could only increase the set of feasible solutions, so the value of the optimum could only decrease). The question we are worried about is how far is $\boldsymbol{\omega}^\mathsf{T}\mathbf{x}^\star$ from OPT, and how to obtain from $\mathbf{x}$ a feasible solution that would not be far away from OPT. In this particular case we show that we are lucky and $OPT \leq 2\boldsymbol{\omega}^\mathsf{T}\mathbf{x}$. The situation is depicted on the following figure:



Among the possible values of the utility function, $\boldsymbol{\omega}^\mathsf{T}\mathbf{x}$ is the smallest. We are interested in OPT, which is the smallest integral solution. If we manage to construct a feasible integral solution with the weight at most $2\boldsymbol{\omega}^\mathsf{T}\mathbf{x}$, we can be sure that its weight is at most $2OPT$.

The most straightforward way to obtain an integral solution from $\mathbf{x}^\star$ is to arithmetically round the values $x_v^\star$. We know that $x_v^\star \geq 0$, and also without loss of generality $x_v^\star \leq 1$ (there is no point in setting any variable $x_v^\star > 1$, since then decreasing it to 1 does not increase the value $\omega_v x_v^\star$, and all constraints remain satisfied), so rounding means to select the vertices for which $x_v^\star \geq \frac{1}{2}$. First we have to verify that the rounding results in a feasible solution. If in the relaxed solution $x_u^\star + x_v^\star \geq 1$ for each edge $(u,v) \in E$, clearly at least one of the values $x_u^\star, x_v^\star \geq \frac{1}{2}$, and the corresponding vertex is selected by the rounding procedure. The fact that the value of the rounded solution $\widehat{\mathbf{x}}$ is at most twice the value of the optimal solution follows from the simple computation

$$\boldsymbol{\omega}^\mathsf{T}\widehat{\mathbf{x}} = \sum_{v \in V} \omega_v \hat{x}_v = \sum_{\substack{v \in V \\ x_v^\star \geq \frac{1}{2}}} \omega_v \hat{x}_v \leq 2 \sum_{\substack{v \in V \\ x_v^\star \geq \frac{1}{2}}} \omega_v x_v^\star \leq 2 \sum_{v \in V} \omega_v x_v^\star = 2\boldsymbol{\omega}^\mathsf{T}\mathbf{x}^\star.$$

The first equality is just a scalar product rewritten. In the second equality we used that vertices v for which $x_v^\star < 1/2$ have $\hat{x}_v = 0$. The next inequality follows from the fact that all the remaining vertices have $x_v^\star = ge\frac{1}{2}$ and $\hat{x}_v = 1$.
That a simple rounding procedure gives reasonable results is actually a rare case, and here it is a consequence of the special structure of the relaxed program (23). In a moment we shall show that every feasible basic solution of (23) is *half-integral*, i.e. $x_v \in \left\{0, \frac{1}{2}, 1\right\}$. This means there is an optimal half-integral solution, and if all values $\frac{1}{2}$ are replaced by ones, the overall weight is at most doubled. To show the half-integrality we use the following general lemma stating that feasible basic solutions of a linear program are the corners of $\mathcal{D}$: a point $\mathbf{x}$ that is not a corner lies on a line segment completely inside $\mathcal{D}$, and can be expressed as $\mathbf{x} = t\mathbf{y} + (1-t)\mathbf{z}$ for two points $\mathbf{y}, \mathbf{z} \in \mathcal{D}$, and $0 < 1 < t$.

Lemma 2.12. *Consider a linear program in the normal form. No feasible solution* $\mathbf{x}$ *that is a convex combination of two feasible solutions* $\mathbf{y}$,$\mathbf{z}$ *(i.e.* $\mathbf{x} = t\mathbf{y} + (1-t)\mathbf{z}$ *for some* $t:\ 0 < t < 1$*) is basic.*

Proof: Consider a feasible basic solution $\mathbf{x}$, and let us suppose, for the sake of contradiction, that $\mathbf{x}$ is a convex combination of feasible solutions $\mathbf{y}$,$\mathbf{z}$. Let B be the basic set of the solution $\mathbf{x}$, i.e. A_B is a non-singular square matrix and $x_j = 0$ for all $j \notin B$. Take a vector $\boldsymbol{\epsilon}$ such that $\tilde{c}_j = 0$ for $j \in B$, and $\tilde{c}_j = -1$ else. Clearly $\boldsymbol{\epsilon}^\mathsf{T}\mathbf{x} = \mathbf{0}$, and at the same time for any $\mathbf{v} \geq \mathbf{0}$ it holds $\boldsymbol{\epsilon}^\mathsf{T}\mathbf{v} \leq \mathbf{0}$. Moreover, if $\mathbf{v}$ has at least one coordinate $v_j > 0$ for $j \notin B$, then $\boldsymbol{\epsilon}^\mathsf{T}\mathbf{v} < \mathbf{0}$. We know that $\mathbf{x} = t\mathbf{y} + (1-t)\mathbf{z}$, and so $\mathbf{0} = \boldsymbol{\epsilon}^\mathsf{T}\mathbf{x} = t\boldsymbol{\epsilon}^\mathsf{T}\mathbf{y} + (1-t)\boldsymbol{\epsilon}^\mathsf{T}\mathbf{z}$ for $\mathbf{y}, \mathbf{z} \geq \mathbf{0}$. It is possible only if $\boldsymbol{\epsilon}^\mathsf{T}\mathbf{x} = \boldsymbol{\epsilon}^\mathsf{T}\mathbf{y} = \boldsymbol{\epsilon}^\mathsf{T}\mathbf{z} = \mathbf{0}$, and so both $\mathbf{y}$ and $\mathbf{z}$ have all $y_j = z_j = 0$, $j \notin B$. However, $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ are all feasible solutions, hence $A\mathbf{x} = A\mathbf{y} = A\mathbf{z} = \mathbf{b}$. Since $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ have zero non-basic coordinates, we obtained $A_B\mathbf{x} = A_B\mathbf{y} = A_B\mathbf{z} = \mathbf{b}$. But A_B is non-singular square matrix, and as such the system $A_B\mathbf{x} = \mathbf{b}$ has unique solution, yielding $\mathbf{x} = \mathbf{y} = \mathbf{z}$. □

In view of the previous lemma it is sufficient to show that any feasible solution $\mathbf{x}$ of the program (23) that has some value $x_v \notin \left\{0, \frac{1}{2}, 1\right\}$ can be expressed as a convex combination of two feasible solutions $\mathbf{y}$ and $\mathbf{z}$.
Take any feasible solution $\mathbf{x}$ in which there is some vertex with value $x_v \notin \left\{0, \frac{1}{2}, 1\right\}$, and divide the vertices with values of $\mathbf{x}$ different from $\left\{0, \frac{1}{2}, 1\right\}$ into two groups:

$$V_+ = \left\{ v \mid 0 < x_v < \frac{1}{2} \right\} \qquad\qquad V_- = \left\{ v \mid \frac{1}{2} < x_v < 1 \right\}$$

For any fixed constant ε define two solutions $\mathbf{y}$ and $\mathbf{z}$ as follows:

$$y_v = \begin{cases} x_v + \varepsilon, & \text{if } x_v \in V_+ \\ x_v - \varepsilon, & \text{if } x_v \in V_- \\ x_v, & \text{else} \end{cases} \qquad\qquad z_v = \begin{cases} x_v - \varepsilon, & \text{if } x_v \in V_+ \\ x_v + \varepsilon, & \text{if } x_v \in V_- \\ x_v, & \text{else} \end{cases}$$

Clearly $\mathbf{y} \neq \mathbf{z}$, and $\mathbf{x} = \frac{1}{2}(\mathbf{y}+\mathbf{z})$. Also, it is quite straightforward that when choosing $\varepsilon < \min\{x_v \mid v \in V\}$, the $\mathbf{y}, \mathbf{z} \geq \mathbf{0}$. It remains to be shown that $\mathbf{y}$,$\mathbf{z}$ are feasible, i.e. the constraints on the edges are satisfied. Consider an edge (u, v), where $x_u + x_v > 1$. If $\varepsilon < \frac{1}{2}(x_u + x_v - 1)$, this constraint holds both in $\mathbf{y}$ and $\mathbf{z}$. If $x_u + x_v = 1$, distinguish two cases: either $x_u, x_v \in \left\{0, \frac{1}{2}, 1\right\}$ or $u \in V_+$, $v \in V_-$ (or the other way round). In either case, regardless of the value of ε, the constraint is satisfied in both $\mathbf{y}$ and $\mathbf{z}$.
At this point we owe the reader some explanation. We have found a 2-approximating algorithm, and we seem to be happy. Why? In reality a solution that can be as bad as twice the value of the optimum is usually useless. There are several answers to the question why are algorithms with provable approximation ratio interesting.

Theoretical[5] **interest.** In spite of many strong complexity-theoretical results, after decades of intense effort we still don't really understand why some problems are "easy" and other are "hard".

[5] the author does not consider the word "*theoretical*" insulting

There are problems for which we can find in polynomial time an arbitrary good approximation, yet we don't know of any polynomial-time algorithm to find the exact optimum. On the other hand, there are problems for which we can show that even an extremely weak approximation guarantee (say, with the approximation ratio $n^{0.99}$) would imply $P = NP$. If we show a, say, 2-approximation algorithm for a given problem, we localize it in a hierarchy of known problems with similar properties. Hopefully, these similarities could lead to a better understanding of the notion of "hardness".

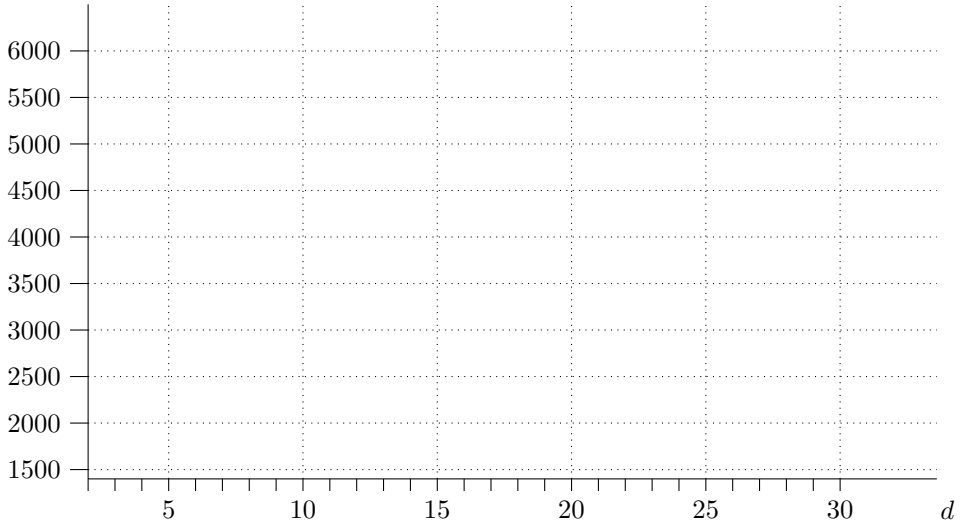
Sanity check. The usual approach to solving (NP-)hard problems is to use various heuristic methods, either ad-hoc heuristics tailored to the specific problem, or generic meta-heuristics like neural networks, simulated annealing, taboo-search and others. Theses methods are often able to find remarkably good solutions. However, they don't come with any guarantees of their performance, and it may indeed happen that in some cases the found solution is way way off. An approximation algorithm that is some orders of magnitude faster, with a guaranteed worst-case performance can be employed as a sanity-check that the solution found by the heuristic is reasonable.

Heuristic use. Approximation algorithms have proven worst-case guarantee, i.e. the proven approximation ratio holds for all inputs. Moreover, in order to prove such guarantee we have to bound the unknown value of the optimum. This is usually the hardest part of the proof, and oftentimes we have to employ bounds that are very rough. One could hope, and often it is also the case, that the algorithms deliver on "real life" inputs much better results than the proven guarantee. However, to formally analyze what inputs are "real life" is a risky business. The average-case analysis relies on a-priori information about the probability distribution over inputs: in reality, however, the distribution of inputs is usually quite complex. In cases where the algorithm is meant for a specific setting (e.g. the use of Min-Vertex-Cover-solving algorithm for the snip-finding problem) one can measure the performance of the algorithm on known data sets (e.g. known conflicts graphs of snips), and hope that in the production the inputs will have similar properties (whatever the key properties may be). However, when designing a general-purpose algorithm, e.g. as a part of some library, there is no reasonable notion of "real world" input: the designers can never predict all possible uses of their library. So the worst-case is the only reasonable measure to work with. In this respect, the algorithms based on LP duality have the benefit of providing, for each particular input, a corresponding bound on the optimum from the opposite side as part of their solution. Hence, for each input, we have not only an approximate solution, but also an interval where the true optimum is located.

To illustrate this concept let us consider the 2-approximation algorithm for the Min-Vertex-Cover problem. As a first test, let us take the list of all connected cubic[6] graphs on 20 vertices (there are 510489 of them). In the unweighted case (all edges have unit weights) are all relaxed optimal solutions of the from $x_{v_1} = \cdots = x_{v_{20}} = \frac{1}{2}$; the value of the relaxed optimum is 10, and after rounding we get the trivial solution with value 20. The average value of integral optimal solution is ≈ 11.5. When we assigned to each vertex a weight that was a random number from the interval $\{1, \ldots, 100\}$, the optimal relaxed solution had more integral entries. The average optimum value was 485.739042, and the average optimum value of the relaxed solution was 475.483179, and after rounding 674.606319, yielding an approximation ratio of 1.352433. When the weights were of the form $1 + 2^i$, where i is a random number from the interval $\{1, \ldots, 10\}$, the approximation ratio was only 1.041625. For graphs with two additional vertices connected to all vertices of the original graphs, the approximation ratio was 1.086788.

Now let us turn our attention to random graphs. We tested 60-vertices graphs; to get an expected degree d, any pair of vertices (u, v) was connected by an edge independently at random with probability $d/59$. From graphs generated by this process we selected 1000 connected ones for every value of d. The following figure shows the relationship of the average relaxed optimum (lower blue line), rounded solution (upper blue line), and the integral optimum (red line) for different values d with weights $1 + 2^i$, where i is a random number from the interval $\{1, \ldots, 10\}$.

[6]such that each vertex has exactly three neighbors



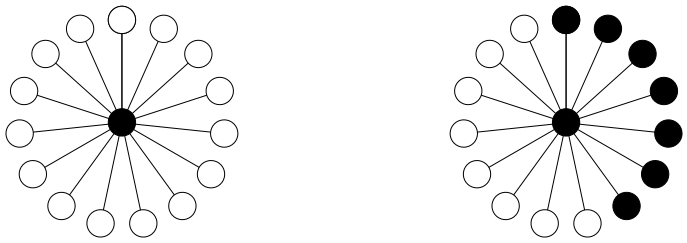
We can see that for sparse graphs (with the expected degree roughly up to seven), the relaxed optimum is expected to be quite close to the integer optimum. Moreover, many of the entries in the relaxed optimum are actually integral, so the rounding does not increase the value too much (for $d \approx 3$ the approximation ratio is $\approx 1.0.1$, i.e. the relative error is roughly 1%, for $d \approx 5$ the error is 8%). As the density increases, the fractional entries are more populated in the relaxed optimum, which increases both the distance between the integral and relaxed optimum, and the distance between relaxed optimum, and the rounded solution. Around the expected degree 15, the relaxed optimum starts to be comprised mostly of $\frac{1}{2}$, so that the rounding costs roughly a factor of two. However, at the same time the cost of the (integral) optimum increases, so the approximation tratio slowly decreases. Thus we see that even a very simple algorithm with a proven guarantee of 2 can perform much better on some clases of inputs.

The ugly: Max-Independent-Set

In the previous part we set a goal in the snip-finding scenario to minimize the overall weight of removed snips in such a way that from every edge of the conflict graph at least one snip is removed. The same task can be formulated in a dual way: we want to select as many snips (w.r.t. overall weight) as possible provided that no two selected snips are connected by an edge in the conflict graph. We naturally get a formulation of the Max-Independent-Set problem:

Definition 2.13. Given is a simple graph $G = (V, E)$ with non-negative weights on vertices, i.e. a function $\omega : V \mapsto \mathbb{R}^+$. The goal of the problem Max-Independent-Set is to select a set of vertices in such a way that no two selected vertices are connected by an edge, and the overall weight of the selected vertices is maximized.

For each independent set S the remaining vertices $V - S$ form a vertex cover: since S is independent, every edge is incident with at most one vertex from S, hence every edge is incident with at least vertex from $V - S$. On the other hand, for any vertex cover C, the remaining vertices $V - C$ form an independent set: every edge is incident with at least one vertex from C, hence no two vertices from $V - C$ are connected by an edge. Problems Min-Vertex-Cover and Max-Independent-Set are thus equivalent from the point of view of the optimal solution. From the point of view of approximation, however, they are quite different.



In an unweighted star with n vertices, the maximum independent set $S^\star$ is of size $n-1$, and the minimum vertex cover $V - S^*$ is of size 1. The white vertices on the right form an independent set S of size $n/2$ which is a 2-approximation of the optimum. However, the $n/2$ vertices from the set $V - S$ is only a $n/2$-approximation of the minimum vertex cover.

In the same way as the MIN-VERTEX-COVER, the MAX-INDEPENDENT-SET can be formulated as an ILP:

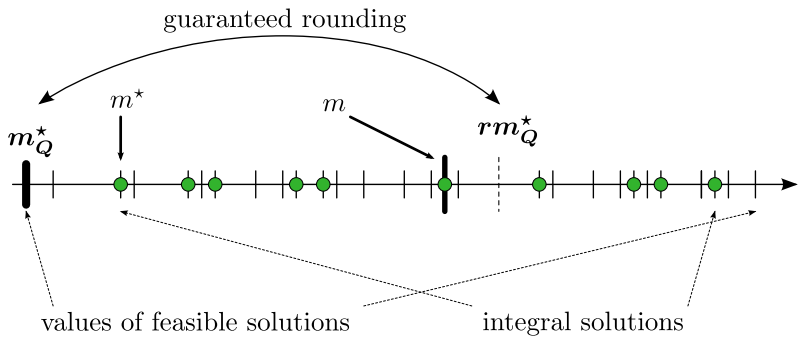
$$\begin{array}{lrcll} \text{maximize} & \sum_{v \in V} \omega_v x_v & & & \\ \text{subject to} & x_u + x_v & \leq & 1 & \forall e = (u,v) \in E \\ & x_v & \geq & 0 & \forall v \in V \\ & x_v & \in & \mathbb{Z} & \end{array} \tag{24}$$

If we consider the special case of unweighted graphs, i.e. $\omega_{v_1} = \cdots = \omega_{v_n} = 1$, similarly as in the case of MIN-VERTEX-COVER the optimal relaxed solution for dense graphs will be of the form $x_{v_1} = \cdots = x_{v_n} = \frac{1}{2}$. Here, however, the similarity with MIN-VERTEX-COVER ends: there are namely many graphs with small maximum independent set, but with the relaxed optimum $n/2$. Hence we cannot find a "rounding" algorithm that would find an integral solution "close" (say, a constant factor) to the relaxed optimum. This is not a coincidence. As was proved in [7], any polynomial-time algorithm that would find a $n^{1-\varepsilon}$-approximation of MAX-INDEPENDENT-SET for some $\varepsilon > 0$ would imply $P = NP$.

In this chapter we showed that various optimization problems can be formulated as ILP. In some cases the optimum of the corresponding relaxed program closely approximated the optimum of the ILP, and in other cases they are urelated. IN the nex chapters we shall show how to use the relaxed linear programs to find a reasonably good approximation.

2.2 Deterministic rounding

In the previous chapter we applied the following general scheme for design of r-approximation algorithms, to the MIN-VERTEX-COVER problem:

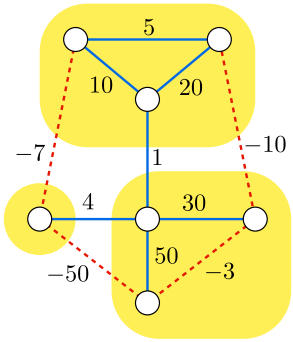


When looking for the minimum of the ILP with value $m^\star$, we first find the (possibly smaller) minimum of the relaxed program with the value $m^\star_Q$, "round" it, and obtain a solution with value m. If we can prove that the "rounding" procedure increases the value of the solution at most

r-times, we have a guaranteed r-approximation algorithm. What we called "rounding" does not have to be the arithmetic rounding of the entries as was the case in the MIN-VERTEX-COVER algorithm in the previous section; any deterministic procedure is fine. We show an example of a more involved rounding procedure.

MIN-MULTI-CUT

As a motivation example let us consider the following problem [7]: we are given a collection of texts with references to various persons. Our goal is to distinguish when two places refer to the same person. This is not as easy as it may appear for the first sight, since the same person can be referred to by her full name, nickname, or indirectly by her position (e.g. "Her Majesty") and so on. Let us suppose that we have at our disposal (possibly as an output of some machine learning algorithms on a training data set) a predictor in the form of a real-valued function over the pairs of references: for given two references, the higher positive is the outcome, the more probably both references describe the same person; the lower negative the value is, the higher is the probability that the two references are of two different persons. A value of 0 means that even the predictor has no clue. We can then construct a graph where the vertices are the references, and the edges are labeled by positive or negative weights. Our goal is to separate the vertices into clusters such that each cluster represents one person. Since the predictor is not fully reliable, it is not always possible to find a clustering consistent with it. Hence, we want to find a clustering that minimizes the inconsistence: the sum of (absolute values of) the weights of positive edges connecting two clusters, and negative edges connecting two vertices in the same cluster.

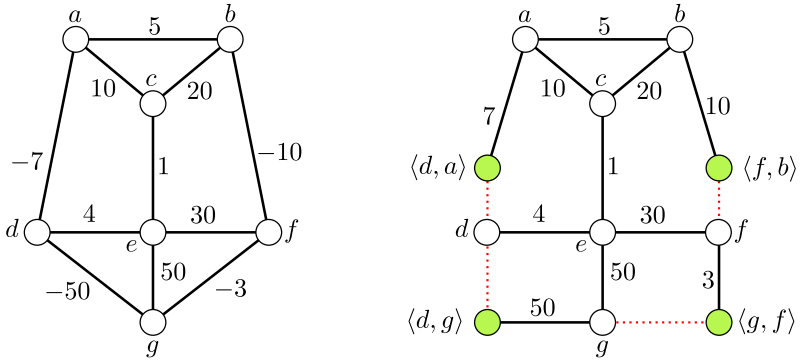


A graph and its clustering with weight 8: the positive edges with weights 4 and 1 connect two distinct clusters, and a negative edge with weight -3 is connects two vertices in a single cluster.

The task to find a clustering with minimum weight is NP-hard, so we settle with an approximate solution. Let us transform the graph as follows: we keep the positive edges. For each negative edge (u, v) with weight $-w$, we add a new vertex $\langle u, v\rangle$ connected with v by an edge of weight w. We get a graph with positive edge weights in which we want to solve the following problem: remove edges with minimum overall weight such that in the remaining graph no two vertices $\langle u, v\rangle$ and u are in the same connected component.

Exercise. Prove that this transformation is equivalent, i.e. prove that for a solution of the original problem with weight c there is a solution of the transformed problem with the same weight and vice versa.

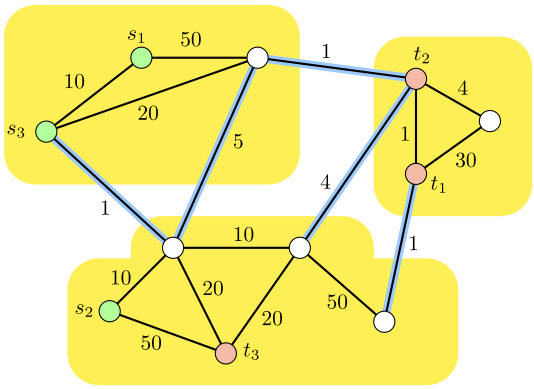
[7] see [2, 5]



Original and transformed graphs (since the original graph is undirected, the transformation is not unique). The added vertices are green, dotted lines connect vertices that must end in different components.

The transformed problem is a special case of the Min-Multi-Cut problem:

Definition 2.14. Given is a simple graph $G = (V, E)$ with non negative edge weights, i.e. a function $\omega : E \mapsto \mathbb{R}^+$, and k pairs of vertices from V: (s_i, t_i), $i = 1, \ldots, k$. The goal of the problem Min-Multi-Cut is to remove from G a set of edges with minimum overall weight such that in the resulting graph no pair of vertices (s_i, t_i) is contained in the same connectivity component.



A solution of an instance on Min-Multi-Cut problem with cost 12.

