Network Flows

Course notes for Optimization
(version 1.1)

Kristoffer Arnsfelt Hansen Peter Bro Miltersen

March 8, 2017

1 Totally unimodular linear programs

1.1 The assignment problem

Let us consider the classical assignment problem. Here we are given a $n \times n$ matrix of costs $C = (c_{ij})$ and are to find a permutation π on $\{1, 2, \ldots, n\}$ minimizing $\sum_{i=1}^{n} c_{i,\pi(i)}$. If we introduce the n^2 decision variables

$$x_{ij} = \begin{cases} 1 & \text{if } \pi(i) = j \\ 0 & \text{otherwise} \end{cases},$$

we can formulate this problem as an integer LP problem as follows:

$$\begin{array}{llcll} min & \sum_{i,j=1}^{n} c_{ij}x_{ij} & & & \\ s.t. & \sum_{j=1}^{n} x_{ij} & = & 1 & i = 1, \ldots, n \\ & \sum_{i=1}^{n} x_{ij} & = & 1 & j = 1, \ldots, n \\ & x_{ij} & \geq & 0 & i, j = 1, \ldots, n \\ & x_{ij} & \in & \mathbb{Z} & \end{array} \tag{1}$$

Running the simplex algorithm on instances of the assignment problem by ignoring the integrality constraints one discovers that the optimal solution one gets as output seems always to be integral. Let us see why. First we rewrite the linear program into standard form, introduce slack variables, and consider the resulting linear system with $4n$ equations in $n^2 + 4n$ variables.

$$\begin{aligned} w_i &= 1 - \sum_{j=1}^{n} x_{ij} & i &= 1, \ldots, n \\ w_{n+i} &= -1 + \sum_{j=1}^{n} x_{ij} & i &= 1, \ldots, n \\ w_{2n+j} &= 1 - \sum_{i=1}^{n} x_{ij} & j &= 1, \ldots, n \\ w_{3n+j} &= -1 + \sum_{i=1}^{n} x_{ij} & j &= 1, \ldots, n \end{aligned}$$

Any basic solution may be obtained from this linear system by setting n^2 of the variables to 0 and solving (uniquely) for the remaining $4n$ basic variables.

In matrix form, the linear system can be written as

$$\left[\begin{array}{r|c} L & \\ -L & I \\ R & \\ -R & \end{array}\right]\begin{bmatrix} x \\ w \end{bmatrix} = \begin{bmatrix} \mathbf{1} \\ -\mathbf{1} \\ \mathbf{1} \\ -\mathbf{1} \end{bmatrix} , \tag{2}$$

where entry $(k,(i,j))$ of L is 1 if $k = i$ and 0 otherwise, and similarly entry $(k,(i,j))$ of R is 1 if $k = j$ and 0 otherwise.

When we set a variables to 0 it corresponds to removing the variable from the system together with the corresponding column of the coefficient matrix. Thus letting $\widehat{x}$ denote the vector of the $4n$ remaining basic variables and letting $\widehat{A}$ be the $4n \times 4n$ matrix formed by the remaining columns of the coefficient matrix we obtain the following linear system:

$$\widehat{A}\widehat{x} = \begin{bmatrix} \mathbf{1} \\ -\mathbf{1} \\ \mathbf{1} \\ -\mathbf{1} \end{bmatrix} . \tag{3}$$

As the linear system has a unique solution it may be solved using *Cramer's rule*, which we will now recall in the general setting.

1.2 Cramer's rule and matrix determinants

Let $A = (a_{ij})$ be a $m \times m$ matrix, $b \in \mathbb{R}^m$, and consider the linear system

$$Ax = b \tag{4}$$

When the system has a unique solution it can be found using Cramer's rule.

Theorem 1 (Cramer's rule). *The system (4) has a unique solution if and only if* $\det(A) \neq 0$, *and in that case it is given by*

$$x_i = \frac{\det(A_i)}{\det(A)} ,$$

where A_i is obtained from A by replacing column i by the vector b.

Observation 1. *Suppose that all entries of A and b are integers. Then* $\det(A_i)$ *is integer for all i. Now, a sufficient condition for x_i to be integer for all i is that* $\det(A) \in \{-1, 1\}$.

Let us recall some properties of determinants. First, Laplace's formula allows us to expand the calculation of the determinant along a row or a column.

Proposition 1. *Let $A = (a_{ij})$ be a $m \times m$ matrix. Then*

$$\det(A) = \sum_{j=1}^{m} (-1)^{i+j} a_{ij} \det(A_{ij}) = \sum_{i=1}^{m} (-1)^{i+j} a_{ij} \det(A_{ij}) ,$$

where A_{ij} is the $(m-1) \times (m-1)$ matrix obtained from A by removing row i as well as column j.

Second, we have a number of formulas describing how the determinant behaves under simple operations on the matrix.

Proposition 2. *Let A be a $m \times m$ matrix. Then*

1. *If B is obtained from A by exchanging two rows or by exchanging two columns, then* $\det(B) = -\det(A)$.
2. *If B is obtained from A by multiplying a row or a column by c, then* $\det(B) = c\det(A)$.
3. *If B is obtained from A by adding a multiple of a row to another row or by adding a multiple of a column to another column, then* $\det(B) = \det(A)$.

1.3 Basic solutions for the assignment problem

Let us now return to the assignment problem. In the linear system (3) all coefficients are integers, so by Observation 1, in order to show that all basic solutions are integer it is sufficient to show that $\det(\widehat{A}) \in \{-1, 1\}$. Recall that we know $\det(\widehat{A}) \neq 0$ since the linear system has a unique solution. We are going to evaluate $\det(\widehat{A})$ by expanding the determinant along rows and columns using Proposition 1, following the rule that as long as there is a row or a column with a single non-zero entry (which must then be -1 or 1) we expand along that row or column. We claim that this process ends up with a 1×1 matrix. And that entry is then (up to a change of sign) the determinant of $\widehat{A}$.

First, observe that the process will have removed all the columns arising from the $4m \times 4m$ identity matrix in system (2). Thus after that we are to compute a determinant of a square submatrix[1] M of the matrix:

$$\begin{bmatrix} L \\ -L \\ R \\ -R \end{bmatrix} .$$

Now observe that if M would contain row i from block L as well as row i from block $-L$, the rows of M would be linearly dependent, and this would mean that $\det(M) = 0$. Similarly if M would contain row j from block R as well as row j from block $-R$, it would again mean that $\det(M) = 0$. Since this is not the case, we may after reordering and multiplying some of the rows by the constant -1 instead consider a square submatrix M' of the matrix

$$\begin{bmatrix} L \\ R \end{bmatrix} ,$$

and of the same dimensions as M for which $\det(M') = \pm\det(M)$. In the column indexed by (i, j) this matrix has exactly 2 non-zero entries, namely 1-entries in row i of block

[1] An $r \times r$ submatrix of an $m \times n$ matrix A is obtained by first removing all but r rows of A and then removing all but r columns of the resulting $r \times n$ matrix

L and row j of block R. If M' is not already a 1×1 matrix, then each column of M' must have exactly two 1-entries as well, one from a row originating from block L and one from a row originating from block R. But then if we sum the rows originating from the block L and sum the rows originating from the block R we obtain the same result, which means that the rows of M' are linearly dependent, and hence $\det(M') = 0$. Since this is not the case, M' must already be a 1×1 matrix, and we get that $\det(\widehat{A}) \in \{-1, 1\}$.

The final conclusion is that all basic solutions of the linear program (1) for the assignment problem are integer.

1.4 Totally unimodular matrices

We are now going to investigate further when a linear program has the property that all its basic solutions are integer.

Definition 1. A matrix A is called *totally unimodular* if every square submatrix of A has determinant either 0, 1 or -1.

It turns out that this property of integer matrices *exactly* characterizes the property we are interested in.

Theorem 2 (Hoffman and Kruskal's Theorem)**.** *Let A be an integer $m \times n$ matrix. Then A is totally unimodular if and only if for every integer vector $b \in \mathbb{Z}^m$ all the basic solutions of $F = \{Ax \leq b, x \geq 0\}$ are integer.*

Here we will only be concerned with establishing the "only if" direction of the above theorem:

Theorem 3. *Let $A = (a_{ij})$ be a totally unimodular $m \times n$ matrix and let $b \in \mathbb{Z}^m$. Then every basic solution of $F = \{Ax \leq b, x \geq 0\}$ is integer.*

Proof. Any basic solution may be obtained by introducing slack variables $x_{m+1}, \ldots, x_{m+n}$, selecting n non-basic variables to be set to 0 in the system

$$x_{m+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \qquad i = 1, \ldots, m \ , \tag{5}$$

and solving (uniquely) for the remaining m basic variables. In matrix form, system (5) can be written as:

$$\begin{bmatrix} A & I \end{bmatrix} x = b \ , \tag{6}$$

where I is the $m \times m$ identity matrix and $x \in \mathbb{R}^{n+m}$ consists of the original n variables and the m slack variables. Let $B \subseteq \{1, \ldots, n+m\}$ be the indices of the m chosen basic variables. Now, setting $x_i = 0$ for $i \notin B$ corresponds to removing column i from the matrix $\begin{bmatrix} A & I \end{bmatrix}$ and removing variable x_i. We end up with a system

$$\widehat{A}\widehat{x} = b \ , \tag{7}$$

where $\widehat{A}$ is the $m \times m$ matrix formed by the columns of A indexed by B and $\widehat{x}$ is the vector of basic variables. By Theorem 1 we have

$$\widehat{x}_i = \frac{\det(\widehat{A}_i)}{\det(\widehat{A})} \qquad i = 1, \ldots, m \ .$$

Since A and b are integer we have that $\det(\widehat{A}_i)$ is integer as well. To evaluate $\det(\widehat{A})$ we expand the determinant along the columns originating from the $m \times m$ identity matrix in the linear system (6) using Proposition 1. Doing this we see that $\det(\widehat{A})$ is up to a change of sign equal to the determinant of a square submatrix of A, which by assumption belongs to $\{-1, 0, 1\}$. Thus $\det(\widehat{A}) \in \{-1, 1\}$, since we know $\det(\widehat{A}) \neq 0$. This furthermore means that all entries of $\widehat{x}$ are integer, which was to be proved. □

We will extend the terminology to linear programs, and call a linear program in standard form for totally unimodular if the coefficient matrix A is totally unimodular and the vector of constants b is integer valued.

Being totally unimodular is an extremely desirable property of a linear program, since in many real-life situations the solutions we are interested in are actually the integer solutions. When the linear program in totally unimodular we are able to efficiently find an optimal integer solution, for instance using the simplex algorithm!

Having now established the usefulness of totally unimodular matrices, we will explore the set of these matrices. We do this by generalizing in several ways the analysis made when we considered the assignment problem.

Lemma 1. *If A is a matrix with entries from $\{-1, 0, 1\}$ with the property that each column contains at most two non-zero entries, at most one being 1 and at most one being -1, then A is totally unimodular.*

Proof. Consider a square submatrix B of A. The proof is by induction in the size of B. When B is a 1×1 matrix, the determinant is simply the single entry of B and the proof is complete. For the inductive step, consider the first column of B. We have three cases:

1. All entries in the first column of B are 0: In this case $\det(B) = 0$ and we are done.

2. Exactly one entry in the first column of B is non-zero: In this case we expand the determinant along the first column using Proposition 1, and we are done by induction.

3. The first column of B contains exactly two non-zero entries, with a 1-entry in row i and a (-1)-entry in row j, say. We now add row i to j and let B' be the resulting matrix. By Proposition 2 we have $\det(B) = \det(B')$. Notice that B' has a single nonzero entry in the first column in row i, and that the matrix B'_i also has the property that each column has at most two non-zero entries, at most one being 1 and at most one being -1. We now expand the determinant of B' along the first column using Proposition 1, and we are again done by induction.

□

From Proposition 2 and Definition 1 we immediately have the following simple consequences:

Lemma 2. *Let A be totally unimodular. Then A^T is totally unimodular. Let B be obtained from A by removing rows or columns, by exchanging rows, by exchanging columns, or by multiplying rows or columns by -1. Then B is also totally unimodular.*

Finally we may build larger totally unimodular matrices by adjoining an identity matrix or by duplicating the matrix:

Lemma 3. *Let A be totally unimodular. Then $\begin{bmatrix} A & I \end{bmatrix}$ and $\begin{bmatrix} A & A \end{bmatrix}$ are totally unimodular.*

Proof. That $\begin{bmatrix} A & I \end{bmatrix}$ is totally unimodular is seen by expanding the determinant of the submatrix under consideration along all the columns originating from the identity matrix using Proposition 1 thereby reducing the determinant computation (up to a sign) to the that of the determinant of a submatrix of A, which is in $\{-1, 0, 1\}$ by assumption.

For $\begin{bmatrix} A & A \end{bmatrix}$, if the submatrix under consideration contains two identical columns, its determinant is 0. Otherwise the columns may be reordered to form a submatrix of A, whose determinant is in $\{-1, 0, 1\}$ by assumption. □

While the above results give a very large class of totally unimodular linear programs, it can still be cumbersome to model directly using such programs. Namely, one would need to check each time that the constraints created satisfy the conditions for total unimodularity given above. This is where the network flow model which we will cover next becomes extremely useful. By using this model we are guaranteed that the corresponding linear program is totally unimodular; furthermore this model gives already the vast majority of the modelling power of totally unimodular linear programs.

2 Networks

A *flow network* or simply a *network* is a directed graph $D = (\mathcal{N}, \mathcal{A})$. A *flow* in a network is an assignment of a real number to each arc, the *flow* on the arc. One may thus think of a flow as a function $x : \mathcal{A} \to \mathbb{R}$. But usually we view the flow as an assignment to variables x_{ij}, for each $ij \in A$.

Various *constraints* can be imposed onto flow networks. We will always have a non-negativity constraint, stating that $x_{ij} \geq 0$ for all $ij \in \mathcal{A}$. The additional constraints we will consider later are so-called node *balance constraints* and arc *capacity constraints.* A flow that satisfies all given constraints is called a *feasible* flow.

A network is often built to model a real-life network and much of the terminology used for flows and networks is borrowed from such real-life networks. An example of such a real-life network is a water distribution network. Here nodes represent water plants supplying water to the network, pipe junction points distributing incoming water flow along a number of pipes to outgoing water flow along other pipes, and finally homes and other consumers of water. Other examples of real-life networks are communication networks, electrical distribution networks, railway networks, etc. Flows in such networks could be data, electrical current, freight, etc. In other circumstances the network may model a completely abstract setting. In some of these cases the flow in these networks could still represent real-life quantities; in other cases the flow could be completely abstract as well.

Consider a fixed flow x in a network $(\mathcal{N}, \mathcal{A})$. For a node $i \in \mathcal{N}$, the *outgoing flow* from i is given by the summation of flow on all outgoing arcs from i, $\sum_{ij \in \mathcal{A}} x_{ij}$. Similarly, the *ingoing flow* from i is given by the summation of flow on all ingoing arcs to i, $\sum_{ji \in \mathcal{A}} x_{ji}$. The *balance* of node i with respect to the flow x is given by the difference of the outgoing flow and the ingoing flow,

$$b_i(x) = \sum_{ij \in \mathcal{A}} x_{ij} - \sum_{ji \in \mathcal{A}} x_{ji} \quad . \tag{8}$$

We say that flow is supplied at a node i if $b_i(x) > 0$, and we call i a *source* node. Similarly we say that flow consumed at a node i if $b_i(x) < 0$, and we call i a *sink* node. If a flow has no sources or sinks we call the flow a *circulation*.