A special case of this problem for $k = 1$ is the Min-Cut problem, which can be solved in polynomial time. However for bigger values of k, the Min-Multi-Cut problem is NP-hard. Let us now try to formulate Min-Multi-Cut as an ILP. Introduce an indicator variable $x_e \in \{0, 1\}$ for each edge e, such that $x_e = 1$ means the edge was removed. The expression $\sum_{e \in E} x_e \omega(e)$ then gives the overall weight of removed edges. What remains is to formulate linear constraints that would ensure that each (s_i, t_i) are in different components. Denote $\mathcal{P}_{s_i,t_i}$ the set of all $s_i - t_i$-paths i.e. all paths in G that start in s_i and end in t_i; we need to remove an edge from each of them. Hence, we obtain the following program (we don't have to write the constraints $x_e \leq 1$ explicitly since they are implied by the minimization: for a feasible solution with some $x_e > 1$, setting $x_e := 1$ keeps all the constraints satisfied, and the value of the utility function is not increased):

$$\begin{array}{lrcll} \text{minimize} & \sum\limits_{e \in E} x_e \omega_e & & & \\ \text{subject to} & \sum\limits_{e : e \in \pi} x_e & \geq & 1 & \forall i \in \{1, \ldots, k\}, \ \forall \pi \in \mathcal{P}_{s_i,t_i} \\ & x_e & \geq & 0 & \forall e \in E \\ & x_e & \in & \mathbb{Z} & \end{array} \tag{25}$$

We now relax the program (25) by removing the constraints $x_e \in \mathbb{Z}$, yielding a linear program

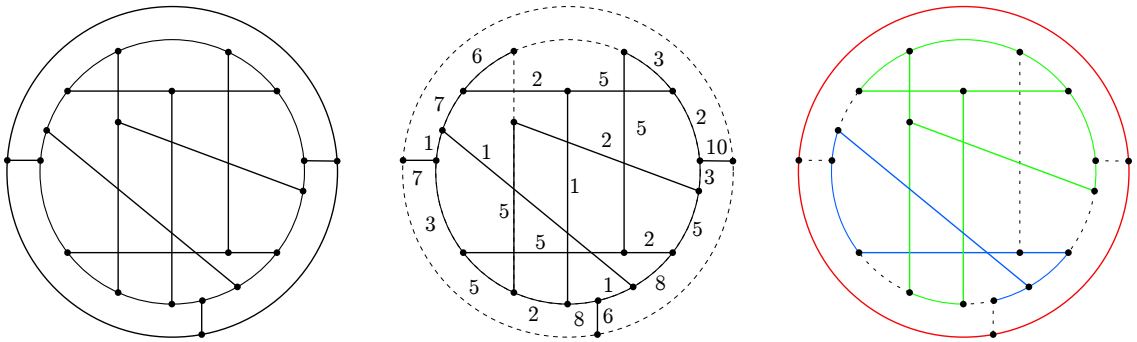
$$\begin{array}{rlcll} \text{minimize} & \sum\limits_{e\in E} x_e\omega_e & & & \\ \text{subject to} & \sum\limits_{e:e\in\pi} x_e & \geq & 1 & \forall i \in \{1,\ldots,k\},\ \forall \pi \in \mathcal{P}_{s_i,t_i} \\ & x_e & \geq & 0 & \forall e \in E \end{array} \tag{26}$$

that can be interpreted as follows: imagine the edges as a system of pipes. Assign a length x_e to each edge e, in such a way that each pair (s_i, t_i) has distance (measured as usual as the length of the shortest path) at least 1. Moreover, if we let ω_e to be the area of a cross-section of edge e, the utility function $\sum_{e\in E} x_e\omega_e$ gives the overall volume of the edges. It means we want to assign lengths as to minimize the overall volume while keeping the prescribed pairs of vertices far apart; naturally, we want to make thick edges short, and thin edges long. The relaxed program has still one problem: it has exponentially many constraints, so the simplex algorithm cannot be used to solve it efficiently. In fact, this is not a major obstacle since there are methods for efficient solving of linear programs provided there is a polynomial-time feasibility oracle: a procedure that for a given solution either tells it is feasible, or returns a constraint that is violated. To make this text self-contained, and to show an idea that may be of independent interest, we transform the program into an equivalent one with only polynomially many constraints. Consider independently each i. Instead of an explicit constraint $\sum_{e\in\pi} x_e \geq 1$ for each path $\pi \in \mathcal{P}_{s_i,t_i}$, we assign (introducing new variables) a potential $p_v^{(i)} \in \mathbb{R}$ to each vertex v, such that $p_{t_i}^{(i)} - p_{s_i}^{(i)} \geq 1$. Hence, for every path $\pi : s_i = v_1, v_2, \ldots, v_z = t_i$ we get a sequence of vertex-potentials $p_{v_1}^{(i)}, p_{v_2}^{(i)}, \ldots, p_{v_z}^{(i)}$. If we enforce that the length of an edge is at least the difference of the potentials, i.e. $x_{(v_j,v_{j+1})} \geq p_{v_{j+1}}^{(i)} - p_{v_j}^{(i)}$, we can conclude that every path has length at least one. Instead of the program (26) we thus can solve the program:

$$\begin{array}{rlcll} \text{minimize} & \sum\limits_{e\in E} x_e\omega_e & & & \\ \text{subject to} & x_e & \geq & p_v^{(i)} - p_u^{(i)} & \forall i \in \{1,\ldots,k\},\ \forall e \in E,\ e = (u,v) \\ & x_e & \geq & p_u^{(i)} - p_v^{(i)} & \\ & p_{t_i}^{(i)} - p_{s_i}^{(i)} & \geq & 1 & \forall i \in \{1,\ldots,k\} \end{array} \tag{27}$$

On one hand, every feasible solution of the program (27) satisfies the constraints of the program (26), on the other hand, for a given solution of the program (26), we can just set the potentials $p_v^{(i)}$ to be the distance (w.r.t. the values x_e) of v from s_i. It is easy to see that these potentials fulfill the requirements of the program (27), so the two programs are equivalent.

The program (27) can be solved e.g. by a simplex algorithm, obtaining an optimal solution with variables $x_e^\star$, and the cost $m_Q^\star = \sum_{e\in E} x_e^\star\omega_e$. If all the $x_e^\star \in \{0,1\}$, we would have a solution of the program (25), and thus of the Min-Multi-Cut. If we try the graph from the previous example, we see that the optimum of (27) is integral. This looks promising. Let us continue with the following experiment: take all 510489 cubic (i.e. each vertex has three neighbors) graphs on 20 vertices, set all weights to 1, and for each of the graphs we randomly select $k \in \{1,\ldots,20\}$ pairs s_i, t_i. The results are as follows: from among the 510489 instances, 268178 (52.5%) have the relaxed program with integral optimal solution, 135328 (26.5%) half-integral, and overall there are 458008 (89.7%) instances with the smallest non-zero value of the optimal solution $\geq \frac{1}{4}$. So if we simply round everything non-zero up, in 89.7% we get a 4-approximation at the worst. Before getting too enthusiastic, let us have a look at the remaining 10.3% of instances. It may actually happen that these cases are not so rare as they appear, and it is only due to the selection of our dataset that they are scarcely populated. The bad instances have the common feature that there are many pairs s_i, t_i (large k). So let us take an arbitrary graph, and try to solve an instance where the s_i, t_i pairs are all pairs of vertices with distance at least four.



The graph on the left has 20 vertices and 30 edges (all edges have weight 1). Consider an instance of Min-Multi-Cut where all vertices with distance at least four must be disconnected (the diameter of the graph is 5, and there are 22 paris with distance at least four). The optimal solution of the relaxed LP is in the middle (the numbers are multiplies of $\frac{1}{15}$) with cost 7. Since only 5 edges (the dotted ones) in the solutions have zero value, by rounding everything up we get a solution with cost 25. On the right there is an integral solution with cost 8: after removing the eight dotted edges, the graph is split into three connected components, and each of them has diameter at most three.

Since we see that a simple "round-everything-nonzero-up" approach can yield poor results, it is time to think about a more refined "rounding" procedure. Denote $m^\star$ the optimum cost of the Min-Multi-Cut solution; ti holds $m^\star_Q \leq m^\star$. We show how to "round" the values of $x^\star_e$ (i.e. how to select edges into the cut) in such a way that the obtained solution of the Min-Multi-Cut instance has cost at most $4\ln(2k)m^\star_Q$.

Our "rounding" algorithm iteratively "bites out" sets $V_1, v_2, \ldots, V_k$ out of the graph. In the first iteration, the algorithm finds a set V_1 such that $s_1 \in V_1$, $t_1 \notin V_1$, and no pair s_i, t_i is included in V_1. Edges separating V_1 from the remainder of the graph are added into the cut, and the vertex set is shrunk to $V - V_1$. In the i-th iteration, if either s_i or t_i are not present in the remaining vertices (at least one of them is present there) then $V_\emptyset$, else find V_i such that $s_i \in V_i$ and $t_i \notin V_i$, and no pair s_j, t_j is contained in V_i. The vertices V_i are then removed from the graph (i.e. edges separating V_i from the remaining vertices are added to the cut), and the next iteration starts. This way, the algorithm constructs a cut that separates every pair s_i, t_i. The question is how to select the sets V_i, and how the knowledge of the optimal solution of the relaxed program can help in it.

Consider the values $x^\star_e$, and imagine the optimum solution of (26 as a network of pipes: the edge e forms a pipe with length $x^\star_e$, and cross-section ω_e, so it has volume $\omega_e x^\star_e$ For an edge $e = (u, v)$ denote the distance $d(u, v) = x^\star_e$, and in a natural way extend the notion of the distance to any two vertices $d(v_1, v_2)$. When the algorithm selects a set V_i, it has to cut all the pipes that connect V_i with the rest of the graph, and to cut a single pipe e it has to pay a cost proportional to the cross-section ω_e. From the feasibility of the relaxed solution we know that $d(s_i, t_i) \geq 1$ for all pairs s_i, t_i. Moreover, the optimum solution of (26) minimizes the overall volume of edges, i.e.

$$\Psi := \sum_{e=(u,v)\in E} \omega_e d(u, v)$$

Denote by $G'_r = (V'_r, E'_r)$ the graph obtained by removing the sets $V_1, \ldots, V_{r-1}$ from G. Define the ball with radius ρ centered in a vertex v in a natural way:

$$\mathcal{B}_\rho(v) = \{u \in V'_r \mid d(u, v) \leq \rho\}$$

If G'_r contains both s_i, and t_i, the selected set V_r will be some ball with a suitable radius ρ centered in s_r. It remains to show how to set the value of ρ. To start with, we want $\rho < 1/2$: this ensures that no pair s_j, t_j will be included in V_r (the distance between s_i, and t_i is at least 1). Let the interior edges of the ball are

$$\mathcal{E}_\rho(v) = \{(w, z) \in E'_r \mid w, z \in \mathcal{B}_\rho(v)\}$$

and the edge boundary is

$$\overline{\mathcal{E}}_\rho(v) = \{(w,z) \in E'_r - \mathcal{E}_\rho(v) \mid w \in \mathcal{B}_\rho(v) \vee z \in \mathcal{B}_\rho(v)\}$$

When defining the volume of a ball, we add to it, from technical reasons that become apparent in a moment, a term Ψ/k:

$$V_\rho(v) = \frac{\Psi}{k} + \sum_{(w,z)\in\mathcal{E}_\rho(v)} \omega_{(w,z)} d(w,z) + \sum_{(w,z)\in\overline{\mathcal{E}}_\rho(v)} \omega_{(w,z)} \left(\rho - \min(d(v,w), d(v,z))\right)$$

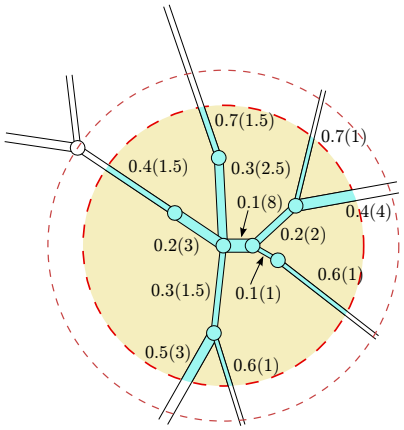
The interior edges of a ball contribute their full volume to the volume of the ball, and the edges from the boundary contribute the respective part. We are, of course, interested in the cost of the ball, i.e. the size of its edge-cut:

$$C_\rho(v) = \sum_{(w,z)\in\overline{\mathcal{E}}_\rho(v)} \omega_{(w,z)}$$

We are finally getting to the point how to set the radius ρ for the set V_r. We want to cut out a ball with minimum unit cost, i.e.

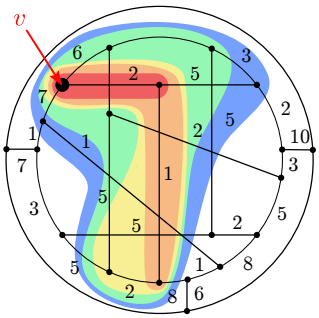
$$F_\rho(v) = \frac{C_\rho(v)}{V_\rho(v)}$$

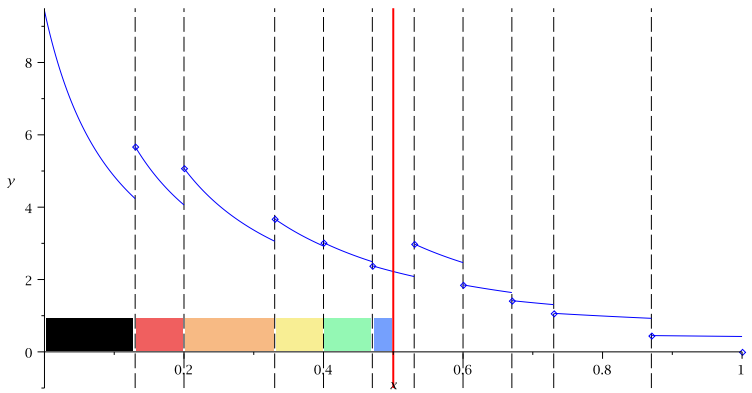
The mapping $F : \rho \mapsto F_\rho(v)$ for a fixed vertex v is discontinuous in values ρ for which there exists a vertex w with distance ρ from v. On every interval between the discontinuity points F is differentiable and decreasing. Hence, it is easy to find the minimum of $F_\rho(s_i)$ on the interval $(0, 1/2)$: just take the smallest of the values $F_\rho(s_i)$ for $\rho = \delta(s_i, u) - \varepsilon$. It remains to show that the cut obtained this way is in fact a good approximation. To this end we show the following statement:



The interpretation of the relaxed solution as a network of pipes. The edges are labeled by their length ($x^\star_e$), and the cross-section (ω_e) is in parentheses. The ball with radius 0.4 has cost 13 and volume $\frac{\Psi}{k} + 4.65$. If the radius is increased to 0.6, the cost remains the same, but the volume is increased.

$$\forall v \exists \rho < 1/2 : \; F_\rho(v) \le 2\ln(2k) \tag{28}$$





The mapping F for one vertex v from the previous example (the lengths are multiples of $1/15$). The overall volume (cost of the optimum solution) is $\Psi = 7$, and $k = 22$. Up to the radius $\frac{2}{15} \approx 0.13$, the interior of the ball is empty and the size of the cut is 3, so for $0 \leq \rho < \frac{2}{15}$ we have $F_\rho(v) = \frac{3}{\frac{7}{22}+3\rho}$. For $\frac{2}{15} \leq \rho < \frac{3}{15} = 0.2$ the interior contains two vertices and an edge of length $\frac{2}{15}$; th size of the cut is 4, so on this interval we have $F_\rho(v) = \frac{4}{\frac{7}{22}+\frac{2}{15}+2\rho+2(\rho-\frac{2}{15})} \approx \frac{4}{0.18+4\rho}$. Then up to the radius $\frac{5}{15} = \frac{1}{3}$ the interior contains three vertices and two edges with an overall volume 0.2, while the cut has size 5, and so on.

The property (28) implies the required approximation: let the algorithm select balls $\mathcal{B}_{\rho_1}(s_{i_1}), \ldots, \mathcal{B}_{\rho_h}(s_{i_h})$. The cost of the cut is

$$m = \sum_{j=1}^{h} C(s_{i_j}, \rho_j) \leq 2\ln(2k) \sum_{j=1}^{h} V(s_{i_j}, \rho_j),$$

where in the last sum the terms Ψ/k from the definition of volume add up to at most Ψ, and the remaining terms in the volumes add up to at most another Ψ; hence, $m \leq 2\ln(2k)2\Psi$.

To finish the proof we prove the property (28): let us fix a vertex v and consider all functions as dependent on the radius ρ. Whenever $V(\rho)$ is differentiable, it holds $V'(\rho) = C(\rho)$,and hence

$$F(\rho) = \frac{V'(\rho)}{V(\rho)} = \left[\ln V(\rho)\right]'.$$

We show (28) by contradiction: suppose that for each $\rho < 1/2$ it holds $F(\rho) > 2\ln(2k)$. If F is differentiable on the interval $(0, 1/2)$, we have

$$\left[\ln V(\rho)\right]' > 2\ln(2k).$$

Take the integral from 0 to $1/2$ on both sides, and get

$$\ln \frac{V\left(\frac{1}{2}\right)}{V(0)} > \ln 2k$$

yielding a contradiction

$$V\left(\frac{1}{2}\right) > 2kV(0) = 2\Psi$$

since $V(\rho) \leq \Psi + \Psi/k$. If F is not differentiable on the whole interval $(0, 1/2)$, just repeat the previous steps on each of the continuity interval (there are finitely many), and sum up the results.

The preceding musings can be summarized as a statement:

Theorem 2.15. *There is a polynomial-time algorithm for the problem* Min-Multi-Cut*, that always returns a solution with cost at most* $4\ln(2k)$ *times the optimum.*

The guarantee we just proved seems to be quite weak, and we would expect that in many cases the we could get a better approximation. Can we, however, prove a stronger guarantee? We now show that it is not possible. Take a regular graph G with n vertices, such that all vertices have degree $d \geq 3$. Select some parameter α, and consider and Min-Multi-Cut instance on G, in which all edges have weight 1, and the pairs s_i, t_i that must be disconnected are all pairs of vertices with distance at least α. Setting $x_e = \frac{1}{\alpha}$ for each edge e, we get a feasible solution of the program (26), hence $m^\star_Q \leq \frac{n}{\alpha}$. We show how to force any feasible integral solution to have cost at least n. Let M be the optimum integral solutions, i.e. the set of edges whose removal disconnects all pairs of vertices with distance at least α. After removing M from G, we get a graph $G' = (V, E - M)$ with some number of connected components, for which it holds

Lemma 2.16. *Every connected component of* G' *has at most* d^α *vertices.*

Proof: Suppose for the sake of contradiction that there is a component K with more than d^α vertices. Take any vertex $v \in K$. Since v has degree d in G, it has at most d neighbors in K. Each of them has again at most d neighbors in K, implying that there may be at most $d+d(d-1) < d+d^2$ vertices with distance at most 2 from v. Let us continue by induction, getting that in G (and thus also in K) there may be at most $d + d^2 + \cdots + d^{\alpha-1}$ vertices with distance at most α from v. However, $1 + d + d^2 + \cdots + d^{\alpha-1} = \frac{d^\alpha - 1}{d-1} < d^\alpha$, so, since K has more than d^α vertices, there is some vertex w in K that is from v further than α. But then v, and w are not allowed to be in the same connected component, a contradiction. □

Now let $S_1, \ldots, S_h$ be connected components of G'. Denote $\delta(S)$ the edge boundary of a set S, i.e. the edges with one endpoint in S, and the other outside. Since every edge has two endpoints, it holds $2|M| = \sum_{i=1}^{h} |\delta(S_i)|$. Moreover, since every vertex is in some connected component (only edges were removed), we have $\sum_{i=1}^{h} |S_i| = n$. In our next investigation we shall use a handy set of weird objects: expander graphs. They are graphs in which the edge boundary of any set is "large".

Definition 2.17. A graph $G = (V, E)$ is called an *expander* if for any set $S \subset V$ of vertices

$$|\delta(S)| \geq \min\{|S|, |\overline{S}|\},$$

where $\overline{S} = V - S$.

Now let us suppose that our graph G is an expander, and set $\alpha = \lfloor \log_d n/2 \rfloor$. Since $d^\alpha \leq n/2$, any connected component of $(V, E - M)$ has at most $n/2$ vertices, and thus we get

$$|M| = \frac{1}{2} \sum_{i=1}^{h} |\delta(S_i)| \geq \frac{1}{2} \sum_{i=1}^{h} |S_i| = \frac{n}{2}.$$

To sum it up, we have an instance on a d-regular graph with n vertices, where the optimal cut has $\Omega(n)$ edges, but there is relaxed solution with cost $O\left(\log d \frac{n}{logn}\right)$. At the same time, we have $k = \Theta(n^2)$ pairs of s_i, t_i, since for each vertex there are at least $n/2$ vertices with distance at least α. If we want to decouple the parameters n, and k, and make them independent, it is sufficient for some ℓ to replace each edge by a path of length ℓ, and keep the set of pairs s_i, t_i. The number of vertices will grow, but the approximation ratio remains the same. So we see that no "rounding" algorithm that would work an all instances can have a better guarantee than to get a $O(\log k)$-multiple of the optimum.
The last thing that remains open is whether expanders actually do exist. They do, and there are actually many of them, but they are somewhat shy. It is quite hard to find a deterministic construction that would produce expanders. On the other hand, it is not too involved to prove (although we are not going to do it now) that the majority of regular graphs are expanders.

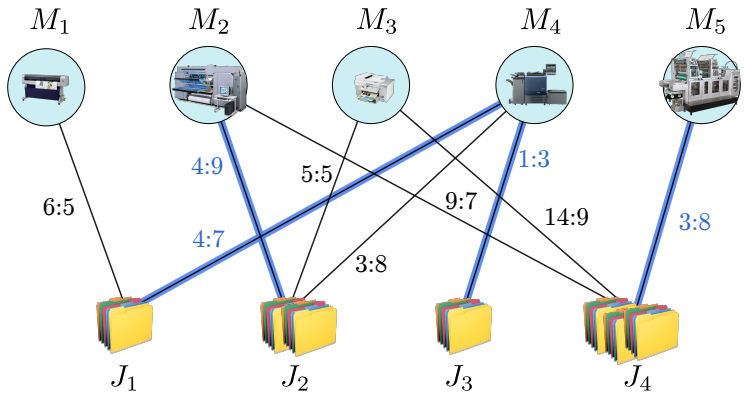
Iterated rouning

In the case of Min-Vertex-Cover we solved the relaxed program, and we were able to show that in any basic solution each variable has some good property (in our case, it was half-integrality, i.e. $x_i \in \{0, \frac{1}{2}, 1\}$), based on which we could bound the error of the rounding procedure. Now we shall look at a case when this nice property holds only in some of the variables: we can fix the values of variables we are happy with, create a smaller linear program from the remaining variables, solve it, and repeat the same approach. We end up with iteratively solving (hopefully) smaller and smaller programs, in each iteration fixing the values of some variables, until we get a solution.
We show this technique on an example. Imagine a printing company that operates different printers (maybe based on different technologies, plotters, offset printers, laser printers, inkjets, etc.). The company receives a contract to print a number of copies of a number of different items (books, leaflets, stickers, etc.). Each such job can be possibly performed on several printers, incurring a different time and cost on each of them (e.g. a black and white text can be printed on an offset

printer, a laser printer, or a color plotter; the last option, however, would take long and be quite expensive). The whole contract has a deadline, and the goal is to schedule the jobs to printers in such a way that the deadline is met, and at the same time, the cost is minimized. This lead to the following generalized problem:

Definition 2.18. Given is a set M of machines, and a set J of jobs, where $|M| = m$, and $|J| = n$. Also, for each pair $i \in M$, $j \in J$ there is given a time $t_{ij} \geq 0$ and cost $c_{ij} \geq 0$ of processing job j on machine i. Finally, a number T is given. The goal of the problem Generalized-Assignment is to assign a machine $u_j \in M$ to each job $j \in J$, such that the overall cost of the assignment $\sum_{j \in J} c_{u_j j}$ is minimized. Moreover, the overall processing time must meet the deadline, i.e. for each machine $i \in M$ it holds $\sum_{j:u_j=i} t_{ij} \leq T$.

A natural visualization of the problem is a bipartite graph $G = (V, E)$, with bipartition $V = M \uplus J$, where of a pair $i \in M$, $j \in J$ there is an edge $(i, j) \in E$, if $t_{ij} \leq T$.



Optimal assignment with cost 12 for the deadline $T = 10$. The first number on an edge denotes the corresponding processing cost, the second number the processing time.

The problem Generalized-Assignment is hard to solve optimally. Even for a very special case where there are only two machines with the processing times of all jobs equal on both of them, i.e. $p_{1j} = p_{2j} = p_j$, and $\sum_{j \in J} p_j = 2T$, it is an NP-complete problem to decide whether the jobs can be assigned to the machines in such a way that every machine has assigned exactly time T (the problem is known as Min-Partition[8]. This means we don't expect an efficient algorithm to optimally solve the Generalized-Assignment. However, we present a solution of a somewhat easier task: imagine that our printing company promised the contract to be produced *"in 5 to 10 days"*. We are going to solve the original task with the deadline $T = 5$ days, but it won't matter if the deadline is slightly overdue, unless it is ready within 10 days. In what follows we present an algorithm for the Generalized-Assignment problem that always finds solution with optimum cost, and the deadline may be overdue at most by a factor of 2 (i.e. the processing times of all machines are at most $2T$). It is important to note that this is not the same as solving the problem with limit $2T$ instead of T, since we are comparing ourselves to the optimum of the original problem; the optimum cost of the problem with deadline $2T$ may be much much smaller.