2.1 Node balance constraints

A balance constraint is specified by *balances* b_i for each $i \in \mathcal{N}$ and states that

$$b_i(x) = b_i \tag{9}$$

for all $i \in \mathcal{N}$. Note that it is necessary to have $\sum_{i \in \mathcal{N}} b_i = 0$ in order to have feasible flows, so this will be an assumption we always have for the balance constraint. Namely, if x is a flow we have

$$\sum_{i \in \mathcal{N}} b_i(x) = \sum_{i \in \mathcal{N}} \left(\sum_{ij \in \mathcal{A}} x_{ij} - \sum_{ji \in \mathcal{A}} x_{ji} \right) = \sum_{ij \in \mathcal{A}} x_{ij} - \sum_{ji \in \mathcal{A}} x_{ji} = 0 \quad .$$

2.2 Arc constraints

Arc constraint are specified by *upper bounds* u_{ij} (also called capacities) and *lower bounds* l_{ij} for each $ij \in \mathcal{A}$ and states that

$$l_{ij} \leq x_{ij} \leq u_{ij} \tag{10}$$

for all $ij \in \mathcal{A}$. Note that is is necessary to have $0 \leq l_{ij} \leq u_{ij}$ for all $ij \in \mathcal{A}$ in order to have feasible flows. Sometimes just upper bounds are specified and in that case we implicitly assume $l_{ij} = 0$ for all $ij \in \mathcal{A}$.

3 The minimum cost flow problem

The minimum cost flow problem is easily the most fundamental of all network flow problems. The problem is simply to minimize the *cost* of a flow in a network subject to certain given constraints. The most basic cost model is a linear cost model that assigns a real-valued cost c_{ij} to each arc $ij \in \mathcal{A}$. The cost of an arc represents the cost per unit flow along that arc, and is in real-life networks typically also the actual price of transporting physical units of flow. However the cost c_{ij} assigned to an arc ij is also allowed to be a negative number!

The cost $c(x)$ of a given flow x is given by

$$c(x) = \sum_{ij \in \mathcal{A}} c_{ij} x_{ij} \quad . \tag{11}$$

In the minimum cost flow problem we are always given balance constraints; capacity constraints are optional.

3.1 Integrality theorem

We now establish that the minimum cost flow problem has optimal integer solutions whenever the constraints are integer valued.

Theorem 4. *Let $D = (\mathcal{N}, \mathcal{A})$ be a network with costs given by $c_{ij} \in \mathbb{R}$, balance constraints given by $b_i \in \mathbb{Z}$ for $i \in \mathcal{N}$, and lower and upper bounds given by $l_{ij}, u_{ij} \in \mathbb{Z}$, where $0 \le l_{ij} \le u_{ij}$, for $ij \in \mathcal{A}$. Then there is a minimum cost feasible flow x with integer valued flow on every arc.*

Proof. We can formulate the problem as a linear program as follows:

$$\begin{array}{lrcll} \textit{min} & \displaystyle\sum_{i,j=1}^{n} c_{ij} x_{ij} & & & \\ \textit{s.t.} & \displaystyle\sum_{ij \in \mathcal{A}} x_{ij} - \sum_{ji \in \mathcal{A}} x_{ji} & = & b_i & i \in \mathcal{N} \\ & x_{ij} & \le & u_{ij} & ij \in \mathcal{A} \\ & x_{ij} & \ge & l_{ij} & ij \in \mathcal{A} \end{array} \tag{12}$$

Rewriting the program in standard form we obtain:

$$\begin{array}{lrcll} \textit{max} & \displaystyle -\sum_{i,j=1}^{n} c_{ij} x_{ij} & & & \\ \textit{s.t.} & \displaystyle\sum_{ij \in \mathcal{A}} x_{ij} - \sum_{ji \in \mathcal{A}} x_{ji} & \le & b_i & i \in \mathcal{N} \\ & \displaystyle -\sum_{ij \in \mathcal{A}} x_{ij} + \sum_{ji \in \mathcal{A}} x_{ji} & \le & -b_i & i \in \mathcal{N} \\ & x_{ij} & \le & u_{ij} & ij \in \mathcal{A} \\ & -x_{ij} & \le & -l_{ij} & ij \in \mathcal{A} \\ & x_{ij} & \ge & 0 & ij \in \mathcal{A} \end{array} \tag{13}$$

Let $A = (a_{k,(i,j)})$ denote the node-arc incidence matrix of D. That it, the rows of A are indexed by nodes $k \in \mathcal{N}$, the columns of A are indexed by arcs $ij \in \mathcal{A}$, and entry $(k,(i,j))$ of A is given by

$$a_{k,(i,j)} = \begin{cases} 1 & \text{if } k = i \\ -1 & \text{if } k = j \\ 0 & \text{otherwise} \end{cases} .$$

We may then write the constraints of the linear program (13) in matrix form using A as $\widehat{A}x \leq \widehat{b}, x \geq 0$, where $\widehat{A}$ and $\widehat{b}$ are given by

$$\widehat{A} = \begin{bmatrix} A \\ -A \\ I \\ -I \end{bmatrix}, \qquad \text{and} \qquad \widehat{b} = \begin{bmatrix} b \\ -b \\ u \\ -l \end{bmatrix} .$$

□

By Lemma 1, the matrix A is totally unimodular, and by Lemma 2 and Lemma 3 it follows that $\widehat{A}$ is totally unimodular as well. Since $\widehat{b}$ is integer the result then follows from Theorem 3.

Remark. We proved the theorem above for minimum cost flow problem that had both node balance constraints as well as arc constraints. The proof can be adapted to the case when there are only balance constraints. However, now there is also the possibility that the problem is unbounded, so the statement becomes that whenever optimal solutions exists, an optimal integer solution exists.

4 Modelling with minimum cost flows

In this section we will see a few classical examples of problems modelled as minimum cost flow problems.

4.1 The assignment problem and the transportation problem

Let's first see how to model the assignment problem as a minimum cost flow problem. Let $C = (c_{ij})$ be the given $n \times n$ matrix of costs. We build a network from a complete bipartite directed graph with n vertices on each side. Nodes on the left are given balance 1 and nodes on the right are given balance -1. Finally the costs comes from the cost matrix C. More precisely, we define a network $D = (\mathcal{N}, \mathcal{A})$, where $\mathcal{N} = \{1, 2, \dots, n\} \cup \{1', 2', \dots, n'\}$. We have an arc ij' for all i, j which is given cost c_{ij}. Finally we have a balance constraint given by $b_i = 1$ and $b_{i'} = -1$ for all i.

The transportation problem is a classical generalization of the assignment problem. Here we are to ships goods from sources to destinations at minimum cost. We are given a set of sources $\mathcal{S}$ and a set of destinations $\mathcal{D}$. Each source $i \in \mathcal{S}$ supplies r_i units of good, and each destination $j \in \mathcal{D}$ requires s_j units of good. The cost of shipping a unit of good from source i to destination j is c_{ij}.

This is modelled as a minimum cost flow problem with a network $D = (\mathcal{N}, \mathcal{A})$ where $\mathcal{N} = \mathcal{S} \cup \mathcal{D}$, $\mathcal{A} = \mathcal{S} \times \mathcal{D}$, forming a complete bipartite directed graph between $\mathcal{S}$ and $\mathcal{D}$. The cost on arc ij for $i \in \mathcal{S}$ and $j \in \mathcal{D}$ is simply c_{ij}. Finally $b_i = r_i$ for $i \in \mathcal{S}$ and $b_j = -s_j$ for $j \in \mathcal{D}$.

4.2 Tanker scheduling

The tanker scheduling problem (Dantzig and Fulkerson, 1954) is concerned with finding the minimum size of a fleet of homogeneous tankers needed to meet a prescribed schedule of deliveries between n ports.

The schedule of deliveries is specified be a list of of triples (i^k, j^k, t^k), $k = 1, \dots, m$. A triple (i, j, t) indicates that a tanker must pick up a shipment at time t at port i for delivery at port j. We are furthermore given matrices of positive numbers a_{ij} and b_{ij}, $i, j = 1, \dots, n$, where a_{ij} is time for loading at port i and travelling loaded to port j, and b_{ij} is the time for unloading at port i and travelling unloaded to port j.

We build a network $D = (\mathcal{N}, \mathcal{A})$, where $\mathcal{N} = \{H\} \cup \{L_1, \dots, L_m\} \cup \{U_1, \dots, U_m\}$. The node H represents the home harbour. For each delivery (i^k, j^k, t^k) we have two nodes L_k and U_k representing loading and unloading of delivery k. The arcs of the network are as follows: From H to L_k we have an arc, representing putting a ship on route. From L_k to U_k we have an arc, representing shipping delivery k. From U_k to H we have an arc, representing pulling a ship off route again. Finally, for every pair of deliveries (i^k, j^k, t^k) and (i^ℓ, j^ℓ, t^ℓ) we have an arc between U_k and L_ℓ whenever $t_\ell \geq t_k + a_{i^k j^k} + b_{j^k i^\ell}$, representing that a ship is able to perform delivery ℓ after performing delivery k.

The arcs from H to L_k are given cost 1; all other arcs are given cost 0. The arcs from L_k to U_k are given lower bound 1; all other arcs are given lower bound 0. All arcs are given upper bound 1. Finally the balances of all nodes are 0, meaning that we are looking for a circulation.

4.3 Optimal loading of a hopping airplane

The hopping airplane problem (Gupta, 1985; Lawania, 1990) is concerned with finding the most profitable way to accept passengers along a “hopping flight” route. We consider a single plane with a passenger capacity of p that visits cities $1, 2, \dots, n$ in this order. At each city the plane can pick up passengers or drop off passengers. Let b_{ij} be the number of passengers available for transporting from city i to j and let f_{ij} be the fare paid per passenger transported from city i to city j.

We build a network $D = (\mathcal{N}, \mathcal{A})$ where $\mathcal{N} = \{C_1, \dots, C_n\} \cup \{T_{ij} \mid i < j\}$. The node C_i represents city i and node T_{ij} represents the source of passengers available for transporting from city i to city j. The arcs of the network are as follows: From C_i to C_{i+1} we have an arc, representing transportation of passengers along the leg from city i to city $i + 1$. From T_{ij} we have arcs to C_i and C_j, where the arc from T_{ij} to C_i represents accepting passengers onto the plane and the arc from T_{ij} to C_j represents rejecting passengers onto the plane (and such rejected passengers must then find other means of getting to city j). The arc from T_{ij} to C_i is given cost $-f_{ij}$; all other arcs are given cost 0. The arc from C_i to C_{i+1} is given upper bound p. The arcs from T_{ij} to C_i and C_j are given upper bound [2] b_{ij}. Finally we assign balance b_{ij} to node T_{ij}, and we assign balance $-(\sum_{i<j} b_{ij})$ to node C_j.

5 The maximum (s,t)-flow problem

We now introduce the maximum (s,t)-flow problem. This problem may be viewed as a (important) special case of the minimum cost flow problem, but here we provide its definition in its own right. We are given a flow network $D = (\mathcal{N}, \mathcal{A})$ with specified upper bounds, together with two of the nodes $s, t \in \mathcal{N}$ being singled out. The node s is thought of as a source node and t is thought of as a sink node. The *flow conservation* constraint states that $b_i(x) = 0$ for all $i \in \mathcal{N} \setminus \{s,t\}$.

Thus, a flow x is feasible if it satisfies the non-negativity constraint, the flow conservation constraint, and the arc constraint given by the upper bounds. For a feasible flow we define the *value* as $|x| = b_s(x)$. The maximum (s,t)-flow problem is now to find a feasible flow x maximizing the value $|x|$.

Let us first see how to model the maximum (s,t)-flow problem as a minimum cost flow problem. Given a network $D = (\mathcal{N}, \mathcal{A})$ as above with specified source s and sink t, we add to $\mathcal{A}$ an arc from t to s obtaining a new network D'. We state the minimum cost flow problem on D' by giving the new arc from t to s cost -1 and upper bound ∞. All other arcs are given cost 0. Finally all nodes are given balance 0. It is now easy to see that feasible flows x in D corresponds to feasible flows x' in D' where the cost of x' is the negative of the value of x. Namely $x'_{ij} = x_{ij}$ for $ij \neq ts$, and $x'_{ts} = b_s(x)$.

5.1 The max-flow min-cut theorem

One reason for the importance of the maximum (s,t)-flow problem is the celebrated max-flow min-cut theorem.

An (s,t)-cut in D is a partition (S,T) of the nodes of D, $\mathcal{N} = S \cup T$, such that $s \in S$ and $t \in T$. The *capacity* $u(S,T)$ of a (s,t)-cut (S,T) is given by

$$u(S,T) = \sum_{\substack{ij \in \mathcal{A} \\ i \in S, j \in T}} u_{ij} \ .$$

The minimum capacity (s,t)-cut problem is the associated minimization problem of finding a (s,t)-cut of minimum capacity.

It is not hard to see that for any feasible flow x and any (s,t)-cut (S,T) we have $|x| \leq u(S,T)$. The max-flow min-cut theorem states that we get an equality for optimal solutions to the maximum (s,t)-flow problem and the minimum (s,t)-cut problem.

[2]These constraints are actually not necessary and could be omitted.

Theorem 5 (Max-flow min-cut Theorem). *Let x be a flow of maximum value and let (S,T) be a (s,t)-cut of minimum capacity. Then*

$$|x| = u(S,T) \ .$$

In the course "Algorithms and Datastructures 2" you have previously seen an algorithmic proof a the max-flow min-cut using the Ford-Fulkerson maximum flow algorithm. We shall now see that the max-flow min-cut theorem is essentially a special case of the duality theorem of linear programming. First we formulate the maximum flow problem as a linear programming problem. Let A denote the node-arc incidence matrix of the network D. Let d be the vector indexed by nodes $i \in \mathcal{N}$ given by

$$d_i = \begin{cases} -1 & \text{if } i = s \\ 1 & \text{if } i = t \\ 0 & \text{otherwise} \end{cases} \ .$$

A linear program for the maximum flow problem is then:

$$\begin{array}{lrcl} \mathit{max} & v & & \\ \mathit{s.t.} & Ax + dv & = & 0 \\ & x & \leq & u \\ & x & \geq & 0 \\ & v & \geq & 0 \end{array} \tag{14}$$

The idea behind this formulation is the same as the above reformulation as a minimum cost flow: An artificial arc from t to s has essentially been added to the network. Then v denotes the flow on this edge, which we then maximize. The equality constraints now state that all nodes (including s and t) should have balance 0.

We now make an observation that simplifies the later arguments: For any x and v we have $Ax + dv = 0$ if and only if $Ax + dv \leq 0$. To see this, note that the sum of all coordinates of $Ax + dv$ is 0. Thus, if $Ax + dv \leq 0$ it is not possible to have a strict inequality in any coordinate, and hence $Ax + dv = 0$.

So we consider instead the following linear program, which is in standard form:

$$\begin{array}{lrcl} \mathit{max} & v & & \\ \mathit{s.t.} & Ax + dv & \leq & 0 \\ & x & \leq & u \\ & x & \geq & 0 \\ & v & \geq & 0 \end{array} \tag{15}$$

The dual linear program of this is:

$$\begin{array}{llll} min & \sum_{ij\in\mathcal{A}} u_{ij}z_{ij} & & \\ s.t. & y_t - y_s & \geq 1 & \\ & z_{ij} + y_i - y_j & \geq 0 & ij \in \mathcal{A} \\ & y_i & \geq 0 & i \in \mathcal{N} \\ & z_{ij} & \geq 0 & ij \in \mathcal{A} \end{array} \tag{16}$$

The dual variables y_i are to be thought of as *potentials*. The dual linear program then asks for an assignment of potentials to nodes such that the potential at the sink t is at least one unit higher than the potential at s. The constraints for the arcs state that when potentials increase along an arc a cost must be incurred at the rate of the capacity of the arc.

We now give a proof of the max-flow min-cut theorem using complementary slackness. Let x and (y, z) be a pair of optimal solutions to the primal and the dual linear program. Define $S = \{i \mid y_i \leq y_s\}$ and let $T = \mathcal{N} \setminus S$. The first constraint of the dual ensures that $y_t \geq y_s + 1$, which means $t \notin S$. Thus (S, T) form a valid (s, t)-cut. We now consider the flow on arcs across the cut.