First, let us formulate the problem Generalized-Assignment as ILP. For each pair $i \in M$, $j \in J$ such that $(i, j) \in E$ we introduce a selector variable $x_{ij} \in \{0, 1\}$ which tells us whether the job j is assigned to machine i. The Definition 2.18 directly translated to an ILP formulation:

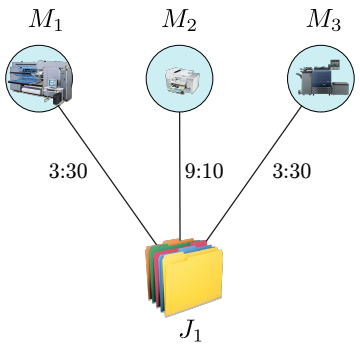
[8]The problem Min-Partition is defined as follows: for a set of natural numbers $A = \{a_1, \ldots, a_n\}$ decide, whether there exists a partition $A = B \uplus C$ such that $\sum_{x \in A} x = \sum_{x \in B} x$.

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j)\in E} x_{ij}c_{ij} & & & \\ \text{subject to} \quad & \sum_{i\in M} x_{ij} & = & 1 & \forall j \in J \\ & \sum_{j\in J} t_{ij}x_{ij} & \leq & T & \forall i \in M \\ & x_{ij} \in \{0,1\} & & & \forall i \in M,\ \forall j \in J \end{aligned} \tag{29}$$

The relaxation of the program (29) can be interpreted as a setting where a job does not have to be assigned to single machine; instead parts of the job can be assigned to various machines. If we try to solve the instance from the above example we may start feeling optimistic, since the optimum relaxed solution is integral. Our optimism is, of course, in vain, since it suffices to set $T = 7$ instead of $T = 10$, and the optimum assignment $J_1 \mapsto M_1$, $J_2 \mapsto M_3$, $J_3 \mapsto M_4$ a $J_4 \mapsto M_2$ has cost 21, while there is a solution of the relaxed program

$$x_{11} = \frac{3}{7} \quad x_{22} = \frac{49}{72} \quad x_{24} = \frac{1}{8} \quad x_{32} = \frac{23}{72} \quad x_{34} = 0 \quad x_{41} = \frac{4}{7} \quad x_{42} = 0 \quad x_{43} = 1 \quad x_{54} = \frac{7}{8}$$

with cost $\frac{7019}{504} \approx 13.9$. We immediately see where the problem is: e.g. M_5 is an cost-efficient but slow machine. In the integral solution we cannot use it, but the relaxed solution may put a part of J_4 there, that exactly fills the time limit. In spite of this we still see that some of the values in the optimum solution of the relaxed program are integral, and we are going to base our strategy on that. We want to fix some integral values, and solve the rest by some similar linear program. We are thus interested if there can be an instance with no integral values in the optimum relaxed solution; such instance would cause problems. The picture on the right shows that it may indeed be the case, so as it seems we are stuck. Back to the drawing board. However, we still save the idea by observing that instances where the optimum solution of the relaxed problem has no integral values must have a very special form that can be used to shrink the program in another way even if no variable is fixed in the given iteration. We would soon realize that instead of the original relaxed program we need to solve a slightly more general one. We save the reader the detour and directly present the generalized program. Instead of forcing all machines to finish before the deadline, we maintain a subset $M' \subseteq M$ that must finish on time; the other ones may work without restrictions. Moreover, every machine $i \in M'$ may have a potentially different deadline. The resulting program is as follows:



An optimal assignment for the deadline $T = 10$ is M_2 with cost 9. Optimum solution of the relaxed program is $x_{11} = x_{21} = x_{31} = \frac{1}{3}$ with cost 5.

(30)

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j)\in E} x_{ij}c_{ij} & & & \\ \text{subject to} \quad & \sum_{i\in M} x_{ij} & = & 1 & \forall j \in J \\ & \sum_{j\in J} t_{ij}x_{ij} & \leq & T_i & \forall i \in M' \\ & x_{ij} & \geq & 0 & \forall i \in M,\ \forall j \in J \end{aligned} \tag{31}$$

The relaxation of (29) is a special case of (31) for $M' = M$ and $T_i = T$. In the following we denote by $deg(k)$ for $k \in J \cup M$ the degree of the job or machine in the graph G (i.e. the number of

incident edges). The key to the whole algorithm is the following characterization of the optimal solutions:

Lemma 2.19. *Let* $\mathbf{x}$ *be a basic solution of (31), where for all* i, j *is* $0 < x_{ij} < 1$. *Then there is a machine* $i \in M'$ *for which either* $deg(i) \leq 1$, *or* $deg(i) = 2$ *and* $\sum_{j \in J} x_{ij} \geq 1$.

Lemma 2.19 tells us that if, after solving the program (31) we are in a situation where there is no integral variable in the solution, we have a machine that either can process only one job, or can process two jobs, and it is assigned a part of each (moreover, these parts sum up to ≥ 1). In order to prove Lemma 2.19 we use the following claim:

Lemma 2.20. *Let* $\mathbf{x}$ *be a basic solution of (31) where all* $x_{ij} > 0$. *Then there is a set* $M'' \subseteq M'$ *of* $|M''| = |E| - |J|$ *machines such that all machines* $i \in M''$ *are working all their available time, i.e.* $\sum\limits_{j \in J} t_{ij} x_{ij} = T_i$.

Proof: Recall the definition of the basic solution (Definition 1.2). First, we required the program to be in a normal form $\max\limits_{\mathbf{x}} \{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq 0\}$ where A has full rank (i.e. the rows are independent). So let us transform the program (31) into the required normal form by changing minimization to maximization, and by introducing a slackness variable s_i for each constraint $\sum_{j \in J} t_{ij} x_{ij} \leq T_i$ transforming it into a constraint $\sum_{j \in J} t_{ij} x_{ij} + s_i = T_i$. This way we obtain a matrix of constraints of dimensions $(n + m') \times (|E| + m')$ in the form

$$A = \left(\begin{array}{c|c} B & 0 \\ \hline C & I \end{array}\right),$$

where $B \in \mathbb{R}^{n \times |E|}$ is the matrix of job constraints, $C \in \mathbb{R}^{m' \times |E|}$ is the matrix of machine constraints, and I is the identity matrix $m' \times m'$ involving the slackness variables s_i. The reader easily verifies that the rows of A are linearly independent.

For example the instance (30) yields a program

$$\max_{\mathbf{x} \in \mathbb{R}^6} \{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq 0\}$$

where

$$\mathbf{c} = \begin{pmatrix} -3 \\ -9 \\ -3 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} \quad A = \left(\begin{array}{ccc|ccc} 1 & 1 & 1 & 0 & 0 & 0 \\ \hline 30 & 0 & 0 & 1 & 0 & 0 \\ 0 & 10 & 0 & 0 & 1 & 0 \\ 0 & 0 & 30 & 0 & 0 & 1 \end{array}\right) \quad \mathbf{b} = \begin{pmatrix} 1 \\ T \\ T \\ T \end{pmatrix}$$

Following Definition 1.2, the basis has size $n + m'$, and all non-basic entries are zeroes. Since we suppose all $x_{ij} > 0$, any basic solution must have $|E| + m' - (n + m') = |E| - n$ zero variables s_i. Each zero slackness variable s_i provides one machine for which $\sum_{j \in J} t_{ij} x_{ij} + s_i = T_i$. □

Proof of Lemma 2.19: Consider a basic solution of (31), where $0 < x_{ij} < 1$ for each i, j, and moreover $deg(i) \geq 2$ for all $i \in M'$. Each job $j \in J$ must have $deg(j) \geq 2$: since $\sum_{i \in M} x_{ij} = 1$, having $deg(j) = 1$ would imply some $x_{ij} = 1$. For the set M'' from Lemma 2.20 we have

$$|J| + |M''| = |E| = \frac{\sum\limits_{j \in J} deg(j) + \sum\limits_{i \in M} deg(i)}{2} \geq \frac{\sum\limits_{j \in J} deg(j) + \sum\limits_{i \in M'} deg(i)}{2} \overset{(\clubsuit)}{\geq} |J| + |M'| \geq |J| + |M''|$$

The inequality (♣) holds because for $i \in M'$ the $deg(i) \geq 2$ holds by assumption, and $deg(j) \geq 2$ we have just shown for all $j \in J$. Since the first and the last expressions in a sequence of inequalities are equal, there must be equality everywhere. As $M'' \subseteq M'$, we have $M' = M''$. Moreover, since

all vertices from J and M' have degree at least 2, the only possibility to keep the equality in (♣) is when it holds $deg(j) = 2$ for all $j \in J$ and $deg(i) = 2$ for all $i \in M'$. To reach equality also in the previous inequality, it must be $deg(i) = 0$ for all $i \in M \setminus M'$.
But then it must be that G is composed of isolated vertices, and vertices of degree two only, i.e. forms a set of disjoint cycles. Since G is bipartite, every cycle C has even length, and so it contains an equal number k machines and k jobs. Since for each job j it holds $\sum_{i \in M'} x_{ij} = 1$, it is $\sum_{(i,j) \in C} x_{ij} = k$ and so there must be a machine i in C, such that $\sum_{j \in J} x_{ij} \geq 1$. □

Given the characterization of the basic solutions from Lemma 2.19, the outline of an algorithm starts to materialize. If in a solution of the program (31) there is some variable $x_{ij} = 1$ we assign the job j to the machine i; we are done with the job j, and the deadline T_i of the machine i is decreased accordingly (this is the reason why we wanted to have different deadlines for different machines in the program (31). If, on the other hand, there is some variable $x_{ij} = 0$, we remove the edge from the graph, obtaining a smaller graph (and, in turn, smaller LP for the next iteration). Further, it there is some machine $i \in M'$ that can process only one job ($deg(i) = 1$) we can disregard its deadline: we can suppose without loss of generality that for each edge, the associated time fits the deadline of the machine (otherwise the edge can never be used, and can be discarded), which means that no matter whether the final solution will assign something to machine i or not, i's deadline will not be exceeded. Finally, the last possible situation that may occur after solving the program (31) is when there is some $i \in M'$ with degree 2 and $\sum_{j \in J} x_{ij} \geq 1$. In this case we can also disregard the deadline: $deg(i) = 2$, so in any feasible solution it holds $\sum_{j \in J} x_{ij} \leq 2$. Since in our solution $\sum_{j \in J} x_{ij} \geq 1$, and the deadline is still met, no matter what happens in the final solution, the deadline may be overstretched at most by a factor of 2. In both cases we obtain a new program that smaller in the sense that the size of M' has decreased. The described algorithm for the Generalized-Assignment problem looks as follows:

---

1 $M' := M$, $\forall i : \ T_i := T$, $F := \emptyset$ (F is the set of assigned edges)

2 while $J \neq \emptyset$

  3 nech **x** is optimum solution of program (31),
  perform one of the cases

  4a if there exists $x_{ij} = 0$,
  remove edge (i, j) from G

  4b if there exists $x_{ij} = 1$,
  $F := F \cup \{(i, j)\}$, $J := J \setminus \{j\}$, $T_i := T_i - t_{ij}$

  4c else let $i \in M'$ such that $deg(i) \leq 1$ or $deg(i) = 2$ and $\sum_{j \in J} x_{ij} \geq 1$
  $M' := M' \setminus \{i\}$

Every iteration of the cycle on line 2 decreased the value of $|E| + |J| + |M'|$, so the algorithm terminates after a polynomial number of iterations. It is easy to see that in the end, every job is assigned to some machine. What remained to be shown is that the performance of the algorithm is a we hoped:

Theorem 2.21. *The presented algorithm solves* Generalized-Assignment *in polynomial time. The found solution has optimal cost, and each machine finished its work no later than in time* $2T$.

Proof: We already shown that the algorithm works in polynomial time, and produces a feasible solution. Now we shall show that this solution is indeed optimal. Let c be the cost of the optimum solution of the program (31) for $M' = M$ and all $T_i = T$. Clearly, $c \leq OPT$. We show by induction that in each iteration of the main loop, the cost of the already assigned jobs F together with the cost of the current solution of (31) from line 3, is at most c. For the induction basis note that in the first iteration the statement holds trivially. Now consider an arbitrary iteration. If the case 4a is executed, neither F nor the optimum of the program (31) is changed. In the case 4b,

the cost of F is increased by c_{ij}, and the cost of the optimum solution of (31) is decreased at least by $c_{ij}x_{ij} = c_{ij}$. In the case 4c, the cost of F is not affected, and the cost of the optimum of (31) is not increased.

Finally, let us show that the overall time spent by any machine is at most $2T$. Denote by $F_{i,\ell}$ the time induced on machine i by the jobs assigned to F during the first $\ell - 1$ iterations, and similarly by $T_{i,\ell}$ the value of T_i at the beginning of the ℓ-th iteration. Fix some machine i. During the first iterations, while $i \in M'$ it holds $T_{i,\ell} + F_{i,\ell} \leq T$ (if a job is assigned to i, its deadline is updated correspondingly on line 4b). Hence, consider the iteration ℓ in which i is removed from M' on line 4c. There are two ways this may happen:

1. $deg(i) \leq 1$. Let j be the only job connected[9] in the iteration ℓ to the machine i. How much time may i spend in the final solution? Surely the time it has already allocated, i.e. $T_{i,\ell}$, plus maybe t_{ij} (it the job j happens to be assigned to i in the end), but surely not more, since no other job is connected to i. Hence, the overall time is at most $F_{i,\ell} + t_{ij} \leq T_{i,\ell} + F_{i,\ell} + t_{ij} \leq 2T$; the last inequality is due to the fact that for all edges it holds $t_{ij} \leq T$, and up to the ℓ-th iteration the machine i was in M' so $T_{i,\ell} + F_{i,\ell} \leq T$.

2. $deg(i) = 2$ **a** $\sum_{j \in J} x_{ij} \geq 1$ Let j_1, j_2 be the only two jobs connected with i. Again, we ask how much time may i need in the final solution. Clearly at most $F_{i,\ell} + t_{ij_1} + t_{ij_2}$. At the beginning of the ℓ-th iteration we had $F_{i,\ell} + T_{i,\ell} \leq T$. When solving (31) in the ℓ-th iteration, $T_{i,\ell}$ is the limit for machine i, so if $\mathbf{x}$ is the solution from line 3, it holds $t_{ij_1}x_{ij_1} + t_{ij_2}x_{ij_2} \leq T_{i,\ell}$. Hence, it holds also $F_{i,\ell} \leq T - t_{ij_1}x_{ij_1} + t_{ij_2}x_{ij_2}$, implying that i never uses more time than

$$\begin{aligned} T - t_{ij_1}x_{ij_1} + t_{ij_2}x_{ij_2} + t_{ij_1} + t_{ij_2} &\leq \\ T + (1 - x_{ij_1})t_{ij_1} + (1 - x_{ij_2})t_{ij_2} &\leq \\ T + (1 - x_{ij_1})T + (1 - x_{ij_2})T &\leq \\ T(3 - x_{ij_1} - x_{ij_2}) &\leq 2T \end{aligned}$$

where the last inequality follows from the assumption that $\sum_{j \in J} x_{ij} \geq 1$.

□

2.3 Randomized rounding

In this section we shall continue with the same general approach as before: if we want to solve an optimization problem (a discrete one), we formulate it as an ILP, solve the relaxed LP version, and think of some way how to use the optimum solution of the relaxed program to obtain some approximation of the original problem. Oftentimes in our ILP formulation we use indicator variables, i.e. variables of the form $x_i \in \{0, 1\}$, that are transformed in the relaxed problem into the form $0 \leq x_i \leq 1$. If the indicator variable is interpreted as a flag whether, say, a given item is selected, then it is natural to interpret the relaxed variable as a probability of selecting the item. In the following we shall build on this intuition, and show how to use in in the design of approximation algorithms. The general approach is to first construct a randomized rounding algorithm that returns a good approximation in expectation, and then we show how to transform it into deterministic algorithm with the same approximation guarantees (this process is called *derandomization*). As usual, let us start with a concrete example:

Max-Sat

When we were speaking about integer linear programs, we mentioned the NP-complete problem Sat (see Definition 2.2). Now we shall consider its optimization version:

[9] To be precise we should index also the graph G by the number of iteration ℓ, since G evolves during the computation. To make the notation less cumbersome, we rely on the reader to understand from the context which graph G is referred to.

Definition 2.22. Given n boolean variables $x_1, \ldots, x_n \in \{true, false\}$, we call *literal* either a variable x_i or its negation $\overline{x_i}$. A *clause* is a disjunction of literals $C = l_1 \vee l_2 \vee \cdots \vee l_{k_C}$, a *formula in conjunctive normal form (CNF)* is a conjunction of m clauses $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$. The problem MAX-SAT is a maximization problem where the input is a formula F (in CNF), in which each clause C_i has a non-negative weight ω_i. The goal of the problem is to find a truth assignment for the variables $x_1, \ldots, x_n$ such that the overall weight of satisfied clauses is maximized.

If all clauses have unit weights, then the formula is satisfiable if and only if the optimum is m; so an optimal algorithm for MAX-SAT can be used to solve SAT. In the sequel let us denote the overall weight of all clauses Z, i.e.

$$Z = \sum_{i=1}^{m} \omega_i.$$

It is easy to see that in any case, an overall weight $Z/2$ can be easily obtained: first, let us set all variables to 0. If the weight is at least $Z/2$, we are done. Otherwise, set all variables to 1. Any clause that was not satisfied in the original solution is surely satisfied now, so we have guaranteed a solution with weight at least $Z/2$.

However, it is not so trivial to achieve a better guarantee. We show now how to obtain a solution with 3/4 of the overall weight.

Let us begin with an extremely simple randomized algorithm `A1` that does not use any linear programming, but just sets every variable to be 0 or 1 independently at random with probability 1/2. What is the expected overall weight of satisfied clauses? Let us denote this weight W; W is a random variable, and we are interested in its expected value $\mathrm{E}[W]$. Let X_i be an indicator random variable such that $X_i = 1$ if and only if C_i is satisfied. We have

$$W = \sum_{i=1}^{m} \omega_i X_i$$

and so from the linearity of expectation

$$\mathrm{E}[W] = \mathrm{E}\left[\sum_{i=1}^{m} \omega_i X_i\right] = \sum_{i=1}^{m} \omega_i \mathrm{E}[X_i] = \sum_{i=1}^{m} \omega_i \mathrm{Pr}[X_i = 1].$$

For a clause C_i, let us denote by $s(C_i)$ its size, i.e. the number of literals it contains. What is the probability that a particular clause C_i is satisfied? A clause is satisfied exactly if at least one of its literals is. Since each variable x_i is set independently at random with probability 1/2, the probability that C_i is *not* satisfied is $2^{-s(C_i)}$, so $\mathrm{Pr}[X_i = 1] = 1 - 2^{-s(C_i)}$. Now if we suppose that every clause has at least k literals for some k, then the algorithm `A1` is expected to gain a weight at least

$$\mathrm{E}[W] = \sum_{i=1}^{m} \omega_i \mathrm{Pr}[X_i = 1] = \sum_{i=1}^{m} \omega_i \left(1 - 2^{-s(C_i)}\right) \geq (1 - 2^{-k})Z. \tag{32}$$

In the general case if the formula can contain clauses with just 1 literal the approach did not help much – we still have only a guarantee $Z/2$. For bigger values of k, however, we are better off.

Derandomization

We have just developed a randomized algorithm that satisfies clauses with expected weight $(1 - 2^{-k})Z$ if every clause has at least k literals. Now the question is if we can develop a deterministic algorithm with the same guarantee, i.e. such that always satisfies clauses with weight at least $(1 - 2^{-k})Z$. The algorithm `A1` uses n random bits. During the derandomization process we shall successively modify `A1` such that we fix the first i (for $i = 1, 2, \ldots$) of the random bits to some values, and let the remaining ones be random, yielding a randomized algorithm using $n - i$ random bits. If we can design the transformation that fixes one additional random bit so that the expected

outcome does not decrease, in the end we obtain a deterministic algorithm with that performs as good as the is expected outcome of the randomized one.
How to fix the first bit? If we fix, e.g. $x_1 = 1$, the expected outcome is

$$\mathrm{E}[W \mid x_1 = 1] = \sum_{i=1}^{m} \omega_i \mathrm{Pr}[X_i = 1 \mid x_1 = 1].$$

Consider the i-th clause. What is the probability that C_i is satisfied if $x_1 = 1$? If C_i contains the literal x_i the probability is 1, if it contains the literal $\overline{x_i}$ the the probability is $1 - 2^{1-s(C_i)}$, and otherwise it is $1 - 2^{-s(C_i)}$. The value $\mathrm{E}[W \mid x_1 = 0]$ can be computed analogously. Since in the original algorithm the probability that $x_1 = 1$ was $1/2$, we get

$$\mathrm{E}[W] = \frac{1}{2}\mathrm{E}[W \mid x_1 = 0] + \frac{1}{2}\mathrm{E}[W \mid x_1 = 1],$$

and hence at least one of the values $\mathrm{E}[W \mid x_1 = 0]$, $\mathrm{E}[W \mid x_1 = 1]$ is at least as big as $\mathrm{E}[W]$. Thus we have an algorithm using $n - 1$ random bits with expected outcome at least $\mathrm{E}[W]$.
The derandomized algorithm `A1det` works in n iterations, and in the t-th iteration it maintains a tuple of constants $c_1, \ldots, c_{t-1}$. For each clauses C_i it computes $\Pr[X_i = 1 \mid x_1 = c_1, \ldots, x_{t-1} = c_{t-1}, x_t = 0]$ and $\Pr[X_i = 1 \mid x_1 = c_1, \ldots, x_{t-1} = c_{t-1}, x_t = 1]$. Based on this it computes the expected values $\mathrm{E}[W \mid x_1 = c_1, \ldots, x_{t-1} = c_{t-1}, x_t = 0]$ and $\mathrm{E}[W \mid x_1 = c_1, \ldots, x_{t-1} = c_{t-1}, x_t = 1]$, and sets c_t according to which of them is bigger. After n iterations the values $c_1, \ldots, c_n$ are returned as the solution.
The same derandomization approach (called *the method of conditional probabilities*) can be used in all cases where there is a deterministic algorithm to compute the expected outcome of the original randomized algorithm with any number of fixed first i random bits:

Theorem 2.23. *Consider a randomize algorithm $\mathtt{A_R}$ solving in polynomial time some optimization problem $\mathcal{P}$ such that on an input $\mathbf{x}$ it performs $R(\mathbf{x})$ random decisions $r_1, r_2, \ldots, r_{R(\mathbf{x})}$; the decisions are independent Bernoulli (binary) random variables with $\Pr[r_i = 1] = p_i$. Let us suppose that there exists a polynomial-time algorithm computing, for any i and a tuple of constants $c_1, \ldots, c_i \in \{0, 1\}$, the expected value $\mathrm{E}[\mathtt{A_R}(\mathbf{x}) \mid b_1 = c_1, \ldots, b_i = c_i]$. Then there exists a deterministic polynomial-time algorithm that computes a solution of $\mathcal{P}$ at least as good as $\mathrm{E}[\mathtt{A_R}(\mathbf{x})]$.*

In this general case a single iteration of the derandomized algorithm uses the fact that

$$\begin{aligned}\mathrm{E}[W \mid r_1 = c_1, \ldots, r_{i-1} = c_{i-1}] = &p_i \mathrm{E}[W \mid r_1 = c_1, \ldots, r_{i-1} = c_{i-1}, r_i = 1] + \\ &(1 - p_i)\mathrm{E}[W \mid r_1 = c_1, \ldots, r_{i-1} = c_{i-1}, r_i = 0],\end{aligned}$$

yielding that no matter what the p_i is, at least one of the expected values for $r_i = 1$, $r_i = 0$ is at least as big as the original expected value.