Consider first an arc $ij \in A$ with $i \in S$ and $j \in T$. By definition of S we have $y_i \leq y_s < y_j$. By the dual constraint for arc ij we have $z_{ij} \geq y_j - y_i > 0$. Thus the dual variable z_{ij} is strictly positive which by complementary slackness means that the corresponding primal constraint is satisfied with slack 0. That is, $x_{ij} = u_{ij}$. Consider next an arc $ij \in A$ with $i \in T$ and $j \in S$. By definition of S we have $y_j \leq y_s < y_i$. For the dual constraint for arc ij we now have $z_{ij} + y_i - y_j \geq y_i - y_j > 0$. Thus the dual constraint for arc ij is satisfied with strictly positive slack which by complementary slackness means that the corresponding primal variable is 0. That is, $x_{ij} = 0$. We now conclude that

$$u(S, T) = \sum_{\substack{ij\in\mathcal{A} \\ i\in S, j\in T}} u_{ij} = \sum_{\substack{ij\in\mathcal{A} \\ i\in S, j\in T}} x_{ij} - \sum_{\substack{ij\in\mathcal{A} \\ i\in T, j\in S}} x_{ij} = |x| \ .$$

Another way to arrive at the same result is to start instead with an integer linear program for the minimum cut problem and use totally unimodularity. Such an integer linear program is as follows:

$$
\begin{array}{lrcll}
min & \sum_{ij \in \mathcal{A}} u_{ij} z_{ij} & & & \\
s.t. & y_t - y_s & \geq & 1 & \\
 & z_{ij} + y_i - y_j & \geq & 0 & ij \in \mathcal{A} \\
 & y_i & \leq & 1 & i \in \mathcal{N} \\
 & z_{ij} & \leq & 1 & ij \in \mathcal{A} \\
 & y_i & \geq & 0 & i \in \mathcal{N} \\
 & z_{ij} & \geq & 0 & ij \in \mathcal{A} \\
 & y_i & \in & \mathbb{Z} & i \in \mathcal{N} \\
 & z_{ij} & \in & \mathbb{Z} & ij \in \mathcal{A}
\end{array}
\tag{17}
$$

Here a (s,t)-cut (S,T) is expressed as $S = \{i \in \mathcal{N} \mid y_i = 0\}$ and $T = \{i \in \mathcal{N} \mid y_i = 1\}$. The constraint $z_{ij} + y_i - y_j \geq 0$ ensures that if $y_i = 0$ and $y_j = 1$ then $z_{ij} = 1$, which means that the capacity c_{ij} is added as part of the value of the solution. We may now observe that the upper bounds on y_i and z_{ij} are not important, and one can instead represent a (s,t)-cut as $S = \{i \in \mathcal{N} \mid y_i \leq y_s\}$ and $T = \{i \in \mathcal{N} \mid y_i > y_s\}$. Furthermore we see that by dropping the integrality constraints, the resulting linear program is totally unimodular and this means that it still models the minimum cut problem. Taking the dual we obtain the linear program (15) for the maximum flow problem, and the max-flow min-cut theorem thus follows by the duality theorem for linear programs.

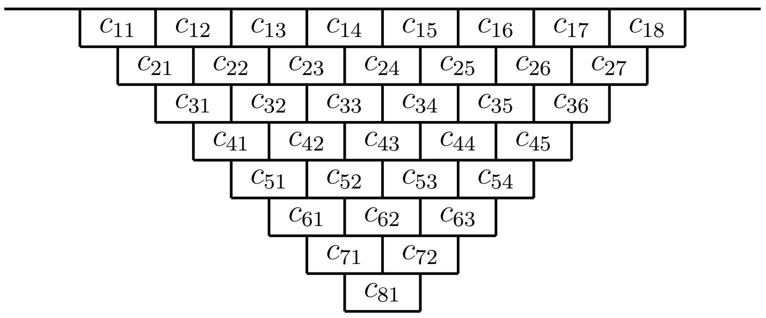
6 Modelling with maximum flows

In this section we will see a few classical examples of problems modelled as the maximum (s,t)-flow problem and the (equivalent) minimum (s,t)-cut problem.

6.1 Preemptive scheduling on uniform parallel machines

The problem of preemptive scheduling on uniform parallel machines (Federgruen and Groenevelt, 1986) is concerned with the scheduling of independent jobs on a number of uniform machines. We have available m uniform machines available onto which we are to schedule n jobs. Each job j is specified by a processor requirement p_j, a release time r_j and a due time d_j. A processor can work on at most a single job at a time, but we allow for preemptions.

We formulate this as a maximum (s,t)-flow problem as follows. First we generate an ordered list of all points in time that is either a release time or a due time, $t_1 < t_2 < \cdots < t_\ell$. We build a network $D = (\mathcal{N}, \mathcal{A})$ where $\mathcal{N} = \{s,t\} \cup \{J_1, \ldots, J_n\} \cup \{I_1, I_2, \ldots, I_{\ell-1}\}$. The node J_j represents job j and the node I_k represents the time interval between t_k and t_{k+1}. The arcs of the network are as follows: From s we have an arc to J_j, representing allocating computation time to job j. From J_j we have an arc to I_k whenever $t_k \geq r_j$ and



c_{11} c_{12} c_{13} c_{14} c_{15} c_{16} c_{17} c_{18}
c_{21} c_{22} c_{23} c_{24} c_{25} c_{26} c_{27}
c_{31} c_{32} c_{33} c_{34} c_{35} c_{36}
c_{41} c_{42} c_{43} c_{44} c_{45}
c_{51} c_{52} c_{53} c_{54}
c_{61} c_{62} c_{63}
c_{71} c_{72}
c_{81}

Figure 1: Regions of an open pit mine with excavation costs.

$t_{k+1} \leq d_j$ representing performing computation of job j in the time interval between t_k and t_{k+1}. From I_k we have an arc to t, representing the computation performed in time interval k. The arc from s to J_j is given upper bound p_j. The arc from J_j to I_k is given upper bound $t_{k+1} - t_k$. Finally is the arc from I_k to t given upper bound $m(t_{k+1} - t_k)$.

6.2 Distributed computing on two-processor systems

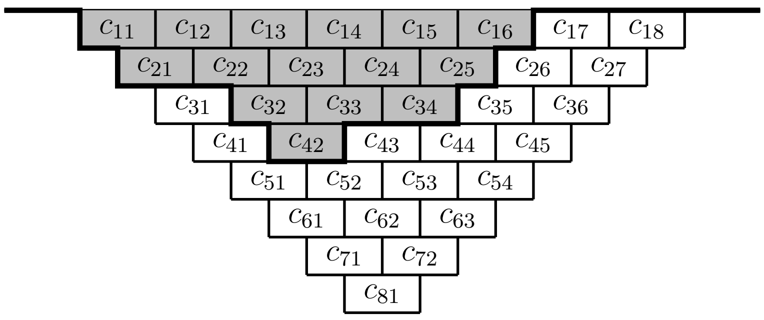
The problem of distributed computing on multi-processor systems (Stone, 1977) is concerned with scheduling tasks onto (not necessarily uniform) machines minimizing the total cost of computation and interprocessor communication. Here we consider the case of a two-processor system. We are given n tasks to distribute onto the two machines. For each task j we have a cost p_j^1 for running the task on processor 1, and a cost p_j^2 for running the task on processor 2. For a pair of tasks i and j we have a communication cost c_{ij} that is incurred if tasks i and j are not scheduled on the same machine.

We formulate this as a minimum (s,t)-cut problem as follows. We build a network $D = (\mathcal{N}, \mathcal{A})$ where $\mathcal{N} = \{s,t\} \cup \{T_1, \ldots, T_n\}$. The node T_j represent task j. The arcs of the network are as follows: From s we have an arc to T_j and from T_j we have an arc to t. Between each pair i and j we have arcs from T_i to T_j and from T_j to T_j. The arc from s to T_j is given upper bound p_j^2. The arc from T_j to t is given upper bound p_j^1. Finally are the arcs between T_i and T_j given upper bound c_{ij}. For a (s,t)-cut (S,T), the nodes $T_j \in S$ are scheduled on processor 1 and the nodes $T_j \in T$ are scheduled on processor 2.

6.3 Open pit mining

The open pit mining problem (Johnson, 1968) is concerned with planning which regions of an open pit mining operation should be excavated in order to maximize the net profit. For a region to be excavated it is necessary that all regions above have already been excavated.

We consider a simplistic 2-dimensional model for planning the excavation contour. We have regions idexeded by (i,j) for $i = 1, 2, \ldots, s$ and $j = 1, 2, \ldots, s - i + 1$. In order to excavate region (i,j) it is necessary to also excavate regions $(i-1,j)$ and region $(i-1,j+1)$. The regions are illustrated for the case of $s = 8$ in Figure 1.



c_{11} c_{12} c_{13} c_{14} c_{15} c_{16} c_{17} c_{18}
c_{21} c_{22} c_{23} c_{24} c_{25} c_{26} c_{27}
c_{31} c_{32} c_{33} c_{34} c_{35} c_{36}
c_{41} c_{42} c_{43} c_{44} c_{45}
c_{51} c_{52} c_{53} c_{54}
c_{61} c_{62} c_{63}
c_{71} c_{72}
c_{81}

Figure 2: A possible excavation of the open pit mine.

Each region (i, j) has a cost c_{ij} for excavation. If excavating a region yields a profit this is indicated by negative costs. A possible excavation is illustrated in Figure 2 that gives a net profit of $c_{11}+c_{12}+c_{13}+c_{14}+c_{15}+c_{16}+c_{21}+c_{22}+c_{23}+c_{24}+c_{25}+c_{32}+c_{33}+c_{34}+c_{42}$.

We formulate this as a minimum (s,t)-cut problem as follows. We build a network $D = (\mathcal{N}, \mathcal{A})$ where $\mathcal{N} = \{s,t\} \cup \{R_{ij} \mid i = 1,2,\ldots,s; j = 1,2,\ldots,s-i+1\}$. Node R_{ij} represent region (i, j). The arcs of the network are as follows: From s we have an arc to R_{ij} if $c_{ij} < 0$. From R_{ij} we have an arc to t if $c_{ij} > 0$. Finally from R_{ij} we have arcs to $R_{i-1,j}$ and $R_{i-1,j+1}$. An arc from s to R_{ij} is given upper bound $-c_{ij}$. An arc from R_{ij} to t is given upper bound c_{ij}. The arcs from R_{ij} to $R_{i-1,j}$ and $R_{i-1,j+1}$ are given upper bound ∞. For a (s,t)-cut (S,T), the nodes $R_{ij} \in S$ are the regions excavated. Note that if $c(S,T) < \infty$ then this is a valid plan of excavation. Also we see then that

$$u(S,T) = \sum_{R_{ij}\in S: c_{ij}>0} c_{ij} - \sum_{R_{ij}\in T: c_{ij}<0} c_{ij} \ ,$$

and hence

$$u(S,T) + \sum_{c_{ij}<0} c_{ij} = \sum_{R_{ij}\in S} c_{ij} \ .$$

7 Solving the minimum cost flow problem

In this section we will consider dedicated algorithms for the minimum cost flow problem. When actually solving an instance of the minimum cost flow problem it can be useful to place certain assumptions on the given instance, in a similar way that we assumed linear programs were in *standard form* when given as input to the simplex algorithm. Two basic assumptions we place on networks are the following:

1. The network is connected when viewed as an undirected graph (meaning that directions on arcs are ignored).

2. The network does not contain any 2-cycles (that is, if ij is an arc, then the arc ji is not present in the network).

The first assumption is without loss of generality: if the network is not connected when viewed as an undirected graph, we may simply consider each of the connected components separately, and solve the minimum cost flow problem for each of them. After this, the solutions for each connected component may simply be combined to a solution for the complete network. The second assumption is made for clarity of presentation. The algorithms we consider work even in the presence of 2-cycles. We leave it as an exercise for the interested reader to work out the details of this. Alternatively, one may transform the network, eliminating 2-cycles by subdividing one of the arcs, introducing an auxiliary node. This process is shown in Figure 3.

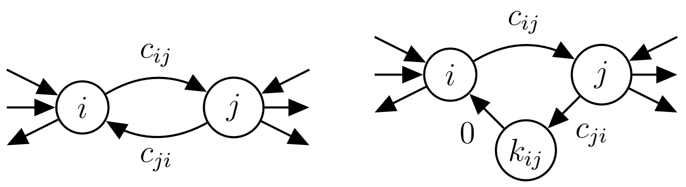


Figure 3: Subdividing an arc of the 2-cycle shown to the left gives the network shown to the right. Only costs are indicated on the arcs. The lower and upper bounds on the arcs jk and k, i are the same as the arc ji.

Chapter 15 of Vanderbei's "Linear Programming" gives an exposition of the *network simplex algorithm*, that assumes that the network is *uncapacitated*, meaning that there are no arc constraints. In the next subsection we shall see how to transform any network into an equivalent uncapacitated network. It should be pointed out, however, that the network simplex algorithm can be extended to work also for networks with arc constraints.

Finally we will present a different algorithm, the cycle cancelling algorithm (also known as Klein's algorithm). Here we present the algorithm in its full generality, allowing arc constraints in the networks.

7.1 Transformation to uncapacitated networks

Transforming a network with arc-constraints to an uncapacitated network can be done by introducing an auxiliary node for each arc, and adjusting balances. Namely we replace an arc ij with lower bound l_{ij}, upper bound u_{ij} and cost c_{ij}, with arcs from i and j to an auxiliary node k_{ij}. The new balance of node i is $b_i - l_{ij}$, the new balance of node j is $b_j + u_{ij}$, and the node k_{ij} is given balance $l_{ij} - u_{ij}$. The cost of the arc (i, k_{ij}) becomes c_{ij} and the cost of the arc (k_{ij}, i) becomes 0. The result is shown in Figure 4.

A flow x in the original network corresponds to the flow x' in the transformed network letting $x'_{i,k_{ij}} = x_{ij} - l_{ij}$ and $x'_{j,k_{ij}} = u_{ij} - x_{ij}$. Conversely a flow x' in the transformed network corresponds to the flow x in the original network letting $x_{ij} = x'_{i,k_{ij}} + l_{ij}$. The costs of the two flows are related by a constant difference, $c(x') = c(x) - \sum_{ij \in \mathcal{A}} l_{ij} c_{ij}$.

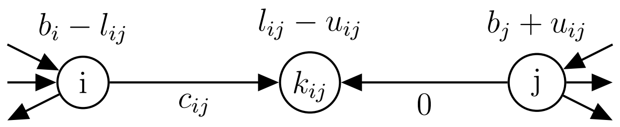


Figure 4: Eliminating arc constraints on the arc ij by introducing an auxiliary node k, and adjusting balances.

7.2 Klein's cycle cancelling algorithm

In this section we give an exposition of Klein's cycle cancelling algorithm for the general case of a network $D = (\mathcal{N}, \mathcal{A})$, given with balances b_i, lower bounds l_{ij}, upper bounds u_{ij}, and costs c_{ij}.

Like the (primal) network simplex algorithm the algorithm works with a feasible flow that is improved successively by pushing flow along cycles. In the network simplex algorithm the current feasible flow was restricted to be *tree solutions* and the search for improving cycles was streamlined by the computation of dual slack variables. A disadvantage to this is that one needs to worry about degenerate updates that can lead to cycling or stalling, although this may be handled by appropriate pivoting rules. The cycle cancellation algorithm instead abandons the restriction to tree solutions and allows *any* cycle in the network to be a candidate for improving the cost of the current flow. If no cycle can be found that improves the cost of the current solution the algorithm terminates and returns the current solution (this should remind the reader of the Ford-Fulkerson algorithm for the maximum flow problem).