Algorithm for short clauses

Now let us get back to the Max-Sat problem. If every clause contained at least two literals, the algorithm `A1det` would already guarantee to achieve 3/4 of the overall weight (ant thus of the optimal solution). The problem is only with singleton clauses (containing only one literal). For a reader familiar with the decision problem Sat this may feel as some minor technical detail: if we have to decide whether the formula is satisfiable or not, singleton clauses play only in our favor, since they effectively fix a value of one variable. This is, however, not the case if we are not deciding satisfiability, but trying to find a maximizing truth assignment: the formula may contain, e.g. a clause x_i with some weight, and a clause $\overline{x_i}$ with some other weight, and it is not clear how to set the value of x_i.
Here finally comes linear programming to play. First, let us express Max-Sat as an ILP. In the most natural way, let us introduce a variable $p_i \in \{0, 1\}$ for each variable x_i. The goal is to maximize the weight of satisfied clauses, so we introduce another selector variable $z_i \in \{0, 1\}$ for

each clause C_i such that $z_i = 1$ if C_i is satisfied by the variables x. Now it is easy to write down the utility function $\sum_{i=1}^{m} \omega_i z_i$. What remains now is to design the linear constraints such that $z_i = 1$ only if C_i is satisfied, i.e. if no literal of C_i is true, then z_i must be zero. Denote C^+ the indices of the variables that appear in C in positive literals, and C^- the indices that appear in C in negated literals. We get the following program:

$$\begin{array}{lrcll} \text{maximize} & \sum_{i=1}^{m} \omega_i z_i & & & \\ \text{subject to} & \sum_{j \in C_i^+} p_j + \sum_{j \in C_i^-} (1 - p_j) & \geq & z_i & \forall i = 1, \ldots, m \\ & z_i, p_i & \in & \mathbb{Z} & \end{array} \tag{33}$$

and its relaxed version

$$\begin{array}{lrcll} \text{maximize} & \sum_{i=1}^{m} \omega_i z_i & & & \\ \text{subject to} & \sum_{j \in C_i^+} p_j + \sum_{j \in C_i^-} (1 - p_j) & \geq & z_i & \forall i = 1, \ldots, m \\ & z_i, p_i & \geq & 0 & \\ & z_i, p_i & \leq & 1 & \end{array} \tag{34}$$

Finally, we get to the point of this section – randomized rounding. The randomized algorithm `A2` solves the program (34) obtaining some optimal values $p_i^\star$, $z_i^\star$, and then set each variable x_i to 1 independently at random with probability $p_i^\star$. What is the expected value of `A2`? Similarly as with algorithm `A1` denote by W the overall weight gained by the algorithm, and let X_i be an indicator whether the i-th clause is satisfied, and let us estimate

$$\mathrm{E}[W] = \sum_{i=1}^{m} \omega_i \Pr[X_i = 1].$$

A clause C_i is not satisfied if no literal is true. A positive literal x_j is false with probability $1 - p_j^\star$ and a negative literal $\overline{x_j}$ is false with probability $p_j^\star$, so

$$\begin{aligned} \Pr[X_i = 1] &= 1 - \prod_{j \in C_i^+} (1 - p_j^\star) \cdot \prod_{j \in C_i^-} p_j^\star && (35) \\ &\geq 1 - \left(\frac{\sum_{j \in C_i^+} (1 - p_j^\star) + \sum_{j \in C_i^-} p_j^\star}{s(C_i)} \right)^{s(C_i)} && (36) \\ &= 1 - \left(1 - \frac{\sum_{j \in C_i^+} p_j^\star + \sum_{j \in C_i^-} (1 - p_j^\star)}{s(C_i)} \right)^{s(C_i)} \\ &\geq 1 - \left(1 - \frac{z_i^\star}{s(C_i)} \right)^{s(C_i)} && (37) \end{aligned}$$

where the inequality (36) is an application of the arithmetic-geometric inequality

$$\frac{a_1 + \cdots + a_n}{n} \geq \sqrt[n]{a_1 \cdot \cdots \cdot a_n},$$

and the inequality (37) follows from the constraints of the program (34). Our next immediate goal is to express a bound on $\Pr[X_i = 1]$ from the line (37) as a linear function, i.e. $\Pr[X_i = 1] \geq \beta z_i^\star$ for some β that may depend on C_i but is independent on $z_i^\star$. Denote

$$g_k(z) := 1 - \left(1 - \frac{z}{k} \right)^k$$

the function of z with $k \geq 1$ as a parameter. We have $g_k(0) = 0$ and $g_k(k) = 1$. Immediately, we can compute

$$\begin{aligned} g_k'(z) &= \left(1 - \frac{z}{k}\right)^{k-1} \\ g_k''(z) &= -\frac{k-1}{k-z}\left(1 - \frac{z}{k}\right)^{k-2}, \end{aligned}$$

so we see that $g_k(z)$ is on the interval $[0, k]$ increasing ($g_k'(z) > 0$), and concave ($g_k''(z) < 0$). Hence the whole graph of $g_k(z)$ on the interval $[0, 1]$ is below the line conecting points $[0, 0]$, and $[1, g_k(1)]$. Thus it follows that for $z \in [0, 1]$ it holds $g_k(z) \geq g_k(1)z$. Now returning to the inequality (37), we get

$$\Pr[X_i = 1] \geq g_{s(C_i)}(1) z_i^\star.$$

Denote

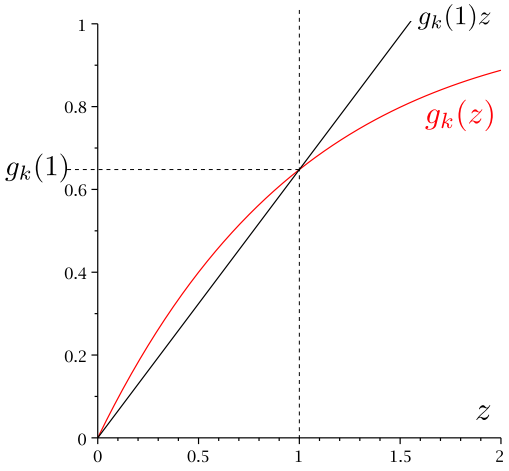
$$\beta(k) := g_k(1) = 1 - \left(1 - \frac{1}{k}\right)^k$$

a function of the variable k. For the expectation of `A2` we have

$$\mathrm{E}[W] \geq \sum_{i=1}^{m} \omega_i \beta\left(s(C_i)\right) z_i^\star. \tag{38}$$

With increasing h, the value $\beta(k)$ decreases, so if all clauses have at most k literals, we have

$$\mathrm{E}[W] \geq \beta(k) \sum_{i=1}^{m} \omega_i z_i^\star.$$



Now we can invoke Theorem 2.23 and construct a deterministic algorithm `A2det`; the corresponding conditional probabilities will be computed analogously according to line (35).

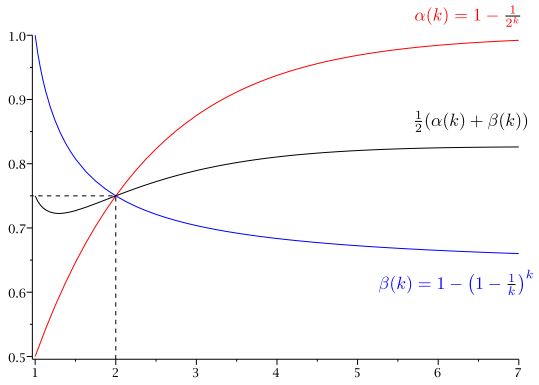
Final algorithm with guaranteed $3/4 \cdot Z$

So far we have two algorithms: `A1det` that works good on instances with long clauses, and `A2det` that performs well on instances with short clauses. What can we do on instances that contain both short and long clauses? We do the most simple of all thing: run both algorithms, and choose the better of the two results. From (32), (38), and from Theorem 2.23 we get for the outcome of the algorithms `A1det` and `A2det` on a formula F:

$$\texttt{A1det}(F) \geq \sum_{i=1}^{m} \omega_i \left(1 - \frac{1}{2^{s(C_i)}}\right) \qquad \texttt{A2det}(F) \geq \sum_{i=1}^{m} \omega_i \beta\left(s(C_i)\right) z_i^\star$$

Denote $\alpha(k) := 1 - 2^{-k}$, and bound the better of the two algorithms (i.e. the maximum of the values) by their average:

$$
\begin{aligned}
&\max\{\texttt{A1det}(F), \texttt{A2det}(F)\} \\
\geq &\frac{1}{2}(\texttt{A1det}(F) + \texttt{A2det}(F)) \\
\geq &\frac{1}{2}\sum_{i=1}^{m} \omega_i \left(\beta(s(C_i))z_i^\star + \alpha(s(C_i))\right) \\
\geq &\sum_{i=1}^{m} \omega_i z_i^\star \left(\frac{\alpha(s(C_i)) + \beta(s(C_i))}{2}\right)
\end{aligned}
$$



where the last inequality follows from the fact that $z_i^\star \leq 1$. Let us now focus on the i-th clause: let $s(C_i) = k$. What can we say about the average $\frac{1}{2}(\alpha(k)+\beta(k))$? For $k \in \{1, 2\}$, the $\frac{1}{2}(\alpha(k)+\beta(k)) = \frac{3}{4}$. For $k \geq 2$ the average is an increasing function, so we can sum it all up to

$$\max\{\texttt{A1det}(F), \texttt{A2det}(F)\} \geq \frac{3}{4}\sum_{i=1}^{m} \omega_i z_i^\star \geq \frac{3}{4}OPT.$$

3 Duality in linear programming

3.1 Duality? What is it?

In the previous parts we tried to persuade the reader that linear programming is a powerful and flexible tool for solving optimization problems, and, in particular, for the design of approximation algorithms. Now we shall introduce another crucial feature of linear programs, called *duality*, that is also often employed in the design of algorithms. Let us imagine that Robert wants to find the minimum of a function

$$f(x_1, x_2, x_3) := 10x_1 + 3x_2 + 5x_3$$

over the values x_1, x_2, x_3 that satisfy

$$\begin{aligned} 6x_1 + x_2 - x_3 &\geq 2 \quad (39)\\ 2x_1 + 2x_2 + 6x_3 &\geq 8 \quad (40)\\ 6x_1 + 3x_2 + 5x_3 &= 30\\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

Angela immediately sees that this is a linear program, and easily finds the minimum. The question is, how she can make Robert, ho knows nothing about linear programming, believe her. For an upper bound, she could present him with a *witness*, i.e. a particular solution (e.g. a vector $x_1 = 2$, $x_2 = 6$, $x_3 = 0$) that makes him believe that the minimum is at most some value (e.g. 38, in this case). But what can Angela do to find a witness of a lower bound? She may try the following argument: *"Look at the line (39). No matter how you choose the three non-negative numbers x_1, x_2, x_3, it will always hold $6x_1 \leq 10x_1$, $x_2 \leq 3x_2$, and $-x_3 \leq 5x_3$, so $6x_1 + x_2 - x_3 \leq f(x_1, x_2, x_3)$. However, according to (39) for any feasible solution it holds $6x_1 + x_2 - x3 \geq 2$, and so for any feasible solution (in particular, for the minimum) it holds $f(x_1, x_2, x_3) \geq 2$"*. Robert will accept this argument, and she can continue e.g. arguing that if the lines (39) and (40) are added together, the same argument can be used. Also, each line can be multiplied by some number (it must be a non-negative number for lines (39) and (40)) and then add them together: if it holds that the resulting linear combination of the left-hand sides of the constraints is component-wise smaller than the utility function f, the corresponding linear combination of the right-hand sides gives a lower bound on the minimum. When Robert is willing to accept all these arguments, Angela, in order to give the best possible lower bound, solves a linear program: to find a maximum of the function

$$g(y_1, y_2, y_3) := 2y_1 + 8y_2 + 30y_3$$

among those values y_1, y_2, y_3 that satisfy

$$\begin{aligned} 6y_1 + 2y_2 + 6y_3 &\leq 10\\ y_1 + 2y_2 + 3y_3 &\leq 3\\ -y_1 + 6y_2 + 5y_3 &\leq 5\\ y_1, y_2 &\geq 0 \end{aligned}$$

In this case, on one hand there is a witness $x_1 = 0$, $x_2 = 5$, $x_3 = 3$ asserting that the minimum of f is at most 30, on the other hand, there is witness $y_1 = y_2 = 0$, $y_3 = 1$ showing that minimum is at least 30. Was Angela just lucky that she managed to find witnesses for the exact optimum?
Now let us have a closer look at what is happening in the general case. Consider a following linear program that we shall call *primary*:

$$\begin{array}{rrcrcl}
\text{minimize} & c_1x_1 & +c_2x_2 & +\cdots & +c_nx_n & \\
\text{subject to} & a_{1,1}x_1 & +a_{1,2}x_2 & +\cdots & +a_{1,n}x_n & \geq & b_1 \\
& a_{2,1}x_1 & +a_{2,2}x_2 & +\cdots & +a_{2,n}x_n & \geq & b_2 \\
& \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
& a_{m,1}x_1 & +a_{m,2}x_2 & +\cdots & +a_{m,n}x_n & \geq & b_m \\
& & & & x_1,\ldots,x_n & \geq & 0
\end{array}$$

or, using a more concise notation

$$(P) : \min_{\mathbf{x}\in\mathbb{R}^n} \left\{ \mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \geq \mathbf{b},\ \mathbf{x} \geq \mathbf{0} \right\}.$$

We seek a lower bound on the minimum of (P) as a suitable linear combination of the constraints: the j-th constraint is multiplied on both sides by some coefficient $y_j \geq 0$, and the results are added together. We want to find the coefficients y_j such that the left-hand side is component-wise smaller than the utility function

$$\sum_{i=1}^{n} c_i x_i \geq \sum_{j=1}^{m} y_j \left(a_{j,1}x_1 + a_{j,2}x_2 + \cdots + a_{j,n}x_n\right) \geq \sum_{j=1}^{m} y_j \cdot b_j.$$

Since all $x_i \geq 0$, if $c_i \geq \sum_{k=1}^{m} y_j a_{j,i}$, the first inequality holds.So the best lower bound we can obtain from this approach is the solution of *dual* linear program

$$\begin{array}{rrcrcl}
\text{maximize} & b_1y_1 & +b_2y_2 & +\cdots & +b_my_m & \\
\text{subject to} & a_{1,1}y_1 & +a_{2,1}y_2 & +\cdots & +a_{m,1}y_m & \leq & c_1 \\
& a_{1,2}y_1 & +a_{2,2}y_2 & +\cdots & +a_{m,2}y_m & \leq & c_2 \\
& \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
& a_{1,n}y_1 & +a_{2,n}y_2 & +\cdots & +a_{m,n}y_n & \leq & c_n \\
& & & & y_1,\ldots,x_m & \geq & 0
\end{array}$$

or, using a more concise notation

$$(D) : \max_{\mathbf{y}\in\mathbb{R}^m} \left\{ \mathbf{b}^\mathsf{T}\mathbf{y} \mid A^\mathsf{T}\mathbf{y} \leq \mathbf{c},\ \mathbf{y} \geq \mathbf{0} \right\}.$$

Our musings so far are summarized in the following theorem, known as a *weak duality theorem*:

Theorem 3.1. *If* $\mathbf{x}$ *is an arbitrary feasible solution of the primal program* (P)*, and* $\mathbf{y}$ *is an arbitrary feasible solution of the dual program* (D)*, it holds*

$$\mathbf{c}^\mathsf{T}\mathbf{x} \geq \mathbf{b}^\mathsf{T}\mathbf{y}$$

Proof:

$$\mathbf{c}^\mathsf{T}\mathbf{x} \geq \left(A^\mathsf{T}\mathbf{y}\right)^\mathsf{T}\mathbf{x} = \mathbf{y}^\mathsf{T}A\mathbf{x} \geq \mathbf{y}^\mathsf{T}\mathbf{b} \tag{41}$$

The first inequality follows from the constraints of the dual program: since $\mathbf{y}$ is a feasible solution of the dual program, it holds $\mathbf{c} \geq A^\mathsf{T}\mathbf{y}$. Similarly, the second inequality follows from the constraints of the primal program: since $\mathbf{x}$ is a feasible solution of the primal program, it holds $A\mathbf{x} \geq \mathbf{b}$. □

Theorem 3.1 should come as no surprise to the reader, since it only states how we created the dual program: all feasible solutions of (D) are lower bounds on the minimum of (P). The curious fact in our case was that the values of the optimal solutions of primal and dual program were equal. In the next part we show that this was not a coincidence. Before we do so, however, let us note some observations about the construction of the dual program. In the introductory example, the goal of the primal program was minimization, and we were seeking the best possible lower bound on the minimum, so the goal of the dual program was maximization. However, all our reasoning was symmetrical with respect to this, so we could start from a maximization program, and seek, using

the same method, an upper bound on the maximum; we would get a minimization dual program. The reader easily verifies that the primal and dual programs are completely symmetrical: if we started from the program (D) as primal, we would get a dual program (P). Hence, a dual program of a dual program is the primal program.

Finally, let us observe the form of the constraints. In the introductory example the primal program had constraints in the form of both inequalities ($\geq$), and equality. In the first case, the corresponding multipliers y_i had to be non-negative, in the case of equality, the multiplier could be an arbitrary real number. At the same time, all the variables x_i in the primal program were constrained to be non-negative, so it was sufficient to have the constraints in the dual program in the form of inequalities. What would happen if some variable, say, x_1 could be also negative? In the same way as when we defined the normal form of a linear program we could replace $x_1 = z_1 - z_2$ for two fresh variables $z_1, z_2 \geq 0$, obtaining a primal program minimize

$$f(z_1, z_2, x_2, x_3) := 10z_1 - 10z_2 + 3x_2 + 5x_3$$

over values z_1, z_2, x_2, x_3 satisfying

$$\begin{aligned} 6z_1 - 6z_2 + x_2 - x_3 &\geq 2 \quad (42)\\ 2z_1 - 2z_2 + 2x_2 + 6x_3 &\geq 8 \quad (43)\\ 6z_1 - 6z_2 + 3x_2 + 5x_3 &= 30\\ z_1, z_2, x_2, x_3 &\geq 0 \end{aligned}$$

The corresponding dual program would ask to minimize the function

$$g(y_1, y_2, y_3) := 2y_1 + 8y_2 + 30y_3$$

over values y_1, y_2, y_3 satisfying

$$\begin{aligned} 6y_1 + 2y_2 + 6y_3 &\leq 10\\ -6y_1 - 2y_2 - 6y_3 &\leq -10\\ y_1 + 2y_2 + 3y_3 &\leq 3\\ -y_1 + 6y_2 + 5y_3 &\leq 5\\ y_1, y_2 &\geq 0 \end{aligned}$$

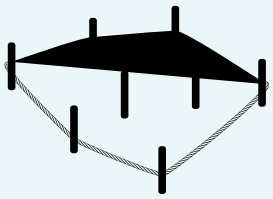
where the first two inequalities can be combined into one equation $6y_1 + 2y_2 + 6y_3 = 10$. Summarizing, the dual program is constructed according to the following rules:

primal program		**dual rpgram**	
minimize	$\mathbf{c}^T\mathbf{x}$	maximize	$\mathbf{b}^T\mathbf{y}$
i-th constraint	$\sum_{j=1}^{n} a_{ij}x_j = b_i$	i-th variable	$y_i \in \mathbb{R}$
i-th constraint	$\sum_{j=1}^{n} a_{ij}x_j \geq b_i$	i-th variable	$y_i \geq 0$
j-th variable	$x_j \in \mathbb{R}$	j-th constraint	$\sum_{i=1}^{m} a_{ij}y_i = c_j$
j-th variable	$x_j \geq 0$	j-th constraint	$\sum_{i=1}^{m} a_{ij}y_i \leq c_j$

Before proceeding towards our goal to show that the common optimum of our primal and dual programs was not a coincidence, let us make a small trip into the land of algebraic topology.

A short visit to convex hulls

The reader probably have already encountered the convex hull in two dimensions: for a given set of points in the plane, the convex hull is the smallest polygon that contains all of them.



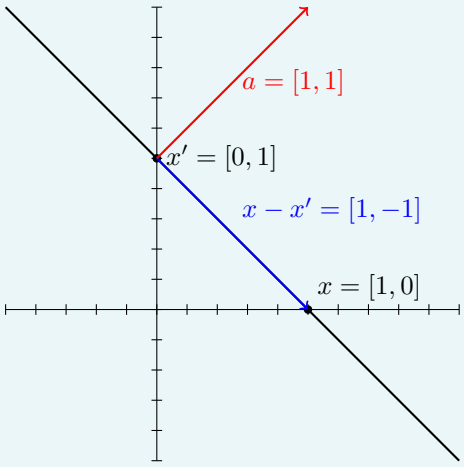
If we imagine the points as poles, if we wrap a rope around them then the poles touched by the rope form the convex hull. This intuition can work quite good also in three dimensions, where the convex hull is obtained by wrapping a set of points by a sheet of paper. Having n-dimensional points and a sheet of $n-1$-dimensional paper is somewhat more complicated to visualize; not all the properties that we take as granted in 2 dimensions hold also in the n-dimensional case. That's why we need to be extra suspicious about arguments like *"it is obvious that..."*, and we need a more formal approach. Recall that a convex body $\mathcal{T}$ is a set of points such that for any two points $\mathbf{x}, \mathbf{y} \in \mathcal{T}$ the whole connecting line is in $\mathcal{T}$, i.e. all the points of the form $t\mathbf{x} + (1-t)\mathbf{y}$ for $0 \leq t \leq 1$. Let us define the convex hull as follows:

Definition 3.2. The convex hull of n points $\mathbf{a}_1, \ldots, \mathbf{a}_n$ is the intersection of all convex bodies that contain the points $\mathbf{a}_1, \ldots, \mathbf{a}_n$.

There are uncountably many convex bodies containing a given set of points, but their intersection is well defined, so we don't care. Moreover, intersection of an arbitrary number of convex bodies is again convex, so our definition is nice in the respect that the convex hull is, as one would expect, convex. The following characterization comes in handy:

Lemma 3.3. *The convex hull of a set of n points $\mathbf{a}_1, \ldots, \mathbf{a}_n$ is formed by the points that are* convex combinations *of the points $\mathbf{a}_1, \ldots, \mathbf{a}_n$, i.e. are of the form $z_1\mathbf{a_1} + \cdots + z_n\mathbf{a}_n$, where $z_1, \ldots, z_n \in \mathbb{R}^+$ a $z_1 + \cdots + z_n = 1$.*

Proof: Denote by K the set of all convex combinations of the points $\mathbf{a_1}, \ldots, \mathbf{a_n}$. It is easy to verify that K is a convex body: for convex combinations $\sum z_i\mathbf{a_i}$ and $\sum z'_i\mathbf{a_i}$ is also $t\sum z_i\mathbf{a_i} + (1-t)\sum z'_i\mathbf{a_i} = \sum(tz_i + (1-t)z'_i)\mathbf{a_i}$ a convex combination, and so belongs to K. Since K is a convex body containing the points $\mathbf{a_1}, \ldots, \mathbf{a_n}$, by definition K contains their convex hull. To prove the lemma it suffices to show that every convex combination is contained in the convex hull.
We prove it by induction on the number of points n. For $n = 1$ it follows directly from the definition of convex hull, for $n = 2$ the convex combinations are $z_1\mathbf{a_1} + (1-z_1)\mathbf{a_2}$ and these, due to convexity, are in any convex body containing $\mathbf{a_1}$ and $\mathbf{a_2}$ (and so they are contained in the intersection of them). Now let $n \geq 3$, and consider a convex combination $\mathbf{x} = \sum z_i a_i$. If $z_n = 1$, $\mathbf{x} = \mathbf{a_n}$ and $\mathbf{x}$ it is in the convex hull by definition. So let $z_n < 1$, and denote $z'_i := \frac{z_i}{1-z_n}$. Considering the point $\mathbf{x'} = \sum z'_i\mathbf{a_i}$, we see it is a convex combination of points $\mathbf{a_1}, \ldots, \mathbf{a_{n-1}}$, and by the induction hypothesis any convex body containing $\mathbf{a_1}, \ldots, \mathbf{a_{n-1}}$ contains also $\mathbf{x'}$. At the same time, $\mathbf{x} = (1-z_n)\mathbf{x'} + z_n\mathbf{a_n}$ implying that any convex body containing both $\mathbf{x'}$ and $\mathbf{a_n}$ contains also $\mathbf{x}$. □



Recall that in an n-dimensional space, a hyperplane $\mathcal{H}$ is a set of points $\mathbf{x}$ satisfying an inequality $a_1x_1+\cdots+a_mx_m = 1$ (the number one on the right hand side is almost without loss of generality: any hyperplane that does not contain the point $[0, \ldots, 0]$ can be normalized to this form). If we use the notation $\langle \cdot, \cdot \rangle$ for the scalar product, $\mathcal{H}$ is defined by $\langle \mathbf{x}, \mathbf{a} \rangle = 1$. The points for which $\langle \mathbf{x}, \mathbf{a} \rangle < 1$ are below the hyperplane, and points $\langle \mathbf{x}, \mathbf{a} \rangle > 1$ are above it. The vector $\mathbf{a}$ is called a *normal* vector of $\mathcal{H}$ and for any two points $\mathbf{x}$, $\mathbf{x'}$ from $\mathcal{H}$ it holds

$$\langle \mathbf{x} - \mathbf{x'}, \mathbf{a} \rangle = \sum_{i=1}^{m} (x_i - x'_i)a_i = \langle \mathbf{x}, \mathbf{a} \rangle - \langle \mathbf{x'}, \mathbf{a} \rangle = 0.$$

In two dimensions the hyperplane is a line; the one on the figure on the left is given by the equation $x_1 + x_2 = 1$.

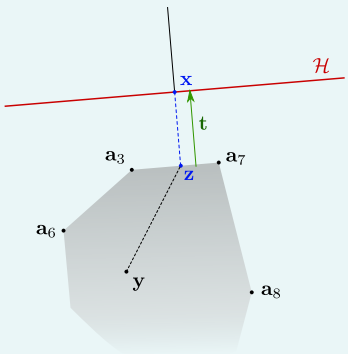
Our discussion about convex hull concludes with the following observation that is completely obvious for two and three dimensions:

Lemma 3.4. *Consider points $\mathbf{x}, \mathbf{a_1}, \ldots, \mathbf{a_n}$ in an m-dimensional space. Let K be the convex hull of $\mathbf{a_1}, \ldots, \mathbf{a_n}$. The it holds:*

- *If $\mathbf{x}$ is inside (i.e. K contains $\mathbf{x}$ with some small neighborhood) K, then there is no hyperplane $\mathcal{H}$ containing $\mathbf{x}$ such that all points from K are on one side (above or below) of $\mathcal{H}$.*
- *If $\mathbf{x} \notin K$, then there exists a hyperplane $\mathcal{H}$ containing $\mathbf{x}$ such that all points from K are on one side (above or below) of $\mathcal{H}$.*

Proof: The first part is easy: let $\mathbf{x}$ be inside K, and suppose there exists a hyperplane $\mathcal{H}$ such that all points from K are one one side of $\mathcal{H}$. Since $\mathcal{H}$ contains $\mathbf{x}$, there are two points in any close neighborhood of $\mathbf{x}$ that lie on different sides of $\mathcal{H}$. But $\mathbf{x}$ is inside K, so there is some small neighborhood of $\mathbf{x}$ contained in K.
Now let us show the second part. Consider a fixed point $\mathbf{x}$, and for all points $\mathbf{y} \in K$ consider the Euclidean distance $f(\mathbf{y}) := ||\mathbf{y} - \mathbf{x}||$. Since K is a compact set, and f is a continuous function with positive codomain, there is a point $\mathbf{z} \in K$ where f reaches minimum. Denote

$$\mathbf{t} = \mathbf{x} - \mathbf{z} \qquad\qquad \kappa = \langle \mathbf{t}, \mathbf{x} \rangle$$



Let$\mathcal{H}$ be the hyperplane consisting of points $\mathbf{y}$ for which $\langle \mathbf{t}, \mathbf{y} \rangle = \kappa$. Clearly $\mathbf{x} \in \mathcal{H}$. It holds

$$0 \leq \sum_{i=1}^{m} (x_i - z_i)^2 = \langle \mathbf{x}, \mathbf{x} \rangle - 2 \langle \mathbf{x}, \mathbf{z} \rangle - \langle \mathbf{z}, \mathbf{z} \rangle$$

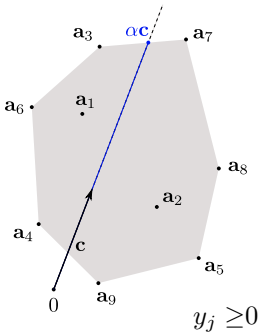
and so

$$\langle \mathbf{t}, \mathbf{z} \rangle = \langle \mathbf{x} - \mathbf{z}, \mathbf{z} \rangle = \langle \mathbf{x}, \mathbf{z} \rangle - \langle \mathbf{z}, \mathbf{z} \rangle \leq$$
$$\leq \langle \mathbf{x}, \mathbf{x} \rangle - \langle \mathbf{x}, \mathbf{z} \rangle = \langle \mathbf{x} - \mathbf{z}, \mathbf{x} \rangle = \langle \mathbf{t}, \mathbf{x} \rangle = \kappa$$

Now it suffices to show that for all points $\mathbf{y} \in K$ it holds $\langle \mathbf{t}, \mathbf{y} \rangle \leq \kappa$, and so they are on the same side of $\mathcal{H}$ as $\mathbf{z}$.

Hence, let us consider $\mathbf{z} \in K$. Since $\mathbf{z}$ minimizes the distance from $\mathbf{x}$, and K is a convex body, for all $0 \leq \lambda \leq 1$ it holds $||\mathbf{z}-\mathbf{x}||^2 \leq ||(1-\lambda)\mathbf{z}+\lambda\mathbf{y}-\mathbf{x}||^2 = ||(1-\lambda)(\mathbf{z}-\mathbf{x})+\lambda(\mathbf{y}-\mathbf{x})||^2$, and from that after straightforward calculation $0 \leq \lambda(\lambda-2)||\mathbf{z}-\mathbf{x}||^2+2\lambda(1-\lambda)\langle \mathbf{z}-\mathbf{x}, \mathbf{y}-\mathbf{x} \rangle+\lambda^2||\mathbf{y}-\mathbf{x}||^2$. Since $\lambda \geq 0$, we have $0 \leq (\lambda-2)||\mathbf{z}-\mathbf{x}||^2 + 2(1-\lambda)\langle \mathbf{z}-\mathbf{x}, \mathbf{y}-\mathbf{x} \rangle + \lambda||\mathbf{y}-\mathbf{x}||^2$ and from that for the limit case $\lambda \mapsto 0$ we get $0 \leq \langle \mathbf{z}-\mathbf{x}, \mathbf{y}-\mathbf{x} \rangle - ||\mathbf{z}-\mathbf{x}||^2 = \langle \mathbf{z}-\mathbf{x}, \mathbf{y}-\mathbf{x} \rangle - \langle \mathbf{z}-\mathbf{x}, \mathbf{z}-\mathbf{x} \rangle = \langle \mathbf{z}-\mathbf{x}, \mathbf{y} \rangle - \langle \mathbf{z}-\mathbf{x}, \mathbf{z} \rangle = \langle \mathbf{t}, \mathbf{z} \rangle - \langle \mathbf{t}, \mathbf{y} \rangle$. That's why $\langle \mathbf{t}, \mathbf{y} \rangle \leq \langle \mathbf{t}, \mathbf{z} \rangle \leq \kappa$, what we wanted to prove. □

Now let us get back to our goal – to show that the same value of the primal and dual program we obtained was not a coincidence. Let us consider the following example: for given n points $\mathbf{a}_1, \ldots, \mathbf{a}_n$ from the m-dimensional space $\mathbb{R}^m$ (i.e. the point $\mathbf{a}_i$ has coordinates $(a_{i1}, a_{i2}, \ldots, a_{im})$), and a vector $\mathbf{c}$, the goal is to find a largest possible number $\alpha \in \mathbb{R}^+$ such that the point $\alpha\mathbf{c}$ is contained in the convex hull K of the points $\mathbf{a}_1, \ldots, \mathbf{a}_n$.



Let us suppose, for the ease of exposition, that there is some solution, i.e. that the ray generated by the vector c hits K. From the convexity it follows that the intersection of K and a line is a line segment. The optimal solution is thus the point $\alpha\mathbf{c}$ where the ray leaves K. Following Lemma 3.3, the points of the convex hull can be expressed as a convex combination of the points $\mathbf{a}_1, \ldots, \mathbf{a}_n$, i.e. our unknown point $\alpha\mathbf{c}$ must be of the form $z_1\mathbf{a}_1 + \cdots + z_n\mathbf{a}_n$ for some $z_1, \ldots, z_n \geq 0$ where $\sum_{i=1}^{n} z_i = 1$. Denoting $y_j = \frac{z_j}{\alpha}$ for an arbitrary feasible point $\alpha\mathbf{c}$ yields

$$\sum_{j=1}^{n} y_j = \frac{1}{\alpha} \qquad \alpha\mathbf{c} = \alpha \sum_{j=1}^{n} \mathbf{a}_j y_j.$$

To find a point that maximizes α means to find a point that minimizes $1/\alpha$. So we want to minimize the value $\sum_{j=1}^{n} y_j$ over those $y_1, \ldots, y_n \geq 0$ for which $\sum_{j=1}^{n} \mathbf{a}_j y_j = \mathbf{c}$; just note that from given $\mathbf{y}$, $\mathbf{z}$ and α can be reconstructed. Our task can be written as a linear program:

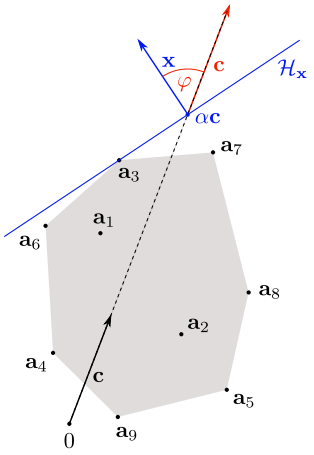
$$\begin{array}{rrcccccc} \text{minimize} & y_1 & + y_2 & + \cdots & + y_n & & \\ \text{subject to} & a_{1,1}y_1 & + a_{2,1}y_2 & + \cdots & + a_{n,1}y_n & = & c_1 \\ & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ & a_{1,m}y_1 & + a_{2,m}y_2 & + \cdots & + a_{n,m}y_n & = & c_m \\ & & & & y_1, \ldots, y_n & \geq & 0 \end{array} \tag{44}$$

or shorter

$$\text{(P)} \quad \min_{\mathbf{y} \in \mathbb{R}^n} \left\{ \mathbf{1}^\mathsf{T}\mathbf{y} \mid A^\mathsf{T}\mathbf{y} = \mathbf{c},\ \mathbf{y} \geq \mathbf{0} \right\},$$

where A is an $n \times n$ matrix with rows formed by the coordinates of points $\mathbf{a}_1, \ldots, \mathbf{a}_n$. We apply our dualization recipe to the program (44) yielding a dual program

$$\text{(D)} \quad \max_{\mathbf{x} \in \mathbb{R}^m} \left\{ \mathbf{c}^\mathsf{T} \mathbf{x} \mid A\mathbf{x} \leq \mathbf{1} \right\} \tag{45}$$



How can we interpret the solution of program (45)? Consider an arbitrary feasible solution $\mathbf{x}$, and let $\mathcal{H}_\mathbf{x}$ be the hyperplane defined by points $\mathbf{y}$ for which $\langle \mathbf{y}, \mathbf{x} \rangle = 1$. The constraints of program (45) tell us that for any point $\mathbf{a_i}$ it holds $\langle \mathbf{a_i}, \mathbf{x} \rangle \leq 1$, and due to convexity the whole convex hull K is below $\mathcal{H}$. Next, if we denote $\alpha = \frac{1}{\mathbf{c}^\mathsf{T}\mathbf{x}}$, $\alpha\mathbf{c}$ is a point of the hyperplane $\mathcal{H}_\mathbf{x}$, since $\langle \alpha\mathbf{c}, \mathbf{x} \rangle = 1$. Hence each feasible solution of (45) corresponds to a point $\alpha\mathbf{c}$ on a ray given by the vector $\mathbf{c}$ that (following Lemma 3.4) is not contained in K. Moreover, it is obvious that the optimum solution is non-negative, so we can without loss of generality assume $\langle \mathbf{x}, \mathbf{c} \rangle = ||\mathbf{x}|| \cdot ||\mathbf{c}|| \cos\varphi \geq 0$, where φ is the angle between the vectors $\mathbf{x}$ and $\mathbf{c}$. Hence we are interested only in the feasible solutions that correspond to the points $\alpha\mathbf{c}$ on the ray generated bu $\mathbf{c}$ after it leaves K.

The same holds also in the opposite direction: Take an arbitrary point $\alpha\mathbf{c}$ from the ray generated by $\mathbf{c}$ after it leaves K. Lemma 3.4 asserts the existence of a hyperplane $\mathcal{H}$ such that all the points from K are below $\mathcal{H}$. Let $\mathcal{H}$ consists of points $\mathbf{y}$ for which $\langle \mathbf{x}, \mathbf{y} \rangle = 1$ for some vector $\mathbf{x}$, then it holds $A\mathbf{x} \leq 1$. At the same time, since $\mathcal{H}$ contains $\alpha\mathbf{c}$, it holds $\langle \mathbf{x}, \alpha\mathbf{c} \rangle = 1 = \alpha \langle \mathbf{x}, \mathbf{c} \rangle$, and so $\alpha = \frac{1}{\mathbf{c}^\mathsf{T}\mathbf{x}}$. For the maximum value $\mathbf{c}^\mathsf{T}\mathbf{x}$ the corresponding α is minimal, so the program (45) actually asks to find a minimum α such that the point $\alpha\mathbf{c}$ is after K on the ray generated by $\mathbf{c}$. This is, however, obviously the point where the ray leaves K, and so the programs (44) and (45) have the same optimal solution. We proved the statement

Lemma 3.5. *If the primal program* $\min_{\mathbf{y} \in \mathbb{R}^n} \left\{ \mathbf{1}^\mathsf{T}\mathbf{y} \mid A^\mathsf{T}\mathbf{y} = \mathbf{c},\ \mathbf{y} \geq \mathbf{0} \right\}$ *has feasible solution, then also the dual program* $\max_{\mathbf{x} \in \mathbb{R}^m} \left\{ \mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \leq \mathbf{1} \right\}$ *has feasible solution, and the values of the optima in both programs are equal.*

If we managed to generalize the Lemma 3.5 to the programs of the form $\min_{\mathbf{y} \in \mathbb{R}^n} \left\{ \mathbf{b}^\mathsf{T}\mathbf{y} \mid A^\mathsf{T}\mathbf{y} = \mathbf{c},\ \mathbf{y} \geq \mathbf{0} \right\}$ for an arbitrary vector $\mathbf{b}$, we would be happy, since any linear program can be expressed in that form. So, let us consider a linear program

$$\text{(P)} \quad \min_{\mathbf{y} \in \mathbb{R}^n} \left\{ \mathbf{b}^\mathsf{T}\mathbf{y} \mid A^\mathsf{T}\mathbf{y} = \mathbf{c},\ \mathbf{y} \geq \mathbf{0} \right\} \tag{46}$$

and the corresponding dual program

$$\text{(D)} \quad \max_{\mathbf{x} \in \mathbb{R}^m} \left\{ \mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b} \right\} \tag{47}$$

First, let us consider all vectors $\mathbf{b}$ with all entries $b_i > 0$. Denote

$$\mathbf{a}'_i = \frac{1}{b_i}\mathbf{a}_i \qquad\qquad y'_j = b_j y_j \tag{48}$$

It holds $\mathbf{b}^\mathsf{T}\mathbf{y} = \sum_{j=1}^n b_j y_j = \mathbf{1}^\mathsf{T}\mathbf{y}'$ and for each $i \in \{1, \ldots, m\}$ we have $\sum_{j=1}^m a_{ji} y_j = \sum_{j=1}^m a'_{ji} b_i \frac{y'_j}{b_j}$. Since $b_j > 0$, the program (46) is equivalent[1] with the program

$$\min_{\mathbf{y'} \in \mathbb{R}^n} \left\{ \mathbf{1}^\mathsf{T}\mathbf{y'} \mid A'\mathbf{y'} = \mathbf{c},\ \mathbf{y'} \geq \mathbf{0} \right\} \tag{49}$$

[1] in the sense that to every feasible solution of the program (46) there is some corresponding feasible solution of (49) with the same value, and vice versa

Where the columns of the matrix A' are the vectors $\mathbf{a}'_i$. Following Lemma 3.5, if the program (49) has a feasible solution, its optimum is the same as the optimum of the program

$$\max_{\mathbf{x'}\in\mathbb{R}^m} \left\{ \mathbf{c}^\mathsf{T}\mathbf{x'} \mid {A'}^\mathsf{T}\mathbf{x'} \leq \mathbf{1} \right\} \tag{50}$$

The constraints of the program (50 are of the form $\sum_{j=1}^m a'_{ji}x'_j \leq 1$ and so by exploiting the notation (48) we get that programs (50) and (47) are equivalent, so if (46) has a feasible solution, (47) has a feasible solution, too, and values of the optima are the same.

Exercise. Modify the previous approach so that it works also for $b_i \geq 0$.

Now let $\mathbf{b}$ have arbitrary values $b_i \in \mathbb{R}$, and let $\bar{\mathbf{x}}$ be a fesible solution of (47). Denote

$$\mathbf{x'} = \mathbf{x} - \bar{\mathbf{x}} \qquad\qquad b'_i = b_i - \mathbf{a}_i^\mathsf{T}\bar{\mathbf{x}} \tag{51}$$

How will the programs (46) and (47) look like in this notation? For any feasible solutions $\mathbf{x}$, $\mathbf{y}$ we have

$$\begin{array}{ll} \mathbf{c}^\mathsf{T}\mathbf{x} & = \mathbf{c}^\mathsf{T}\mathbf{x'} + \mathbf{c}^\mathsf{T}\bar{\mathbf{x}} \\ \mathbf{b}^\mathsf{T}\mathbf{y} & = \sum_{i=1}^n b_i y_i = \sum_{i=1}^n b'_i y_i + \sum_{i=1}^n y_i(\mathbf{a}_i^\mathsf{T}\bar{\mathbf{x}}) = \mathbf{b'}^\mathsf{T}\mathbf{y} + \sum_{i=1}^n y_i \sum_{j=1}^m a_{i,j}\tilde{x}_j = \\ & = \mathbf{b'}^\mathsf{T}\mathbf{y} + \sum_{j=1}^m \tilde{x}_j(\sum_{i=1}^n y_i a_{i,j}) = \mathbf{b'}^\mathsf{T}\mathbf{y} + \mathbf{c}^\mathsf{T}\bar{\mathbf{x}} \end{array}$$

where the last equality follows from $A^\mathsf{T}\mathbf{y} = \mathbf{c}$. Since $\mathbf{c}^\mathsf{T}\bar{\mathbf{x}}$ is constant, programs (46) and (47) can be equivalently written

$$(\mathrm{P'}) \quad \min_{\mathbf{y}\in\mathbb{R}^n} \left\{ \mathbf{b'}^\mathsf{T}\mathbf{y} \mid A^\mathsf{T}\mathbf{y} = \mathbf{c},\ \mathbf{y} \geq \mathbf{0} \right\} \qquad\qquad (\mathrm{D'}) \quad \max_{\mathbf{x'}\in\mathbb{R}^m} \left\{ \mathbf{c}^\mathsf{T}\mathbf{x'} \mid A\mathbf{x'} \leq \mathbf{b'} \right\}$$

Moreover, from the feasibility of $\bar{\mathbf{x}}$ for (47 it follows that $\mathbf{b'} \geq \mathbf{0}$, and so the previous case applies. Hence, we have sketched[2] the proof of a fundamental theorem in the theory of linear programming:

Theorem 3.6 (Strong duality theorem)**.** *For a pair of linear programs*

$$(P) \quad \min_{\mathbf{y}\in\mathbb{R}^n} \left\{ \mathbf{b}^\mathsf{T}\mathbf{y} \mid A^\mathsf{T}\mathbf{y} = \mathbf{c},\ \mathbf{y} \geq \mathbf{0} \right\} \qquad\qquad (D) \quad \max_{\mathbf{x}\in\mathbb{R}^m} \left\{ \mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b} \right\}$$

exactly on of the following cases holds:

1. *neither (P) nor (D) has any feasible solutions*
2. *(P) is unbounded, and (D) has no feasible solutions*
3. *(D) is unbounded, and (P) has no feasible solutions*
4. *both (P) and (D) have feasible solutions, and the optimum values of (P) and (D) are equal.*

[2]For a complete proof some more special cases have to be covered, a boring work that we gladly leave up to the enthusiastic reader.

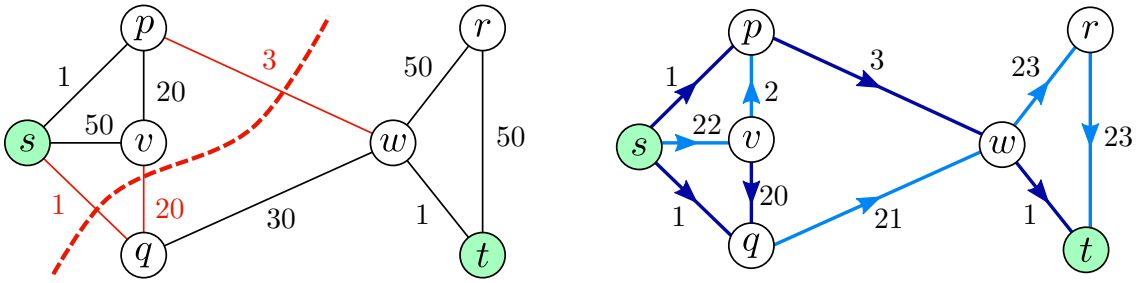
3.2 Max-Flow–Min-Cut in the view of LP duality

The LP duality, introduced in the previous section, often helps to better (or at least differently) understand min-max characterizations of computational problems. In this part we employ the duality theory to give an alternative proof of the known fact that the size of the maximum $s-t$ flow equals to the capacity of the minimum $s-t$ cut. Since we suppose that the reader is already familiar with the Max-Flow and Min-Cut problems, we abstain from motivational examples, and just recall the definitions.

Given is an (undirected) graph $G = (V, E)$ with non-negative edge weights, i.e. a mapping $c : E \mapsto \mathbb{R}^+$; instead of $c(\{u, v\})$ we shall use a shorthand notation $c_{uv} = c_{vu}$. In the graph, there are two distinguished vertices a and t, The Max-Flow problem aims to maximize the flow from s to t, where a flow is a mapping $f : V^2 \mapsto \mathbb{R}^+$, such that $f(u, v)$ represents the amount of a liquid that flows from u to v in an unit time. The flow must hence satisfy the following properties:

1. if $f(u, v) \neq 0$ then $(u, v) \in E$, i.e. the flow can be only along the edges of the graph
2. $f(u, v) = -f(v, u)$, i.e. the sign gives the direction of the flow
3. for each $v \notin \{s, t\}$ it holds $\sum_{u \in V} f(u, v) = 0$, known as a flow conservation law
4. $|f(u, v)| \leq c_{uv}$, i.e. the flow is bounded by the capacity of the edge

The size of the flow is $\sum_{v \in V} f(s, v)$. A cut in G is a set of edges in G whose removal disconnects s, and t (i.e. they are in different connected components of the resulting graph). The Min-Cut problems asks to find a cut with minimal capacity. The following figure depicts a graph with a minimum cut (left), and a maximum flow (right; the dark edges are those whose capacity is fully used by the flow) with value 24.



The Max-Flow problem has a number of natural formulations as a linear program. For reasons we hope to become apparent by the end of this section we choose the following one: for each edge $(u, v) \in E$ we introduce two non-negative variables x_{uv} and x_{vu} that denote the amount of flow along the edge (u, v) (note that, by doing this, we allow to have a non-negative amount of flow x_{uv} from u to v, and at the same time, some non-negative amount of flow x_{vu} from v to u; however, it is easy to verify that simply subtracting the two values maintains the feasibility and value of a solution, and makes it compatible with our definition of flow). Next, we introduce a variable f that will denote the amount of the whole flow (and the goal will be to maximize f): $f = \sum_{u:(s,u)\in E} x_{su} - \sum_{u:(s,u)\in E} x_{us}$. Flow conservation law, and the capacity constraints can easily be expressed as linear inequalities, yielding the following program:

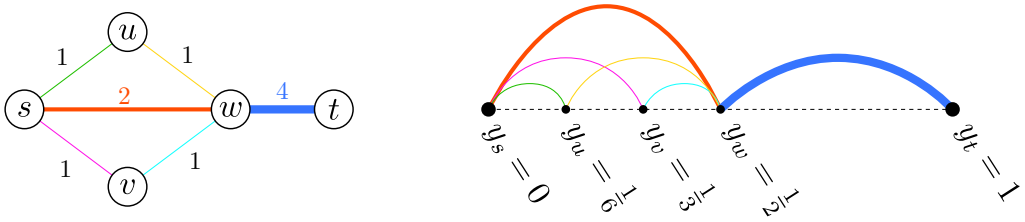
$$
\begin{array}{lll}
\text{maximize} & f & \\
\text{subject to} & \sum_{u:(s,u)\in E} x_{su} - \sum_{u:(s,u)\in E} x_{us} - f = 0 & \\
& \sum_{u:(t,u)\in E} x_{ut} - \sum_{u:(s,u)\in E} x_{tu} + f = 0 & \\
& \sum_{u:(u,v)\in E} x_{vu} - \sum_{u:(u,v)\in E} x_{uv} = 0 & \forall v \in V - \{s,t\} \\
& x_{uv} \leq c_{uv} & \forall (u,v) \in E \\
& x_{vu} \leq c_{uv} & \forall (u,v) \in E \\
& x_{uv} \geq 0 & \forall (u,v) \in E
\end{array}
\tag{52}
$$

The first two constraints define the size of the flow f as the net amount that leaves s, or enters t, respectively. All remaining vertices (the next set of constraints) obey the flow conservation law.
Now let us apply our dualization recipe, and look at the dual minimization program to the program (52) Every constraint in the primal program gives rise to a variable in the dual program, In the program (52) there are two types of constraints: the constraints of the form $\cdots = 0$ for each vertex, and constraints $x_{uv} \leq c_{uv}$. Introduce a dual variable $y_v \in \mathbb{R}$ for each vertex $v \in V$ (y_s corresponds to the first constraint, y_t to the second one, and the rest to the remaining vertices), and a pair of non-negative variables z_{uv} a z_{vu} for each edge $(u,e) \in E$. The i-th dual variable is a multiplier of the i-th constraint in the linear combination with the requirement that sum of the left-hand sides is component-wise bigger than the maximized utility function. The sum of the right-hand sides is in our case $\sum_{(u,v)\in E} (z_{uv} + z_{vu}) c_{uv}$. The maximized function is zero in all components but f.
The variable f appears only in the first and second constraints of (52), hence we get in the dual a constraint $y_t - y_s = 1$ (note that although in the optimum solution f clearly is non-negative, we did not require the variable $f \geq 0$). Every variable x_{uv} appears in three constraints: the constrain corresponding to vertex the u with positive sign, the constraint corresponding to the vertex v with negative sign, and in the constraint $x_{uv} \leq c_{uv}$. Summarizing, we get the following program:

$$
\begin{array}{lll}
\text{minimize} & \sum_{(u,v)\in E} (z_{uv} + z_{vu})\, c_{uv} & \\
\text{subject to} & y_t - y_s = 1 & \\
& y_v - y_u + z_{uv} \geq 0 & \forall (u,v) \in E \\
& y_u - y_v + z_{vu} \geq 0 & \forall (u,v) \in E \\
& z_{uv}, z_{vu} \geq 0 & \forall (u,v) \in E
\end{array}
\tag{53}
$$

From the strong duality theorem we know that programs (52) and (53) have the same value of the optimum. How can we interpret the program (53)? Without loss of generality we can suppose that at least one of the pair of variables z_{uv}, z_{vu} is zero: the only constraints concerning z_{uv} or z_{vu} are the constraints $z_{uv} \geq y_u - y_v$ and $z_{vu} \geq y_v - y_u$. Clearly, at least one of the values $y_u - y_v$ and $y_v - y_u$ is not positive, and so we can set at least one of them to zero in any feasible solution, and not increase the value of the utility function. Hence, we can denote $\bar{z}_{uv} = \max\{z_{uv}, z_{vu}\}$; the constraints then ensure that $\bar{z}_{uv} \geq |y_u - y_v|$ and using the same arguments as before we can suppose without loss of generality $\bar{z}_{uv} = |y_u - y_v|$. Now we can see the solving of the program as placing the vertices of the graph on a line: the vertex v is placed in the point y_v, where again without loss of generality we may assume that s is placed in 0, and t in 1. The value $\bar{z}_{uv}$ is the length of the edge (u,v) in this placement of vertices. Altogether, the optimal solution of the

program (53) places the vertices of the graph on a line segment of length 1 in such a way that s and t are on the endpoints, and the overall length of edges, weighted by their capacity, is minimized.