In what follows, a *cycle* in D will be a simple cycle in the D when viewed as an undirected graph together with a direction. That is, C is just a sequence of nodes $i_1, i_2, \ldots, i_\ell \in \mathcal{N}$, with $i_\ell = i_1$ but all other i_k's being different, and such that either $(i_k, i_{k+1}) \in \mathcal{A}$ or $(i_{k+1}, i_k) \in \mathcal{A}$, for all $k = 1, \ldots, \ell - 1$. If $(i_k, i_{k+1}) \in \mathcal{A}$ we call (i_k, i_{k+1}) a *forward* edge of C, and if $(i_{k+1}, i_k) \in \mathcal{A}$ we call (i_{k+1}, i_k) a *backwards* edge of C.

Let C be a cycle in D, let F be the set of forward edges of C and let B be the set of backward edges of C. We then define the cost $c(C)$ of the cycle C by

$$c(C) = \sum_{ij \in F} c_{ij} - \sum_{ij \in B} c_{ij} \quad . \tag{18}$$

Given a real number δ, define the *cycle flow* γ_C^δ as

$$\gamma_C^\delta(ij) = \begin{cases} \delta & \text{if } ij \in F \\ -\delta & \text{if } ij \in B \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

We make the following two easy observations.

Observation 2. *Let C be a cycle in D.*

1. *The flow γ_C^δ is a circulation. That is $b_i(\gamma_C^\delta) = 0$ for all $i \in \mathcal{N}$.*

2. The cost of γ_C^δ is given by $c(\gamma_C^\delta) = \delta \cdot c(C)$.

From these we see that if x is a flow satisfying the balance constraints, then $x + \gamma_C^\delta$ is also a flow satisfying the balance constraints and

$$c(x + \gamma_C^\delta) = c(x) + c(\gamma_C^\delta) = c(x) + \delta \cdot c(C) \ . \tag{20}$$

This lead to the important definition of an *augmenting cycle*, which is simply a cycle C for which the cost of feasible flow x can be improved by sending flow along C while maintaining feasibility.

Definition 2. Let x be a feasible flow in D and let C be a cycle in D. We say that C is an *augmenting cycle* relative to x if $c(C) < 0$ and there exists $\delta > 0$ such that $x + \gamma_C^\delta$ is a feasible flow in D.

When C is a cycle such that $c(C) < 0$ then C is an augmenting cycle relative to x if and only if $x_{ij} < u_{ij}$ for all $ij \in F$ and $l_{ij} < x_{ij}$ for all $ij \in B$. When C is an augmenting cycle, the maximum $\delta > 0$ for which $x + \gamma_C^\delta$ remains a feasible flow, which we will denote by $\delta(C)$, is given by

$$\delta(C) = \min \left\{ \min_{ij \in F}(u_{ij} - x_{ij}), \min_{ij \in B}(x_{ij} - l_{ij}) \right\} \ . \tag{21}$$

The basic outline of Klein's algorithm is shown in Figure 5.

```
1: Let $x$ be a feasible flow.
2: **while** exist augmenting cycle $C$ **do**
3:     $x := x + \gamma_C^{\delta(C)}$
4: **end while**
5: **return** $x$
```

Figure 5: Outline of Klein's cycle cancelling algorithm.

The two main components of the algorithm remains to be specified: how to find the first feasible flow, and how to find an augmenting cycle if one exists. Finally we must argue that the algorithm correctly solves the minimum cost flow problem.

7.2.1 Finding the first feasible flow

There are several ways to deal with the problem of finding the first feasible flow. Our choice here will be to formulate the problem as a maximum flow problem, which can then be solved by either the Ford-Fulkerson or Edmonds-Karp algorithms known from the "Algorithms and Datastructures 2" course.

Given a flow network $D = (\mathcal{N}, \mathcal{A})$ we build a new flow network $D' = (\mathcal{N}', \mathcal{A}')$, where $\mathcal{N}' = \{s, t\} \cup \mathcal{N} \cup \{k_{ij}^\ell \mid ij \in \mathcal{A}, \ell = 1, 2\}$. The arcs $\mathcal{A}'$ of the network D' corresponds to nodes and arcs of D. The arcs in D' corresponding to nodes of D are as follows. From s we have an arc to each $i \in \mathcal{N}$ for which $b_i > 0$, and to t we have an arc from each $i \in \mathcal{N}$

for which $b_i < 0$. The arcs in D' corresponding to arcs $ij \in \mathcal{A}$ are as follows. First we have arcs forming a path from i to j through the nodes k_{ij}^1 and k_{ij}^2, and we have also an arc from s to k_{ij}^2 and an arc from k_{ij}^1 to t.

An arc from s to i is given upper bound b_i, and an arc from b_i to t is given upper bound $-b_i$. The arcs from i to k_{ij}^1 and from k_{ij}^2 to j are given upper bounds u_{ij}. The arcs from s to k_{ij}^2 and from k_{ij}^1 to t are given upper bound l_{ij}. The arcs of D' corresponding to an arc $ij \in \mathcal{A}$ together with their given upper bounds are shown in Figure 6.

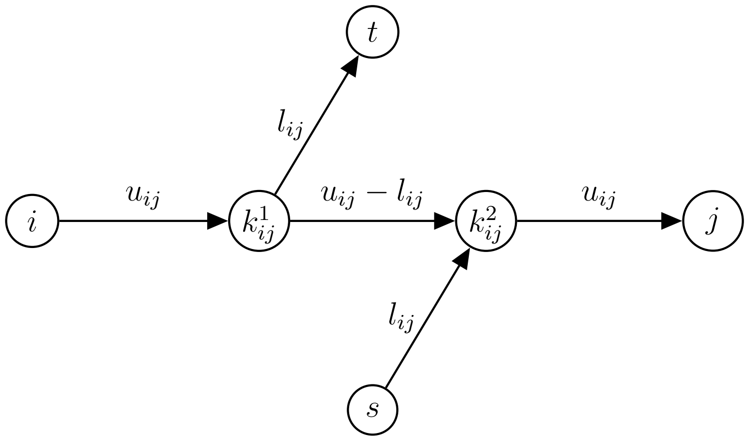


Figure 6: The part of D' corresponding to the arc ij of D.

We now claim that D has a feasible flow if and only if D' has a (s,t)-flow x' that saturates all arcs from the source s, which means $|x'| = \sum_{\substack{i\in\mathcal{N}\\ b_i>0}} b_i + \sum_{ij\in\mathcal{A}} l_{ij}$. In that case a feasible flow x in D is given by $x_{ij} = x'_{i,k_{ij}^1}$. We leave the verification of this claim as an exercise.

7.2.2 Finding an augmenting cycle

In this section we show how to reduce the problem of finding an augmenting cycle to another classical graph problem: finding a negative weight directed cycle in a weighted directed graph. For this we will make use of the notion of the *residual network* also known from max-flow algorithms such as the Ford-Fulkerson and Edmonds-Karp algorithms. Let $D = (\mathcal{N}, \mathcal{A})$ be a network with balances b_i, lower bounds l_{ij}, upper bounds u_{ij}, and costs c_{ij}, and let x be a feasible flow in D. The residual network relative to x is the network $D_x = (\mathcal{N}, \mathcal{A}_x)$ on the same set of nodes as D but with the set of arcs that indicate how the flow x may be changed maintaining feasibility. The arcs $\mathcal{A}_x$ of D_x are given by:

$$\mathcal{A}_x = \{ij \mid ij \in \mathcal{A} \;\&\; x_{ij} < u_{ij}\} \cup \{ji \mid ij \in \mathcal{A} \;\&\; l_{ij} < x_{ij}\} \tag{22}$$

Thus each arc $ij \in \mathcal{A}$ of D can give rise to 0, 1, or 2 arcs of D_x between the nodes i and j. When $x_{ij} < u_{ij}$ the arc $ij \in \mathcal{A}_x$ is given upper bound $(u_x)_{ij} = u_{ij} - x_{ij}$, specifying how much the flow on arc ij in D can be increased, and it is given cost $(c_x)_{ij} = c_{ij}$. When

$l_{ij} < x_{ij}$ the arc $ji \in \mathcal{A}_x$ is given upper bound $(u_x)_{ji} = x_{ij} - l_{ij}$, specifying how much the flow on arc ij in D can be decreased, and it is given cost $(c_x)_{ji} = -c_{ij}$. Traditionally these upper bounds are called *residual capacities*. The balances of all nodes of D_x are set to 0. Likewise are all arcs of D_x given lower bound 0.