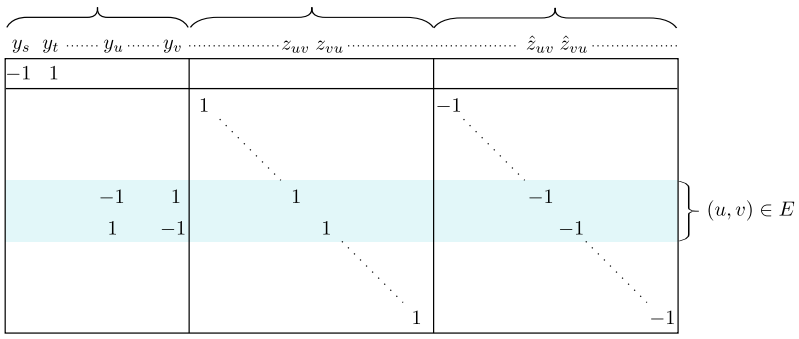


A graph with edge weights (left) and one of possible optimal solutions (right) of the program (53): $\bar{z}_{su} = \bar{z}_{vw} = \frac{1}{6}$, $\bar{z}_{sv} = \bar{z}_{uw} = \frac{1}{3}$, $\bar{z}_{sw} = \bar{z}_{wt} = \frac{1}{2}$. The resulting value is

$$\sum_{(u,v)\in E} c_{uv}\bar{z}_{uv} = \frac{1}{6} + \frac{1}{3} + 2\frac{1}{2} + \frac{1}{3} + \frac{1}{6} + 4\frac{1}{2} = 4.$$

Now let us rewrite the program (53) in normal form $\max\{-\mathbf{c}^{\mathsf{T}}\boldsymbol{\beta} \mid A\boldsymbol{\beta} = 0,\ \boldsymbol{\beta} \geq 0\}$. Introduce a slackness variable $\hat{z}_{uv} \geq 0$ to each variable z_{uv}, obtaining the constraints of the form $y_v - y_u + z_{uv} - \hat{z}_{uv} = 0$. If the vector $\boldsymbol{\beta}$ contains the values ordered as $y_s, y_t, y_{v_1}, \ldots, z_{uv}, z_{vu}, \ldots, \hat{z}_{uv}.\hat{z}_{vu}, \ldots$, the matrix A has the following structure:

$A =$



The following straightforward chores are left to the reader:

Exercise. Using Theorems 2.6 and 2.8 prove that the matrix A is TUM.

Theorem 2.5 implies that there exists an integral optimal solution of the program (53). It means that all vertices have $y_v \in \{0, 1\}$, i.e. they are placed in one of the endpoints of the line, and each edge has length 0 or 1. That means that every $s - t$ path must contain at least one edge with length 1, which implies that the removal of edges with length 1 disconnects s and t. So we can say

Theorem 3.7 (Max-Flow-Min-Cut theorem)**.** *The size of the maximal flow is equal to the capacity of the minimal cut.*

Proof: The size of the maximal flow is the optimum value of the program (52), which is the same as the optimum of the program (53). There exists an integral optimum of (53) and at the same time there is a bijection between $s - t$ cuts and integral solutions of (53). □

It is quite possible that the reader is familiar with a proof that is significantly simpler, and does not involve the linear programming. The reason why we present this argument (apart from the fact that we want to give a more concrete example of the LP duality) is that the Max-Flow-Min-Cut pair is seen as a special case of a general class of problems; at the same time, it sheds a new light into the question why a similar result does not hold for seemingly similar problems (e.g. we would have more sources and sinks, and the goal would be to maximize the flow of several

commodities in common networks of pipes, we would get a problem that is dual to the relaxed version of the MIN-MULTI-CUT from Definition 2.14, however, a similar result does not hold) via the unimodularity of the corresponding matrices.

3.3 The primal-dual method

We hope to have persuaded the reader that duality is an interesting property of linear programs. Now it is time to show that it is also a useful one. Let us consider the primal-dual pair of linear programs

$$(P): \quad \min_{\mathbf{x}\in\mathbb{R}^n} \{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x} \geq \mathbf{b}, \mathbf{x} \geq 0\}$$
$$(D): \quad \max_{\mathbf{y}\in\mathbb{R}^m} \{\mathbf{b}^\mathsf{T}\mathbf{y} \mid A^\mathsf{T}\mathbf{y} \leq \mathbf{c}, \mathbf{y} \geq 0\}$$

From the strong duality theorem we know that they have the same value of the optimum, i.e. there exist vectors $\mathbf{x}^\star \geq 0$ and $\mathbf{y}^\star \geq 0$, such that $A\mathbf{x}^\star \geq \mathbf{b}$, $A^\mathsf{T}\mathbf{y}^\star \leq \mathbf{c}$ and $\mathbf{c}^\mathsf{T}\mathbf{x}^\star = \mathbf{b}^\mathsf{T}\mathbf{y}^\star$. Recall the inequalities from the proof of Theorem3.1 that hold for all pairs of feasible solutions of a primal-dual pair, and so in particular for $\mathbf{x}^\star$, $\mathbf{y}^\star$:

$$\mathbf{c}^\mathsf{T}\mathbf{x}^\star \overset{(\clubsuit)}{\geq} \left(A^\mathsf{T}\mathbf{y}^\star\right)^\mathsf{T}\mathbf{x}^\star = \mathbf{y}^{\star\mathsf{T}}A\mathbf{x}^\star \overset{(\diamondsuit)}{\geq} \mathbf{y}^{\star\mathsf{T}}\mathbf{b} \tag{54}$$

Since $\mathbf{c}^\mathsf{T}\mathbf{x}^\star = \mathbf{b}^\mathsf{T}\mathbf{y}^\star$, both ($\clubsuit$) and ($\diamondsuit$) must be equal. Let us have a look at the ($\clubsuit$), and expand the scalar product of vectors into a sum:

$$\sum_{j=1}^{n} c_j x_j^\star = \sum_{j=1}^{n} \left[A^\mathsf{T}\mathbf{y}^\star\right]_j x_j^\star, \tag{55}$$

where the symbol $[\cdot]_j$ denotes the j-th coordinate of a vector. From the feasibility of the dual solution we know that $\mathbf{c} \geq A^\mathsf{T}\mathbf{y}^\star$; the inequality of the vectors holds in every coordinate, so for each j it is $c_j \geq \left[A^\mathsf{T}\mathbf{y}^\star\right]_j$. For each j there is $x_j^\star \geq 0$, and so also $c_j x_j^\star \geq \left[A^\mathsf{T}\mathbf{y}^\star\right]_j x_j^\star$, which implies that in order the equality (55) to hold, the equality must hold separately in every coordinate.
The same reasoning goes for the inequality ($\diamondsuit$), yielding the following characterization of the optimum solutions:

Theorem 3.8 (slackness conditions)**.** *Let* $\mathbf{x}$*,* $\mathbf{y}$ *be feasible solutions of the primal and dual programs, respectively. Then* $\mathbf{x}$*,* $\mathbf{y}$ *correspond to the common optimal solution if and only if the following conditions hold:*

- *primal slackness conditions:*

$$\forall\, 1 \leq j \leq n: \quad \text{either} \;\; x_j = 0 \;\; \text{or} \;\; \sum_{i=1}^{m} a_{ij} y_i = c_j$$

- *dual slackness conditions:*

$$\forall\, 1 \leq i \leq m: \quad \text{either} \;\; y_i = 0 \;\; \text{or} \;\; \sum_{j=1}^{n} a_{ij} x_j = b_i$$

Edmonds' algorithm for MIN-1-FACTOR

When we spoke about the simplex method of solving linear programs, we argued that it is an efficient, albeit not polynomial-time, algorithm. Also, we mentioned that there exist polynomial

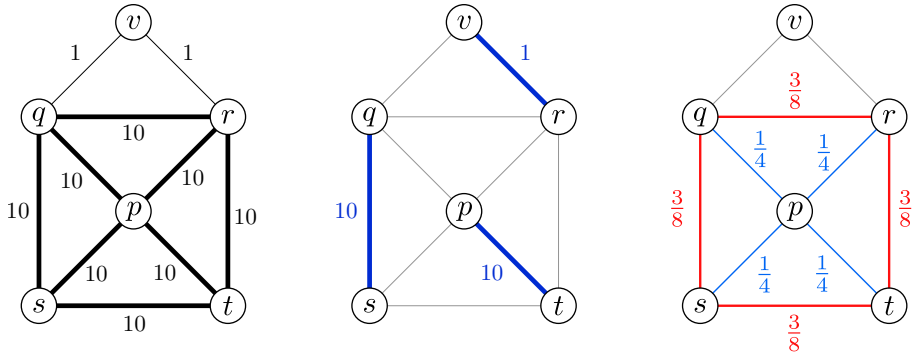
methods for solving LP. The truth is that, polynomial or not, solving LP can be quite expensive, especially if the program is large. The slackness conditions present a tool to significantly reduce the complexity of LP-based algorithms: since we have an equivalent characterization of the optimal solution, we don't have to solve the LP. Instead, if we construct, using an arbitrary algorithm, solutions of the primal and dual programs that satisfy the slackness conditions, we know that we have optimum solution. The usual approach in the primal-dual method is to maintain a pair of solutions of the primal-dual pair of programs, such that we start with a good (but not feasible) primal, and a feasible (but far from optimum) dual solutions. In a sequence of iterations, we improve the value of the dual solution while maintaining feasibility, and at the same time adjust the primal solution so that it is "closer" to being feasible. In the end, we have a feasible primal-dual pair, with hopefully the same (optimal) value. We shall illustrate this approach on an example. Recall the problem Max-Weighted-Bipartite-Matching:

Definition 2.3. Given a bipartite graph with edges labeled by non-negative weights, the Max-Weighted-Bipartite-Matching problem asks to select a set of edges with maximal overall weight in such a way that no two selected edges share a common vertex.

In section 2.1 we formulated the problem as an ILP:

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} \omega_e x_e \\ \text{subject to} \quad & \sum_{\substack{e \in E \\ e=(v,w)}} x_e \leq 1 \qquad \forall v \in V \\ & x_e \geq 0 \qquad \forall e \in E \\ & x_e \in \mathbb{Z} \end{aligned}$$

where $\boldsymbol{\omega} \in \mathbb{R}^n$ is the vector of edge-weights. We showed that the matrix of constraints is TUM, and thus it is sufficient to solve the relaxed program, and we have the integrality of the optimum solutions for free. Quite naturally, we can ask: What happens if the graph is not bipartite? The formulation of the ILP is still valid (we haven't used the fact that the graph was bipartite), but it is not true anymore that the optimum solution is always integral.



On the left there is a graph with its edge-weights where the maximum matching has value 21: edges of weight 10 are only among vertices $\{p, q, r, s, t\}$, so in any solution there can be at most two of them. From the remaining two edges, any solution can contain only one. On the right there is a solution of the relaxed program with value 25.

Before we continue we reformulate our problem slightly. A matching that covers all vertices, i.e. the set of edges $E' \subseteq E$ such that every vertex is incident with exactly one edge from E' will be called a *perfect matching* or a *1-factor*. Obviously, in order a graph could have a 1-factor, it has to have even number of vertices. Instead of finding the heaviest matching, it is sufficient to be able to find the heaviest 1-factor: we first adjust the input graph G such that if it has odd number of vertices we add a new isolated vertex, and then we connect all pairs of vertices that are not connected by an existing edge, by new edges with weight 0. It is easy to check that matchings in

G correspond to 1-factors in G', and vice-versa. Next, if $\omega_{\max}$ is maximum weight of an edge in G'. If we replace the weights of all edges from ω_e to $\omega_{\max} - \omega_e$, we get the following definition:

Definition 3.10. Given is a complete graph $G = (V, E)$ with even number of vertices and non-negative edge weights $\omega_e \in \mathbb{R}^+$. The problem MIN-1-FACTOR is to find a 1-factor of G that minimizes the sum of weights, i.e. with weight $\min_{E'} \sum_{e \in E'} \omega(e)$, where the minimum is taken over all 1-factors of E' of G.

The reader who was patient enough to bear with us up to now can easily express MIN-1-FACTOR as an ILP: for each edge $e \in E$ we introduce a variable $x_e \in \{0, 1\}$ denoting whether e is selected in the matching or not. The selected set of edges is a 1-factor exactly if there is exactly one selected edge incident to any vertex; this can be expressed in a straightforward way

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} \omega_e x_e \\ \text{subject to} \quad & \sum_{\substack{e \in E \\ e=(u,v)}} x_e = 1 \qquad \forall v \in V \\ & x_e \in \{0,1\} \qquad \forall e \in E \end{aligned} \tag{56}$$

If we relax this program by considering $x_e \geq 0$ instead of $x_e \in \{0, 1\}$ (note that $x_e \leq 1$ is implied by the minimization), the optimum is not necessarily integral. We shall now present the primal-dual approach according to Edmonds [**?**]. Before that, however, we introduce one more notation that simplifies the subsequent text:

Definition 3.11 (edge boundary of a set of vertices)**.** Given a graph $G = (V, E)$, and a set of vertices $S \subseteq V$, the edge boundary of the set S, denoted $\partial(S)$, is the set of edges with one endpoint in S, and the other outside of S, i.e.

$$\partial(S) := \{e \in E \mid e = (u, v),\ u \in S,\ v \in V \setminus S\}$$

How should we cope with the fact that the program (56) has no integral optimum? The main idea is to augment the program with many additional constraints that have no impact on the ILP, but they force the relaxed program to have integral optimum. Let $\mathscr{S}$be the family of all sets of vertices with odd size, containing at least three vertices, i.e.

$$\mathscr{S} := \{S \subseteq V \mid\ |S| > 1,\ |S| \text{ odd }\}$$

Every edge has two endpoints, so the vertices from any $S \in \mathscr{S}$ cannot be paired among themselves, meaning that in any 1-factor there must be at least one edge outgoing from S. We add these constraints explicitly to the relaxed program (56), obtaining

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} \omega_e x_e \\ \text{subject to} \quad & \sum_{e \in \partial(\{v\})} x_e = 1 \qquad \forall v \in V \\ & \sum_{e \in \partial(S)} x_e \geq 1 \qquad \forall S \in \mathscr{S} \\ & x_e \geq 0 \qquad \forall e \in E \end{aligned} \tag{57}$$

…and *voilà!* We have a linear program with integral optimum solution. However, now we not only have to show that indeed the optimum solution of the program (57) is integral, but we have to cope with a program that has exponentially many constraints. Neither of these problems is unsurmountable, as there are methods that allow us to solve, under some assumptions that are satisfied in this case, in polynomial time even linear programs with exponentially many (in fact, even with infinitely many) constraints. However, we have a more clever solution in mind: We avoid solving (57) by using duality, and the integrality proof comes for free.

Let us now use our dualization recipe and write down the dual program to (57). It is a maximization program with a variable for each constraint of the original program. There are two types of constraints in the original program: the ones concerning vertices and the ones concerning sets, so let us introduce two sets of variables: $r_v \in R$ for $v \in V$ and $w_S \in \mathbb{R}^+$ for $S \in \mathscr{S}$. Every primal variable x_e contributes $x_e\omega_e$ to the utility function, and appears in two vertex-constraints (the ones for the endpoints of e) and in the set-constraints for the sets $S \in \mathscr{S}$ where $e \in \partial(S)$. We obtain the following program (note that r_v may be also negative):

$$\begin{aligned} \text{maximize} \quad & \sum_{v\in V} r_v + \sum_{S\in\mathscr{S}} w_S \\ \text{subject to} \quad & r_u + r_v + \sum_{\substack{S\in\mathscr{S}\\ e\in\partial(S)}} w_S \;\leq\; \omega_e \qquad \forall e=(u,v)\in E \\ & w_S \;\geq\; 0 \qquad \forall S\in\mathscr{S} \end{aligned} \tag{58}$$

For the explanatory reasons, and also because in our pair of programs not all variables are non-negative, let us recall the inequality (54):

$$\sum_{e\in E} x_e\omega_e \overset{\clubsuit}{\geq} \sum_{\substack{e\in E\\ e=(u,v)}} x_e(r_u+r_v+\sum_{\substack{S\in\mathscr{S}\\ e\in\partial(S)}} w_S) \overset{\heartsuit}{=} \sum_{v\in V}(r_v \sum_{e\in\partial(\{v\})} x_e)+\sum_{S\in\mathscr{S}} w_S \left(\sum_{e\in\partial(S)} x_e\right) \overset{\diamondsuit}{\geq} \sum_{v\in V} r_v+\sum_{S\in\mathscr{S}} w_S$$

The equality in ($\heartsuit$) comes due to the fact that every vertex $v \in V$ contributes by $r_v x_e$ to all edges e incident with v, and every set $S \in \mathscr{S}$ contributes by $w_s x_e$ to all edges from the edge boundary of S. The constraints of the program (57) imply that the blue sum $\sum_{e\in\partial(\{v\})} x_e = 1$ and the red sum $\sum_{e\in\partial(S)} x_e \geq 1$; hence, the slackness conditions are of the form

$$\textbf{S1}(\clubsuit) \quad \forall e=(u,v)\in E: \quad x_e > 0 \Rightarrow r_u + r_v + \sum_{\substack{S\in\mathscr{S}\\ e\in\partial(S)}} w_S = \omega(e)$$

$$\textbf{S2}(\diamondsuit) \qquad \forall S\in\mathscr{S}: \quad w_S > 0 \Rightarrow \sum_{e\in\partial(S)} x_e = 1$$

Now have a look at the program (58) and try to find some intuitive interpretation. Imagine that there is a bubble around every vertex $v \in V$, and every set $S \in \mathscr{S}$, carrying a charge of r_v, and w_S, respectively. Clearly, the sets $S \in \mathscr{S}$ may overlap, making the image of bubbles sort of weird, however, in the final algorithm we end up using only selections of non-overlapping bubbles. The program (58) aims at maximizing the overall charge. The edge weights can be considered a capacity, and the constraints require that no edge e is overcharged: the overall charge on all bubbles that e crosses cannot be more than its capacity. An edge e for which $r_u + r_v + \sum_{\substack{S\in\mathscr{S}\\ e\in\partial(S)}} w_S = \omega(e)$ will be called *full*. The conditions **S1** and **S2** can, together with the strong duality theorem, provide the following observation

Lemma 3.12. *If we find a 1-factor M, and the assignment of the charge to the bubbles* $\mathbf{r}$ *and* $\mathbf{w}$ *in such a way that*

(I1) *no edge is overcharged,*

(I2) *all $w_S \geq 0$,*

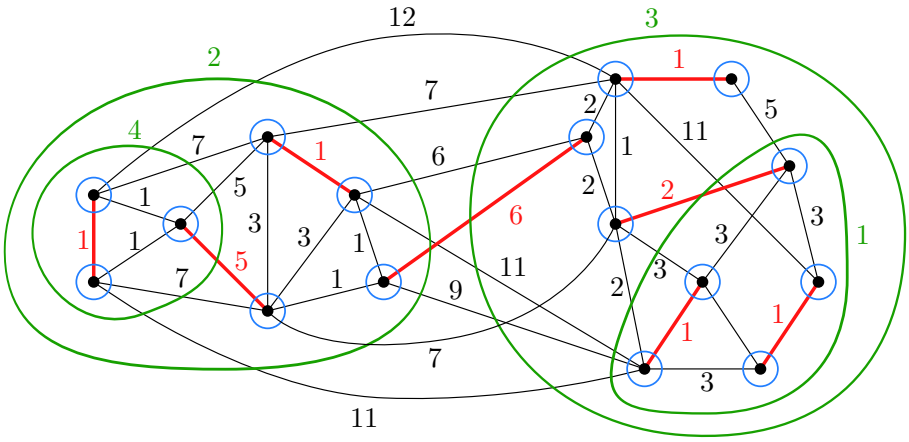
(I3) *all edges from M are full, and*

(I4) *from each non-zero bubble $S \in \mathscr{S}$ there is exactly one outgoing edge from M,*

then we have an optimal solution of the primal-dual pair of programs (57) and (58, with an integral value of $\mathbf{x}$*, and thus a minimum 1-factor.*

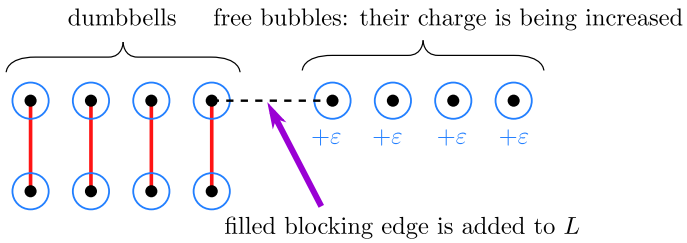
Proof: The conditions **(I1)** and **(I2)** guarantee the feasibility of the dual program (58). The fact that M is a 1-factor guarantees the feasibility of the primal program (57). The conditions **(I3)** and **(I4)** are, respective, the slackness conditions **S1** and **S2**, so the statement is a corollary of Theorem 3.8. □

From this moment on we can forget about the whole linear programming business, and concentrate on finding an algorithm that constructs the desired objects M, $\mathbf{r}$, and $\mathbf{w}$. As we already hinted, since the whole vector $\mathbf{w}$ is too big, we shall explicitly store only the non-zero entries; at the same time, we shall be looking only for solutions where the sets with non-zero bubbles do not intersect each other. We are seeking the following structure:

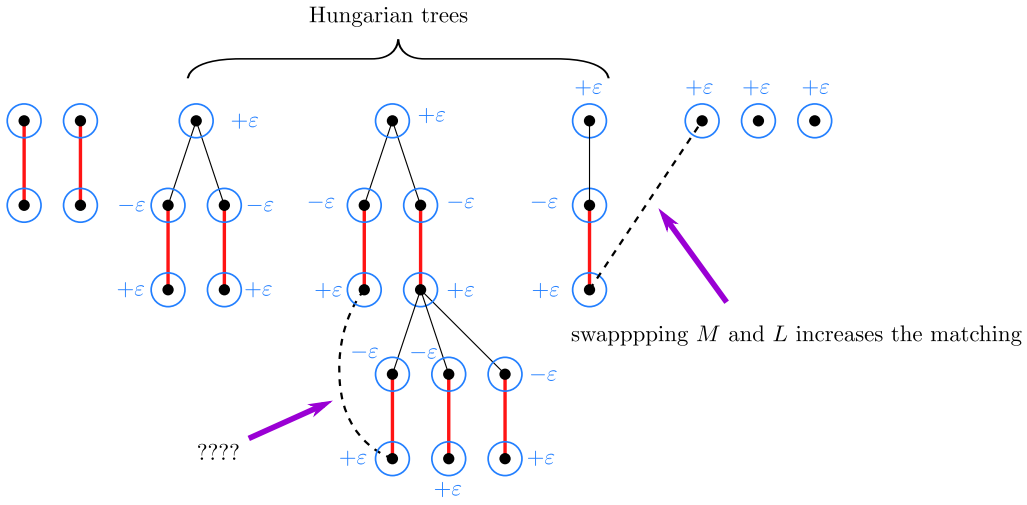


The blue bubbles have all value 1/2, the edges that are not drawn have weight ∞. The red edges form a 1-factor. No edge is overcharged, all red edges are full, and there is exactly one outgoing red edge from each green bubble. Hence, we know that the red 1-factor is minimum.

Of course, so far we have no guarantee that such a configuration always exists. However, if we prove it, and find an algorithm that constructs it, we can check the task as finished and celebrate. Let us begin with an informal description of the algorithm. During the whole computation, the conditions **(I1)**, **(I2)** and **(I3)** will be maintained as invariants. The algorithm starts with an empty matching M and all bubbles zero. Gradually, the charge of some bubbles will be increased, and some edges will be added to the matching so that, in the end, M is 1-factor, and the condition **(I4)** holds; then M is the minimum 1-factor. In the first step the charge is put to all bubbles r_v simultaneously (in the rest of the description, we shall call these bubbles *blue*, whereas the bubbles w_S will be called *green*). The first step ends when no more charge can be put to blue bubbles, because some edge $e = (u, v)$ became full. The edge e is then added to the matching M, and the bubbles r_u and r_v won't be incremented in the next step: they will form a *dumbbell*. Ultimately, an edge e, whose one endpoint is a vertex that is already incident with an edge from M, becomes full. This edge is full, meaning that the bubbles at its ends cannot be increased, however, it cannot be added to M. We shall call it a *blocking* edge, and the algorithm shall maintain a set of blocking edges L.