Let x and y be feasible flows in D, and note that the flow $z = y - x$ in D is a circulation. We define the corresponding flow $\widetilde{z}$ in D_x by giving positive flow according to the following rules:

- If $z_{ij} > 0$ we let $\widetilde{z}_{ij} = z_{ij}$. Note that $z_{ij} > 0$ implies $x_{ij} < y_{ij} \leq u_{ij}$ and hence is the arc ij present in D_x.
- If $z_{ij} < 0$ we let $\widetilde{z}_{ji} = -z_{ij}$. Note that $z_{ij} < 0$ implies $x_{ij} > y_{ij} \geq l_{ij}$ and hence is the arc ji present in D_x.

Arcs of D_x not assigned flow by these rules are given flow 0. We now have the following simple lemma.

Lemma 4. *$\widetilde{z}$ is a feasible flow in D_x and $c_x(\widetilde{z}) = c(z) = c(y) - c(x)$.*

We can of course do the same in the opposite direction. Let x be a feasible flow in D, and let $\widetilde{z}$ be a feasible flow in D_x. Define the flow z in D by the following rules specifying the flow z_{ij} on arc $ij \in \mathcal{A}$.

- If $ij \in \mathcal{A}_x$ and $ji \in \mathcal{A}_x$, let $z_{ij} = \widetilde{z}_{ij} - \widetilde{z}_{ji}$.
- If $ij \in \mathcal{A}_x$ and $ji \notin \mathcal{A}_x$, let $z_{ij} = \widetilde{z}_{ij}$.
- If $ij \notin \mathcal{A}_x$ and $ji \in \mathcal{A}_x$, let $z_{ij} = -\widetilde{z}_{ji}$.

Let $y = x + z$. We now have the following simple lemma.

Lemma 5. *y is a feasible flow in D and $c(y) = c(x) + c(z) = c(x) + c_x(\widetilde{z})$.*

While the conversion between flows in the original network D and the residual network D_x is a bit cumbersome, we can now see the usefulness of the residual network. Namely, an augmenting cycle C in D relative to a feasible flow x is a directed cycle in D_x, and we have $c(C) = c_x(C)$ and $\delta(C) = \delta_x(C)$, where $\delta_x(C) = \min_{ij \in C}(u_x)_{ij}$. Thus the problem of finding an augmenting cycle in D relative to x is precisely the problem of finding a negative weight directed cycle in D_x when using the costs as weights. We leave it as an exercise how to use either the Bellman-Ford or the Floyd-Warshall shortest path algorithms as seen in the course "Algorithms and Datastructures 2" for this task.

7.2.3 Partial correctness

We now show that the algorithm return a minimum cost feasible flow whenever it terminates. For this we shall use the following lemma which is also of general interest.

Lemma 6 (circulation decomposition lemma). *Let x be a non-negative circulation flow in a network $D = (\mathcal{N}, \mathcal{A})$ (for instance, D may be a residual network). Then there exist cycles $C_1, \ldots, C_k$ and non-negative reals $\delta_1, \ldots, \delta_k \geq 0$ such that*

$$x = \gamma_{C_1}^{\delta_1} + \gamma_{C_2}^{\delta_2} + \cdots + \gamma_{C_k}^{\delta_k} \ .$$

Proof. The proof is an induction in the number of arcs $ij \in \mathcal{A}$ for which $x_{ij} > 0$. In the base case, x is the all-zero flow and the statement is vacuously true. So assume that x is not the all-zero flow. Thus there exists $(i_0, i_1) \in \mathcal{A}$ such that $x_{i_0,i_1} > 0$. Since x is a circulation flow, $b_{i_1}(x) = 0$ and hence there must be an arc $(i_1, i_2) \in \mathcal{A}$ such that $x_{i_1,i_2} > 0$. We continue this process, finding a sequence of nodes $i_0, i_1, i_2, \ldots$. We stop the process when we encounter a previously seen node, that is at the minimum ℓ such that $i_\ell = i_{\ell'}$ for some $\ell' < \ell$. This forms a directed cycle $C_1 = (i_{\ell'}, i_{\ell'+1}, \ldots, i_\ell)$ in D. Let $\delta_1 = \min(x_{i_h,i_{h+1}})$ for $h = \ell', \ldots, \ell - 1$, and consider $x' = x - \gamma_C^\delta$. By construction $x' \geq 0$ is a non-negative circulation flow in D, which has at least one additional arc with a flow of 0 than x. By induction we may write $x' = \gamma_{C_2}^{\delta_2} + \cdots + \gamma_{C_k}^{\delta_k}$ and since $x = \gamma_C^\delta + x'$ this completes the proof. □

We can now establish the partial correctness of Klein's cycle cancelling algorithm.

Lemma 7. *Let D be a network with a feasible flow x. If x is not a minimum cost flow, then there exist an augmenting cycle.*

Proof. Assume that x is a feasible flow in D that is not a minimum cost flow, and let y be any minimum cost feasible flow in D. By Lemma 4 there exist a feasible flow $\widetilde{z}$ in D_x corresponding to $z = y - x$ for which $c(\widetilde{z}) = c(y) - c(x) < 0$. Now by lemma 6 we can write

$$\widetilde{z} = \gamma_{C_1}^{\delta_1} + \gamma_{C_2}^{\delta_2} + \cdots + \gamma_{C_k}^{\delta_k} \ ,$$

and we have

$$c_x(\widetilde{z}) = c_x(\gamma_{C_1}^{\delta_1}) + c_x(\gamma_{C_2}^{\delta_2}) + \cdots + c_x(\gamma_{C_k}^{\delta_k}) < 0 \ .$$

Thus there must be ℓ such that $c_x(\gamma_{C_\ell}^{\delta_\ell}) < 0$, which means that C_ℓ is an augmenting cycle. □

7.2.4 Termination

As in the case of Ford-Fulkerson, we shall only prove termination for the case of integer inputs. Indeed, for general reals, Klein's cycle cancelling algorithm may not terminate unless care is taken on how to choose the augmenting cycles.

If all balances, lower bounds, and upper bounds are integers and an implementation of Ford-Fulkerson is used to find the first feasible flow (which is therefore integer valued), we note that all flows along all arcs and all residual capacities will remain integer valued throughout the algorithm. This means that for each arc ij there is only a finite number of possibilities for the value of the flow as it will be an integer between l_{ij} and u_{ij}. Thus, there are also only finitely many possibilities for the entire feasible flow. As the cost of the feasible flow decreases in each iteration of the while loop, we will not see at particular flow more than once during the executing of the algorithm. Thus, the algorithm terminates.

7.2.5 Complexity

The dominating task of each iteration of Klein's cycle cancelling algorithm is the task of finding an augmenting cycle. As mentioned earlier this can be one efficiently using either the Bellman-Ford or the Floyd-Warshall shortest path algorithms.

What remains is to determine how many iterations the algorithm may take before terminating. As stated above, in the case of non-integer inputs, we are not certain of termination and even in the case of integer input, the algorithm may take exponential time. In this way, the algorithm is very similar to the Ford-Fulkerson method for the max flow problem.

As in that case, it *is* possible to obtain a polynomial time algorithm for min cost flow by specifying more precisely *which* augmenting cycle to pick in case there is more one. The resulting analysis is somewhat more complicated than the analysis of the analogous Edmonds-Karp algorithm for maximum flow and is therefore not included in these notes. In practice, Klein's cycle cancelling algorithm tend to terminate in fairly few iterations, even if no special care is taken about how to choose the augmenting cycle.