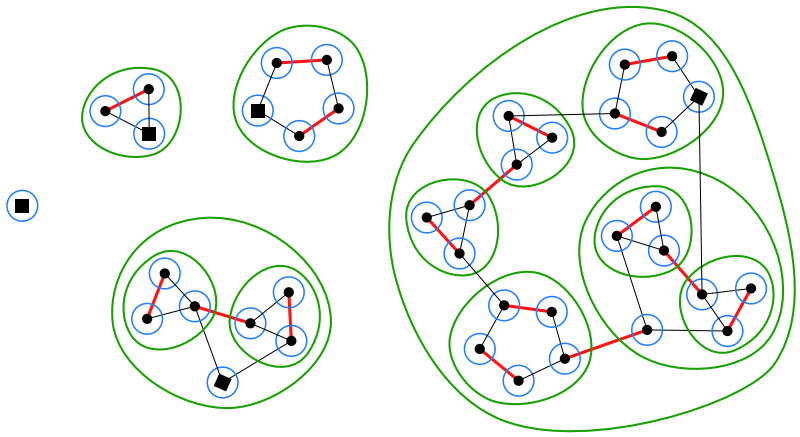


Apart from free bubbles and dumbbells, a new structure has just formed: a path of length 3 composed of a blocking edge from L, and a matching edge from M; paths containing alternating edges from L and M will be called *alternating paths*. If, in the next iteration, the charge of the *free* bubbles will be increased by $+\varepsilon$, the odd and even bubbles from the alternating paths will receive the change of $+\varepsilon$ and $-\varepsilon$, respectively, as to not overcharge any edge; recall that, unlike the green bubbles, the blue bubbles can have negative charge. In the next iteration, edges connecting to bubbles where the charge is being increase may become full, which gives rise to trees of the alternating paths (called *Hungarian trees*). A welcomed case is if the newly filled edge creates an alternating path with odd number of vertices, in this case the matching can be increased by swapping the membership of L and M edges along the path, and instead of an alternating path we have a set of dumbbells.



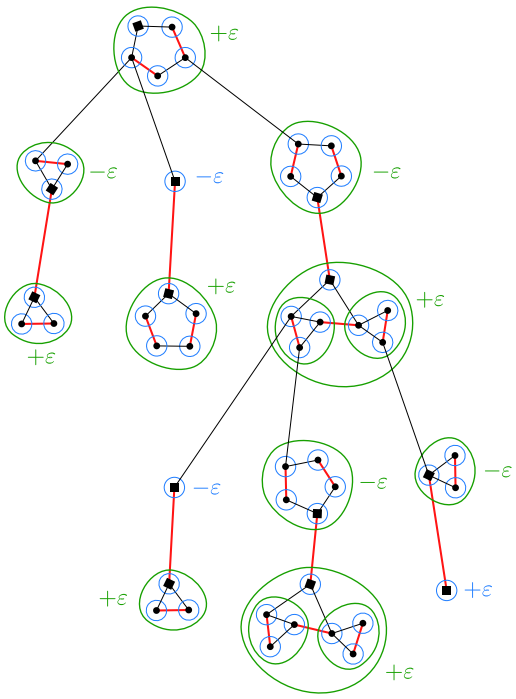
What should we do, however, if an edge becomes full, that connects two bubbles in a single tree? This is the moment when the time has come for the green bubbles to play their part, and to the hierarchic structures of bubbles.

The basic data structure of the algorithm is a *flower*. Each flower has an outer bubble, and inside it a distinguished vertex called *stem*. The most simple flower is a single vertex surrounded by a blue bubble. More complex flowers are formed by induction: take an odd number of flowers $K_1, K_2, \ldots, K_{2r+1}$, $r \geq 1$ (i.e. at least three flowers) such that the stems of the flowers K_{2i} and K_{2i+1} for $i = 1, \ldots, r$ are connected by an edge from M, and at the same time for each pair of flowers $A := K_{2i-1}$ and $B := K_{2i}$ for $i = 1, \ldots, r$, and $A := K_{2r+1}$ and $B := K_1$, there exist vertices $u \in A$ and $v \in B$, such that the edge $(u, v) \in L$. Then the green bubble surrounding $K_1, \ldots, K_{2r+1}$ created a new flower, whose stem is the stem of the flower K_1. Flowers that are not parts of another flower are called *outer flowers*.



Examples of flowers with stems denoted by a square. The red edges are from M, the black ones are from L.

It is worth noting that a flower encapsulates a part of the graph that is "almost done": with the exception of the stem, all vertices of the flower are pairwise matched by edges from M. At the same time exactly one edge from M leaves any bubble in flower that does not contain the stem; from the bubbles that contain the stem, on the other hand, leaves no edge from M. The previous observation is crucial for the understanding of the algorithm, and we recommend the reader to devise a detailed proof of it using induction. If there is a full edge connecting stems of two flowers, we can add it to M, and obtain a dumbbell. If all vertices of the graph are included in dumbbells, the algorithm is finished.

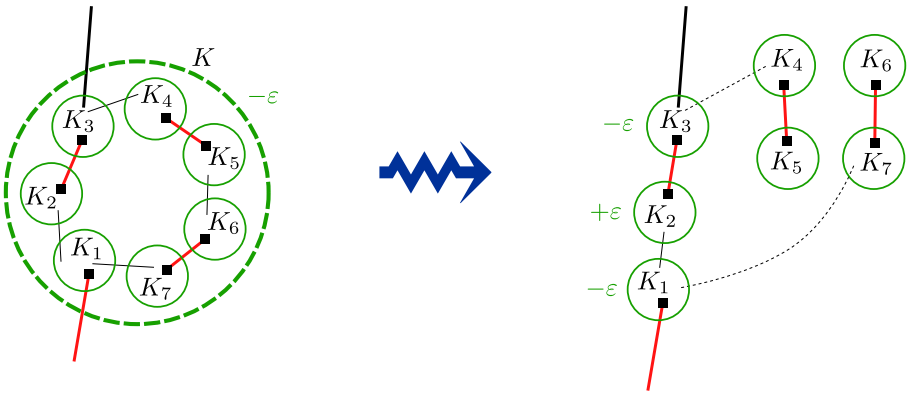


The *shift* operation on a Hungarian tree.

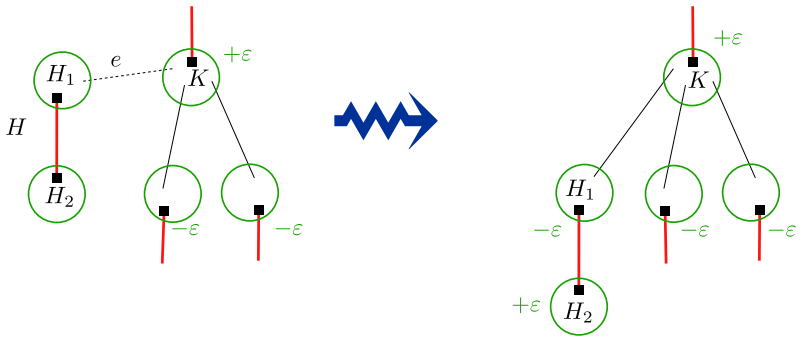
Free flowers that do not form dumbbells are organized in Hungarian trees. The flower on level 0 is *root* of the tree, the stem of a flower on level $2h-1$ is connected by an edge from M with the stem of a flower of level $2h$. If K is a flower on level $2h$, and H is a child of K in the tree, then there is an edge from L connecting some vertex from K to some vertex from H. The intuition is as follows: the root of the tree is a flower that we would like to include in a dumbbell, but in order to do so, we need a full edge connecting its stem to the stem of some other flower. However, there is no such edge, and we cannot add more charge to the outer bubble of the flower to make some edge full, because there are other edges crossing the outer bubble that are already full: the blocking edges from L; these edges lead to the flowers that are the children of the root, and so on.
During its computation, the algorithm maintains a set of Hungarian trees, and the rest of the graph is covered by dumbbells. The algorithm works in iterations: in one iteration it performs a *shift* operation on all trees. The operation adds some ε charge to the outer bubbles of all flowers on even levels, and subtracts the same value from the outer bubbles of flowers on odd levels. The value of ε is chosen as the biggest value that does not violate the conditions **(I1)**, **(I2)**. The conditions may be violated by several ways:

(P1) The charge of a green bubble on odd level dropped to 0. Let K be the flower

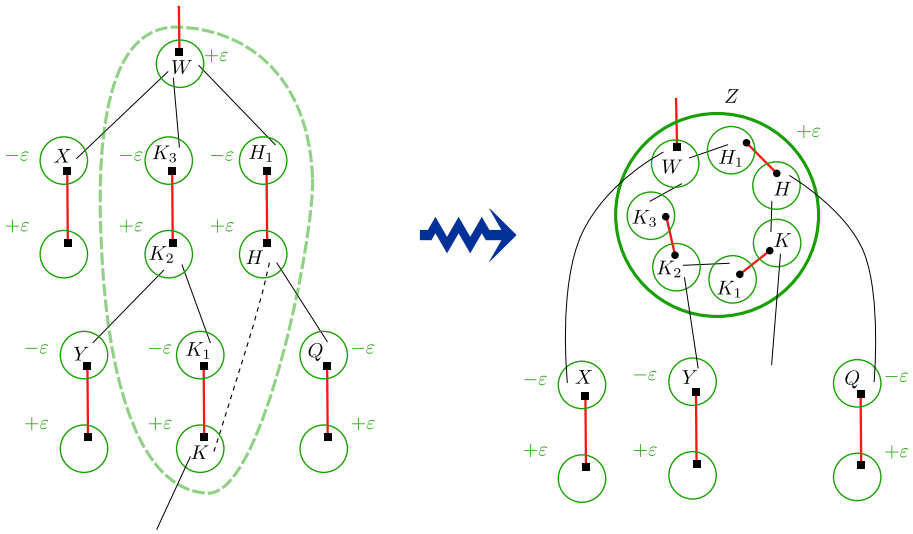
whose outer bubble reached zero charge. By the definition of a flower, K contains an odd number of flowers $K_1, \ldots, K_{2r+1}$, such that the stem of K is in K_1. Since K is on odd level in the tree, K has one parent, and (exactly) one child connected stem-to-stem by an edge from M. Let the edge from L connecting K to its parent is from some vertex in the flower K_t, and without loss of generality, let t be odd. Then the path $K_1, K_2, \ldots, K_t$ contains an odd number of flowers and can replace K in the tree. The pairs of flowers K_{t+1}, K_{t+2} up to K_{2r}, K_{2r+1} form dumbbells. The full edges between K_t and K_{t+1}, and K_1 and K_{2r+1} can be removed from L, since in the new tree, each of them has one endpoint on an odd level, and the other one in a dumbbell, so in the next iteration they will not be full anymore.



(P2) An edge e connecting a flower K on an even level, and a dumbbell H, has become full. Let the dumbbell H consist of flowers H_1 and H_2, so that e leads to some vertex in H_1. The edge e will be included in L, and H will join the corresponding tree so that K (on even level) will have a child H_1 (on odd level), and H_1 will have a child H_2 (on even level).



(P3) An edge e connecting flowers K and H in one tree has become full. Clearly, both K and H are on even level. Let W be the closest common ancestor of K and H. Since W has at least two children, it must be also on an even level. Let $K, K_1, \ldots, K_{2k+1}, W$ and $H, H_1, \ldots, H_{2r+1}, W$ be two paths in the tree. From the parity of their lengths it follows that they can be wrapped in a new outer bubble to obtain a flower Z on even level, whose stem is the stem of W. The children of Z are all the children of included flowers, they remain on odd levels.

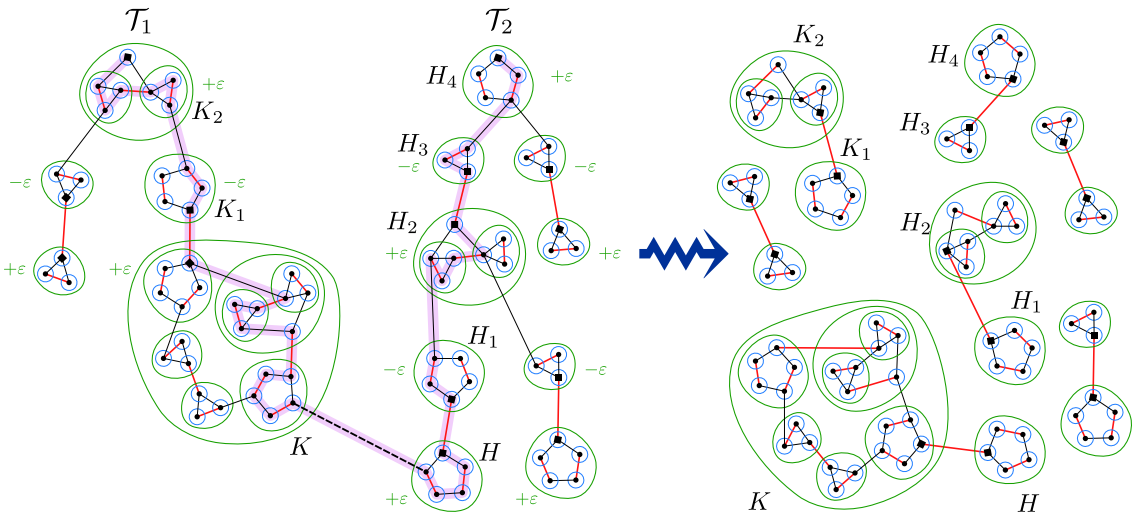


(P4) An edge e connecting flowers K and H in two different trees $\mathcal{T}_1$ and $\mathcal{T}_2$ has become full. This is actually the core of the algorithm where the matching M is increased. It is done by finding an alternating path connecting the stem of the root of $\mathcal{T}_1$ with the stem of the root of $\mathcal{T}_2$. Since an edge became full, both flowers K and H are on even levels in their respective trees, meaning that there is a path $K, K_1, \ldots, K_{2r}$ in $\mathcal{T}_1$, and a path $H, H_1, \ldots, H_{2q}$ in the tree $\mathcal{T}_2$, where K_{2r} and H_{2q} are roots of the respective trees. Both paths are formed by the edges of the trees, which are also edges of the original graph G, and edges belonging to L and M are alternating on them, such that an edge from M connects stems of neighboring flowers. In order to augment this alternating path in the trees to an alternating path in the graph G we observe the following:

Lemma 3.13. *Let K be a flower with a stem u, and let v be an arbitrary vertex from K. Then there exists an alternating path in G from u to v fully contained in K, and if this path is non-empty it starts with an edge from L and ends with an edge from M.*

Proof: We prove the lemma by induction on the depth of recursion of the flower. For a flower with a single vertex the statement holds trivially. Further, let K be a flowers composed of sub-flowers $K_1, \ldots, K_{2r+1}$ such that the stem of K is the stem of K_1. Without loss of generality let $v \in K_{2t-1}$ for some t (if $v \in K_{2t}$ we just change the direction of the numbering of the sub-flowers). If $t = 1$ we can use the induction hypothesis on the flower K_1. So let us assume $t > 1$. From the definition of the flower we can infer the existence of an edge $(q, w) \in L$ such that $q \in K_1$ and $w \in K_2$. By induction, there is an alternating $u - q$ path in K_1 that ends by an edge from M. At the same time there is an alternating path in K_2 from the stem to w that ends by an edge from M. Joining these two paths together, we get an alternating path from u to the stem of K_2 that ends by and edge from L, and so can be extended to the stem of K_3. Iterating this construction we obtain an alternating path to the stem of K_{2t-1} that can, again by the induction hypotheses, be extended to v. □

The construction from Lemma 3.13 enables us to find an alternating path from the stem of K_{2r} and the stem of K_{2q}. Along this path, we swap the membership of edges to L and M, thus increasing the number of edges in M by one. Subsequently, the trees $\mathcal{T}_1$ and $\mathcal{T}_2$ can be dismantled: the flowers K_{2i-1} and K_{2i} (and, similarly, H_{2j-1} and H_{2j}) form after the swap a set of dumbbells. To see this it is sufficient to note that the path from the proof of Lemma 3.13 either does not cross a given flower, or crosses it exactly twice, once by an edge from M, and once by an edge from L. Hence, after the swap we keep the invariant that every bubble is crossed by exactly one edge from M, and the stem of the tree is moved to the vertex v specified by Lemma 3.13. Similarly, the flowers K and H form a dumbbell, and the remaining subtrees of both $\mathcal{T}_1$ and $\mathcal{T}_2$ form dumbbells in a natural way.



The final algorithm works in a cycle: while M is not a 1-factor, one iteration finds the smallest value ε such that the shift by ε violates some of the conditions **(I1)**, **(I2)** from Lemma 3.12. According to what situation occured, one of the actions **(P1)**, **(P2)**, **(P3)** or **(P4)** is executed, and a next iteration begins. The condition **(I3)** remains true as an invariant. When the algorithm terminates, M is a 1-factor, and no flower can be a root of a tree (it would mean its stem is not matched in M, so M would not be a 1-factor); hence, all vertices are included in dumbbells, and the condition **(I4)** holds, too. Lemma 3.12 then implies that when the algorithm terminates, M is a minimum 1-factor.

The only thing to show now is that the algorithm indeed terminates, and if possible also that it terminates soon enough. Our argument is based on the following observation:

Lemma 3.14. *The described algorithm for the* Min-1-Factor *problem performs at most* $O(n^2)$ *iterations on an* n*-vertex graph* G*.*

Proof: The size of M never decreases, and is increased by 1 in every execution of the action **(P4)**. Hence, the algorithm may perform at most $O(n)$ times the action **(P4)**. In order to prove the statement it is sufficient to show that between two consecutive executions of the action **(P4)** there are at most $O(n)$ iterations of the algorithm, performing actions **(P1)**, **(P2)** and **(P3)**.

The first thing to note is that at any point during the execution there are at most $O(n)$ bubbles overall (including the bubbles inside flowers). This is due to the fact that every flower contains at least three sub-flowers, and so if we call the *depth* of the flower the length of a maximum chain of included bubbles, then at any time the algorithm can maintain at most n flowers of depth 0, $n/3$ flowers of depth 1, $n/3^2$ flowers of depth 2, and so on, forming a geometric series.

Now let us consider the computation of the algorithm between two executions of the action **(P4)**. Another important observation is that the charge of an outer bubble of a flower on even level will never decrease: either it remains as an outer bubble of a flower of even level, or it becomes part of another flower of even level in action **(P3)**). We shall call as *safe bubble* a bubble that was, at some point during the considered period of computation between two executions of **(P4)**, an outer bubble of a flower on even level. To finish the proof it is sufficient to show that actions **(P1)**, **(P2)** and **(P3)** increase the number of safe bubbles. <since the safe bubbles never disappear and there are linearly many bubbles overall, we obtain that between two consecutive executions of **(P4)**, at most linearly many other iterations may occur.

The action **(P1)** destroys a bubble B on odd level, and at least one bubble B' from its inside appears on an even level. This bubble, however, was not safe before, since a safe bubble never becomes a part of a flower on odd level. The action **(P2)** adds a safe bubble from the dumbbell, and the action **(P3)** creates a new safe bubble. □

A straightforward implementation of one iteration runs in time $O(nm)$, where n is the number of vertices, and m is the number of edges: for each edge, check all bubbles and check what largest ε this edge still allows. Select the edge with smallest bound, and perform the corresponding action. Hence, we can say:

Theorem 3.15. *The* Min-1-Factor *problem is solvable in time* $O(n^3m)$.

For an insightful reader we may add that using a more clever data structures, the running time can be significantly improved; it is, however, beyond the scope of this text.

Relaxed slackness conditions

In the previous part we introduced the primal-dual method based on the characterization of optimal solutions by slackness conditions. For a primal-dual pair of programs

$$\begin{aligned}(P): &\quad \min_{\mathbf{x}\in\mathbb{R}^n}\{\mathbf{c}^\mathsf{T}\mathbf{x} \mid A\mathbf{x}\geq\mathbf{b}, \mathbf{x}\geq 0\}\\ (D): &\quad \max_{\mathbf{y}\in\mathbb{R}^m}\{\mathbf{b}^\mathsf{T}\mathbf{y} \mid A^\mathsf{T}\mathbf{y}\leq\mathbf{c}, \mathbf{y}\geq 0\}\end{aligned}$$

the vectors $\mathbf{x}$ and $\mathbf{y}$ are optimal solutions of (P) and (D), respectively, if and only if

$$\begin{aligned}\forall\, 1\leq j\leq n: &\quad \text{either } x_j=0 \text{ or } \sum_{i=1}^m a_{ij}y_i=c_j\\ \forall\, 1\leq i\leq m: &\quad \text{either } y_i=0 \text{ or } \sum_{j=1}^n a_{ij}x_j=b_i\end{aligned}$$

Now we want to use this method to obtain an approximate solution of some NP-hard optimization problems. Can this characterization be of any help? Can we, for example, say that if we have a pair of vectors $\mathbf{x}$, $\mathbf{y}$ that violate the slackness conditions, but only a "little bit", then we can obtain a solution that is "almost" optimal? Indeed, we can modify the Theorem 3.8 as follows:

Theorem 3.16. *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be feasible solutions of program* (P) *and* (D)*, respectively, and let it, for some* $\alpha,\beta\geq 1$*, hold*

$$\begin{aligned}\forall\, 1\leq j\leq n: &\quad \text{either } x_j=0 \text{ or } c_j/\alpha\leq\sum_{i=1}^m a_{ij}y_i\leq c_j\\ \forall\, 1\leq i\leq m: &\quad \text{either } y_i=0 \text{ or } b_i\leq\sum_{j=1}^n a_{ij}x_j\leq\beta b_i\end{aligned}$$

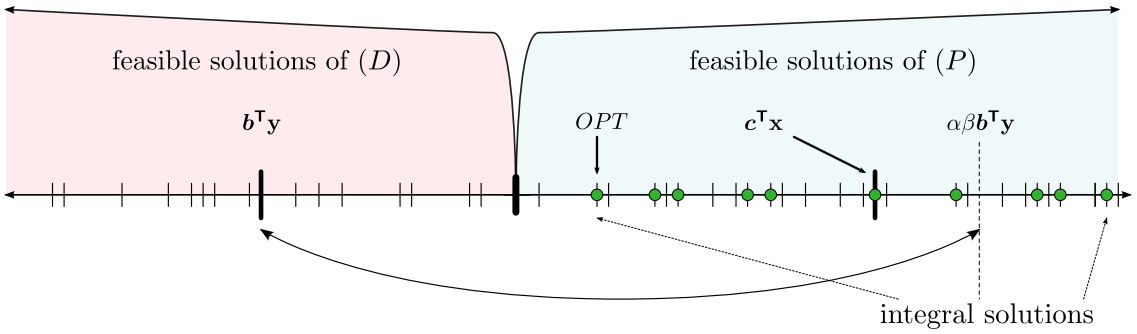
Then $\mathbf{c}^\mathsf{T}\mathbf{x}\leq\alpha\beta\mathbf{b}^\mathsf{T}\mathbf{y}$.

Proof:

$$\sum_{j=1}^n c_jx_j\leq\sum_{j=1}^n\alpha\left(\sum_{i=1}^m a_{ij}y_i\right)x_j\leq\alpha\sum_{i=1}^m y_i\left(\sum_{j=1}^n a_{ij}x_j\right)\leq\alpha\beta\sum_{i=1}^m y_ib_i$$

□

A typical application of this theorem looks like this: we are looking for the smallest integral solution of (P). If we manage, by some means, to find an integral solution $\mathbf{x}$, and some corresponding dual feasible solution $\mathbf{y}$ that satisfy the conditions of Theorem 3.16 , we are in a situation



Both OPT and the found solution $\mathbf{c}^\mathsf{T}\mathbf{x}$ are between $\mathbf{b}^\mathsf{T}\mathbf{y}$ and $\alpha\beta\mathbf{b}^\mathsf{T}\mathbf{y}$, and so the found solution is at most $\alpha\beta$ times the optimum.

As a concrete example we show (again) the 2-approximation algorithm for Min-Vertex-Cover (recall the Definition 2.10). It will, however, be much faster than the algorithm we already provided, since it will never need to solve any linear program explicitly. The start is the same: we want to solve the integer program

$$\begin{aligned} \text{minimize} \quad & \sum_{v\in V} \omega_v x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \qquad \forall e=(u,v)\in E \\ & x_v \geq 0 \qquad \forall v \in V \\ & x_v \in \mathbb{Z} \end{aligned} \tag{22}$$

We relax it to obtain

$$\begin{aligned} \text{minimize} \quad & \sum_{v\in V} \omega_v x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \qquad \forall e=(u,v)\in E \\ & x_v \geq 0 \qquad \forall v \in V \end{aligned} \tag{23}$$

Now, however, instead of solving (23) explicitly, we construct the dual program:

$$\begin{aligned} \text{maximize} \quad & \sum_{e\in E} y_e \\ \text{subject to} \quad & \sum_{\substack{e\in E \\ e=(u,v)}} y_e \leq \omega_u \qquad \forall u \in V \\ & y_e \geq 0 \qquad \forall e \in E \end{aligned} \tag{59}$$

While the program(22) asks to select vertices with minimum weight such that at least one endpoint of any edge is covered, the program (59) can be viewed as assigning a non-negative charge y_e to each edge e, and ω_v is the capacity of a vertex. The goal is to put as much charge (weighted by the ω_w) as possible to the graph, but no vertex can be overcharged: the sum of charges of edges incident to any vertex v cannot exceed its capacity. Let us write down the slackness conditions:

S1 $$\forall v \in V: \quad x_v > 0 \Rightarrow \sum_{\substack{e\in E \\ e=(u,v)}} y_e = \omega_u$$

S2 $$\forall e=(u,v)\in E: \quad y_e > 0 \Rightarrow x_u + x_v = 1$$

Following Theorem 3.8 we can say this: if we can select a set of vertices (i.e. integral values of $\mathbf{x}$), and put some charge to the edges such that no vertex is overcharged (feasible dual solution), every selected vertex is full (slackness conditions **S1**), and exactly one endpoint is selected from every edge with non-zero charge (slackness conditions **S2**), we would have an optimum solution. Clearly the most problems are caused by the conditions **S2**. However, if we settle with an approximation

with $\alpha = 1$ and $\beta = 2$, we don't have to care about **S2** at all, since $x_u + x_v \leq 2$ always holds. Theorem 3.16 ensures that we have a 2-approximation then.

We propose a simple greedy algorithm: it maintains a set of selected vertices C, and fore each edge e its currently assigned charge y_e. It starts by assigning $C = \emptyset$ and $y_e = 0$ for each edge e, and continues by a sequence of iterations: while C is not a vertex cover, pick an arbitrary edge $e = (u, v)$ that is not covered by C, and increase y_e to the maximum value that still overcharges neither u not v. After this increase, obviously, at least one of the vertices u, v is full, and so some full vertex from $\{u, v\}$ can be entered into C.

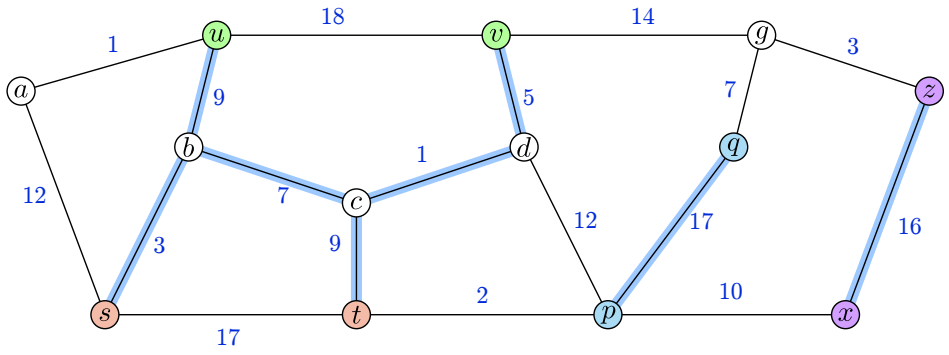
After the algorithm terminates, it holds that C is a cover: the main cycle of the algorithm does not finish while there is some uncovered edge. Moreover, no vertex is overcharged and every selected vertex is full, since these conditions were maintained as invariants. Hence, the previous theorem implies that the cost of the found solution is at most twice the optimum cost. The algorithm works in time $O(n + m)$, where n is the number of vertices and m is the number of edges.

3.4 Weaker than relaxed slackness: Min-Steiner-Forest

We have seen how to use the relaxed version stated in Theorem 3.16 of the slackness conditions to design approximation algorithms. In this part we introduce an example where we will not be able to ensure the relaxed slackness conditions to hold, and yet we will be able to design a primal-dual approximation algorithm by arguing about the slackness conditions "on average".

Consider the following toy example: given is a railroad network modelled by a graph: vertices represent cities, and edges are the railroad tracks. A carrier who wants to operate certain connections has to rent some tracks from the owner, and for each track there is a fixed non-negative rental cost. The carrier plans to operate a set of connections between pairs of cities, and seeks to rent tracks that enable these connections to be operated while minimizing the rental costs. A formal definition could look like this:

Definition 3.17. Given is an undirected graph $G = (V, E)$ with edge costs $\omega : E \mapsto \mathbb{R}^+$, and a (symmetric) mapping $r : V \times V \mapsto \{0, 1\}$. The goal of the Min-Steiner-Forest problem is to find a set of edges $F \subseteq E$ such that for any pair of vertices u, v for which $r(u, v) = 1$ there is a $u - v$ path containing only edges from F, and the overall cost of edges from F is minimized.



An example of a graph with edge costs (blue). The required connections are $u - v$, $s - t$, $p - q$, and $x - z$ (i.e. $r(u, v) = r(v, u) = r(s, t) = r(t, s) = r(p, q) = r(q, p) = r(x, z) = r(z, x) = 1$, and all remaining values $r(\cdot, \cdot)$ are zero). The highlighted edges form optimum solution with cost 51. Note that an optimal solution of Min-Steiner-Forest is always a forest.

A reader familiar with the theory of NP-completeness, considers it an easy exercise to show that Min-Steiner-Forest is an NP-hard problem, and thus would not expect to find a polynomial-time algorithm that solves it optimally. We present here a 2-approximation algorithm, i.e. a polynomial-time algorithm that always finds a solution with the cost not bigger than twice the cost of the optimum solution. Let us start as usual by formulating the problem as an ILP. Quite naturally, for each edge e we introduce a variable $x_e \in \{0,1\}$ that would denote whether e is selected or not. We need now to express the connectivity requirements in the form of linear constraints. Since we are about to use the primal-dual method, we don't care about the size of the program, and we happily introduce exponentially many constraints. In order to simplify the exposition, let us call a set S *hungry* if there exist $u \in S$, $v \in V \setminus S$ for which $r(u,v) = 1$, i.e. at least one edge leaving S must be included in any feasible solution. We observe the following:

Lemma 3.18. *A set of edges F is a feasible solution if and only if for each hungry set S there is at least one edge from F that leaves S.*

Proof: Obviously, in any feasible solution there must be at least one edge leaving any hungry set. In order to prove the opposite direction, consider such set F, and arbitrary vertices $u, v \in V$ such that $r(u,v) = 1$; we find a $u - v$ path in F. Let us construct, by induction, a sequence of sets $\{u\} = S_0 \subseteq S_1 \subseteq \cdots$ with the property that for every vertex $w \in S_i$, there is some $u - w$ path in F. For $\{u\} = S_0$ the statement obviously holds. Take now an arbitrary set S_i. If $v \in S_i$, we have a $u - v$ path from F, and we are done. If $v \notin S_i$, then S_i is hungry and so there is some edge from F leaving S_i; let this edge lead to a vertex $w \in V \setminus S_i$. Setting $S_{i+1} := S_i \cup \{w\}$ keeps the property that there is an $u - w$ path in F for every vertex $w \in S_{i+1}$ is satisfied. Since there are n vertices, and we add one vertex to the set in every step, we eventually reach a situation where $v \in S_i$. □

We shall write $f(S) = 1$, if S is hungry, and $f(S) = 0$ otherwise. Following Definition 3.11 let us call $\partial(S)$ the edge boundary of a set S. The problem Min-Steiner-Forest can be expressed as an ILP as follows:

$$\begin{array}{rlll} \text{minimize} & \sum_{e \in E} \omega_e x_e & & \\ \text{subject to} & \sum_{e \in \partial(S)} x_e \;\geq\; f(S) & \forall S \subseteq V & (60) \\ & x_e \in \{0,1\} & \forall e \in E & \end{array}$$

This program can be subsequently relaxed by changing the integrality constraints to $x_e \geq 0$; the condition $x_e \leq 1$ is implied by minimization. The relaxed program is

$$\begin{array}{rlll} \text{minimize} & \sum_{e \in E} \omega_e x_e & & \\ \text{subject to} & \sum_{e \in \partial(S)} x_e \;\geq\; f(S) & \forall S \subseteq V & (61) \\ & x_e \;\geq\; 0 & \forall e \in E & \end{array}$$

Now let us construct a dual program to (61): we introduce variables y_S for each set $S \subseteq V$, and write

$$\begin{array}{rlll} \text{maximize} & \sum_{S \subseteq V} y_s f(S) & & \\ \text{subject to} & \sum_{S : e \in \partial(S)} y_S \;\leq\; \omega_e & \forall e \in E & (62) \\ & y_S \;\geq\; 0 & \forall S \subseteq V & \end{array}$$

Programs with similar structure have already appeared several times, so we have a natural way to interpret the program: there is a bubble with some charge around each set. The goal is to maximize the overall charge of hungry sets (sets that are not hungry do not contribute to the final solution, and we now make a firm resolution to never ever increase any dual variable belonging

to a non-hungry set). The requirement is not to overload any edge: the sum of the charges on bubbles that a particular edge crosses must not exceed the capacity of the edge. Note that to increase the charge of a set y_S has the same effect as to increase the set $y_{V \setminus S}$. Let us now observe the slackness conditions:

$$\textbf{S1} \quad \forall e \in E: \quad x_e > 0 \Rightarrow \sum_{S: e \in \partial(S)} y_S = \omega_e$$

$$\textbf{S2} \quad \forall S \subseteq V: \quad y_S > 0 \Rightarrow \sum_{e \in \partial(S)} x_e = f(S)$$

From the slackness conditions we can conclude that in a hypothetical optimum solution with integral $\mathbf{x}$, every selected edge is full, and from every non-zero bubble around a hungry set leaves exactly one selected edge (we already promised not to increase any bubble around a non-hungry set).

Our algorithm will iteratively add edges to the set of selected edges, until all connectivity requirements are satisfied. At the same time it will maintain invariants that all selected edges are full, and that no edge is overloaded. Thus when the algorithm terminates it will have a feasible solution of the program (61), and a feasible solution of the dual program (62). Also, the conditions **S1** will be satisfied. If we wanted to use Theorem 3.16 to guarantee the approximation ratio of 2, we would need to ensure, additionally, the conditions

$$\textbf{S2}' \quad \forall S \subseteq V: \quad y_S > 0 \Rightarrow \sum_{e \in \partial(S)} x_e \leq 2f(S),$$

i.e. that from every non-zero bubble (around a hungry set) there are at most two outgoing selected edges. We will not be able to guarantee this, but as we shall see, a weaker statement that *on average* there are at most two outgoing edges from any bubble will be both sufficient and provable.

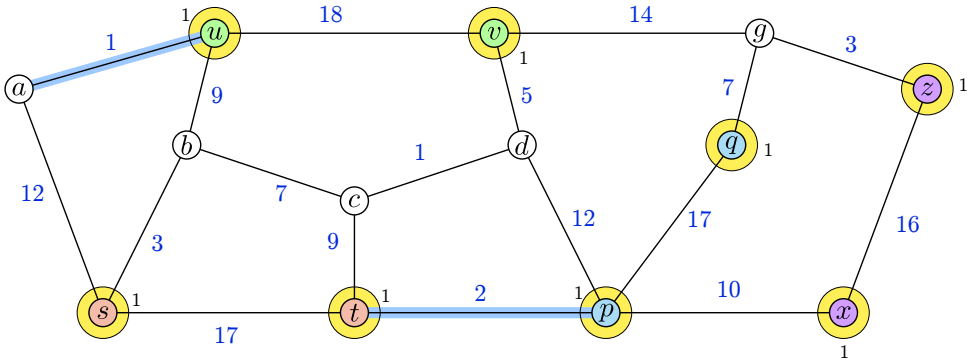
The structure of the algorithm will be very similar to the primal-dual algorithm for Min-Vertex-Cover from the previous section. The algorithm starts with an empty set of selected edges, and all-zero bubbles. While the algorithm for Min-Vertex-Cover selected in one iteration a single variable, increased it as much as possible, and selected one from the filled vertices, our algorithm will in one iteration increase several dual variables in parallel, and admits one of the filled edges to the solution.

To what bubbles we are going to add charge in a single iteration? Clearly, they must correspond to hungry sets (remember, no charge ever to sets that are not hungry). Moreover, since all edges selected so far are already full, a bubble around a hungry set from which already leaves at least one selected edge, cannot be charged more. This leaves us only the possibility to put charge to bubbles around hungry sets (there must be some edge leaving them in any feasible solution), with no selected edge that leaves them; such bubbles will be called *unhappy*. These are the sets that violate the feasibility of program (61)). Potentially, there may be exponentially many unhappy sets, which could pose problems, because the algorithm may explicitly store only polynomially many dual variables. However, it is easy to see that

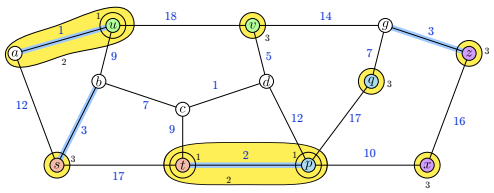
Lemma 3.19. *Unhappy sets that are minimal with respect to inclusion are the connected components of the graph induced by the selected edges.*

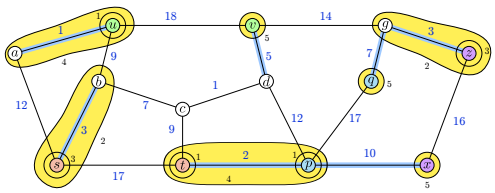
Proof: A set if unhappy if it is hungry and there is no selected edge leaving it. Hence any unhappy set is a union of several connected components of the graph induced by selected edges. An inclusion-minimal unhappy set is then a single connected component. □

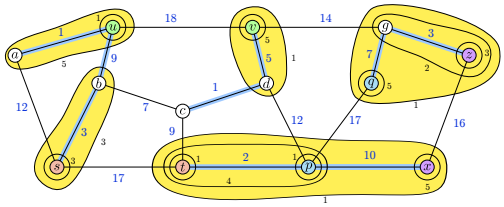
The algorithm increases in every iteration the dual variables corresponding to the unhappy connected components. At the beginning there are no selected edges, and so the unhappy components are singleton bubbles around vertices that need to be connected. For the instance from the introductory example, these are the sets $\{u\}, \{v\}, \{s\}, \{t\}, \{p\}, \{q\}, \{x\}, \{z\}$. The algorithm increases the corresponding dual variables until some edges become full. In our case, if the dual variables are increased to 1, edges $\{a, u\}$ and $\{t, p\}$ become full.

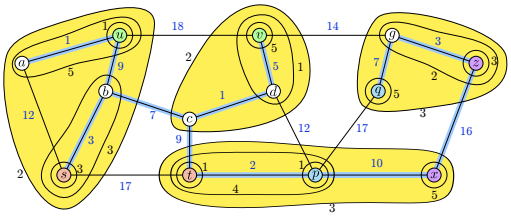


In the next iteration the unhappy components are $\{a, u\}, \{s\}, \{v\}, \{t, p\}, \{q\}, \{z\}, \{x\}$, and the algorithm increases their charge by 2, thus filling the edges $\{s, b\}$ and $\{g, z\}$. New unhappy components are formed, and the computation continues as follows:



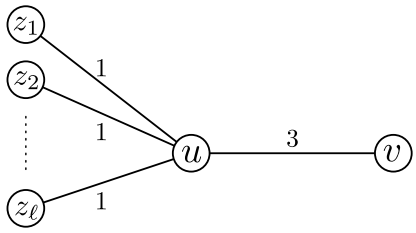






The algorithm terminates if all the connected components formed by the selected edges are happy. For an ease of presentation, it is easier to suppose that in each iteration exactly one edge is selected. If several edges connecting different components are filled, the subsequent iterations increase the charge by zero amount; of course, an actual implementation would avoid this.

The algorithm, as presented so far, has one tiny problematic detail: it does not work at all. Consider, e.g. the following simple graph there the only connectivity requirement is $r(u, v) = 1$.



At the beginning the unhappy components are $\{u\}$ and $\{v\}$ and the algorithm assigns a charge 1 to both of them. This, however, fills the edges $\{z_i, u\}$, so the algorithm subsequently selects all the edges, creating a solution with size $\ell+3$. The optimal solution, however, has cost three. Let us try to save our algorithm with a quick fix: after the algorithm terminates, the solution is modified by removing all unnecessary edges; an edge is *enecessary* if $F - \{e\}$ is not a feasible solution.

The final algorithm looks as follows:

---

1 $F := \emptyset$, $y_{\{v\}} := 0$ for each $v \in V$

2 while there exists an unhappy connected component induced by edges from F

  3 increase y_S for all sets S corresponding to unhappy connected components from F until some edge e is filled

  4 $F := F \cup e$

5 $F' := F$

6 for each edge $e \in F$

  7 if $F - \{e\}$ is feasible, $F' := F' - \{e\}$

First let us make sure that the algorithm is correct:

Lemma 3.20. *After the algorithm terminates, edges F' form a feasible solution of the program (60), and the values y_S form a feasible solution of the program (62).*

Proof: After the cycle on line 2 terminates, there is no unhappy connected component induced by the edges F. Since every unhappy set is a union of some connected components, F contains an outgoing edge from every hungry set, and so F is a feasible solution of (60). It remains to check that the lines 6 and 7 do not change this property. Recall that an edge e is necessary if $F - \{e\}$ is not feasible. We show that if all unnecessary edges are removed at the same time from F, the remaining set F' is feasible. This follow immediately from the observation that the edges from F induce an acyclic graph: indeed, on line 3 only y_S values for a connected components S are increased, so an edge inside a connected component can not become full. Hence, every selected edge connects two components, and so cannot induce a cycle. For any two vertices u, v that must be connected, i.e. $r(u, v) = 1$, there is exactly one $u - v$ path in F, so all edges on it are necessary, and thus end up in F'.
On the other hand, the values y_S are only increased to the extent that no edge is overcharged, so during the whole computation, the y_S are a feasible solution of (62). □

We see that after termination of the algorithm, we have some feasible solution F', and some feasible solution $\mathbf{y}$ of the dual program. In order to derive a guarantee on the approximation ratio, we need to compare the value of the solution F' with the value of the optimum. We, of course, don't know the exact optimum, but we know it is at least as big as any feasible solution of the dual program. So in order to prove 2-approximation it is sufficient to show that the value of F' is at most twice the value of $\mathbf{y}$, i.e.

Theorem 3.21.

$$\sum_{e \in F'} \omega_e \leq 2 \sum_{S \subseteq V} y_S f(S)$$

Proof: Since we never put any charge to non-hungry sets, $f(S) = 0$ implies $y_S = 0$, so we want to prove

$$\sum_{e \in F'} \omega_e \leq 2 \sum_{S \subseteq V} y_S.$$

Now let us introduce the following notation: for some sets $W \subseteq E$ and $S \subseteq V$, denote $\deg_W(S) := |W \cap \partial(S)|$, i.e. the number of edges from W that have one endpoint in S and the other outside S. Every edge that was selected to F (and hence to F') is full, since the charge is never decreased, and the edge was full when if was included in F; for the left-hand side we thus get

$$\sum_{e \in F'} \omega_e = \sum_{e \in F'} \left(\sum_{S : e \in \partial(S)} y_S \right) = \sum_{S \subseteq V} \deg_{F'}(S) y_S.$$

We need to prove

$$\sum_{S \subseteq V} \deg_{F'}(S) y_S \leq 2 \sum_{S \subseteq V} y_S \tag{63}$$

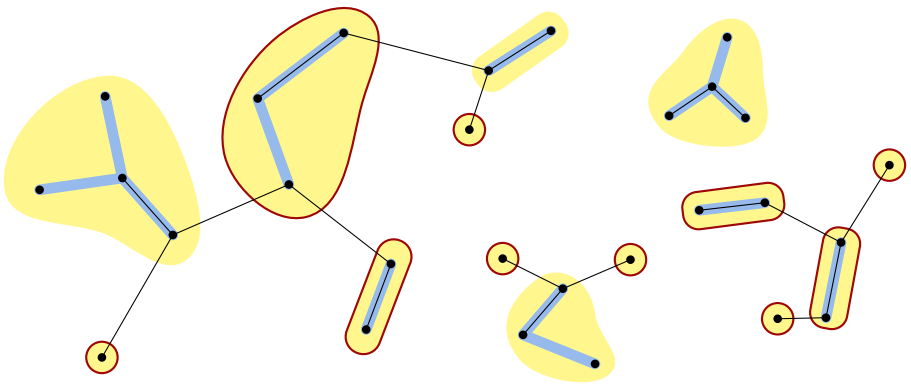
We show this by induction on the number of the iterations of the algorithm. At the beginning, the statemet (63) holds trivially, since all the values $y_S = 0$. Consider now some iteration ℓ, in which all the values y_S of the unhappy sets S were increased by Δ. How this changed the (63)? Each unhappy component S adds $\Delta \deg_{F'}(S)$ to the left-hand side, and 2Δ to the right-hand side. In order to prove (63), we show that the increase to the right-hand side is bigger than the increase to the left-hand side, i.e.

$$\Delta \left(\sum_{S \in \mathscr{S}_\ell} \deg_{F'}(S) \right) \leq 2\Delta |\mathscr{S}_\ell|, \tag{64}$$

where $\mathscr{S}_\ell$ is the family of all unhappy components in this iteration. The inequality (64) can be rewritten as

$$\frac{\sum_{S \in \mathscr{S}_\ell} \deg_{F'}(S)}{|\mathscr{S}_\ell|} \leq 2,$$

i.e. we need to show that the average degree of an unhappy component, with respect to F', is at most 2. If we denote by F_ℓ the set of selected edges at the beginning of iteration ℓ, the situation looks as follows:



The black edges are the final solution F'. The blue edges are the currently selected edges F_ℓ. The highlighted components are unhappy.

Both the edges F' and F_ℓ are subsets of F. Since F induce a forest (see the proof of Lemma 3.20), $F' \cup F_\ell$ induce a forest, too. If every connected component of F_ℓ is contracted to a single vertex (the yellow blobs on the previous figure), a forest H is obtained, whose vertices correspond to connected components of F_ℓ, and the edges are formed by the edges of F'. To prove the inequality (64) means to prove that the average degree of a vertex in H is at most two, where the average is taken only from the vertices that correspond to unhappy components. If not for this last restriction, we could end the proof here: any forest with n vertices has at most $n-1$ edges, implying that the sum of the degrees is at most $2(n-1)$, and the average is thus less than 2. However, we need to bound the average degree of the vertices corresponding to unhappy components only. In order to do so, we show that the vertices corresponding to *happy* sets cannot have degree 1: they are either isolated, or have degree at least two. After we prove this, the proof will be concluded as follows: no unhappy component has degree 0 in H, because there is at least one outgoing edge in F' from it. Isolated vertices in H thus correspond to happy components. If we remove isolated vertices from H, we obtain a new forest H': it is sufficient to prove that the average degree of unhappy components in H' is at most 2. However, the average degree of all components (vertices of H') is at most 2, and every happy component has degree at least two. So it must be that the average degree of unhappy components cannot be more than 2.

To conclude we show that a happy component cannot have degree 1 in H. Suppose by contradiction that there exists a connected component C in F_ℓ, such that there is exactly on outgoing edge e from C in F'. The edge e survived from F to F', because it is necessary, i.e. it is part of the unique path in F connecting two vertices u, v that must be connected ($r(u,v) = 1$). However, if e is the only outgoing edge from C in F', then the vertices u, v must be located one in C and the other outside C, But then C cannot be happy in F_ℓ, a contradiction.

□

Bibliography

[1] G. Ausiello. *Complexity and approximation: combinatorial optimization problems and their approximability properties*. Springer, 1999.

[2] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. *Machine Learning*, 56:89–113, 2004.

[3] R. G. Bland. A combinatorial abstraction of linear programming. *J. Comb. Theory, Ser. B*, 23(1):33–57, 1977.

[4] V. Chvátal. *Linear Programming*. Series of Books in the Mathematical Sciences. W.H. Freeman, 1983.

[5] D. Emanuel and A. Fiat. Correlation clustering - minimizing disagreements on arbitrary weighted graphs. In G. D. Battista and U. Zwick, editors, *ESA*, volume 2832 of *Lecture Notes in Computer Science*, pages 208–220. Springer, 2003.

[6] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric algorithms and combinatorial optimization*. Algorithms and combinatorics. Springer-Verlag, 1988.

[7] J. Håstad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[8] V. Klee and G. J. Minty. How Good is the Simplex Algorithm? In O. Shisha, editor, *Inequalities III*, pages 159–175. Academic Press Inc., New York, 1972.

[9] G. Lancia, V. Bafna, S. Istrail, R. Lippert, and R. Schwartz. Snps problems, complexity, and algorithms. In F. Meyer auf der Heide, editor, *ESA*, volume 2161 of *Lecture Notes in Computer Science*, pages 182–193. Springer, 2001.

[10] A. Lubotzky. *Discrete Groups, Expanding Graphs and Invariant Measures*. Progress in Mathematics. Birkhäuser, 1994.

[11] J. Matoušek and B. Gärtner. *Understanding and Using Linear Programming*. Universitext (En ligne). Springer, 2007.

[12] A. Sankar, D. A. Spielman, and S.-H. Teng. Smoothed analysis of the condition numbers and growth factors of matrices. *SIAM J. Matrix Analysis Applications*, 28(2):446–476, 2006.

[13] D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004.

[14] V. V. Vazirani. *Approximation Algorithms*. Springer, Mar. 2004.

[15] D. Williamson and D. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.