Winfried Hochstättler
Alexander Schliep

CATBox – An Interactive Course in Combinatorial Optimization

SPIN XXX

Monograph – Mathematics –

March 4, 2011

Springer
Berlin Heidelberg New York
Barcelona Hong Kong
London Milan Paris
Tokyo

Preface

As naturally as static pictures were used to communicate mathematics from the very beginning, the advent of algorithmic and computational mathematics—and the availability of graphical workstations—introduced us to dynamic displays of mathematics at work. These animations of algorithms looked in the case of graph algorithms quite similar to what you see in CATBox. In fact there has been a substantial literature on the topic and software systems implementing such animations is obtainable from various sources. Nevertheless, these systems have not found very widespread use, with few notable exceptions.

Indeed, this incarnation of CATBox, both concept and name are due to Achim Bachem, was motivated by similar experiences with two prior projects under the same name. The difficulty in using these systems in teaching was that the animations were decoupled from algorithmic code or simply displayed pseudo code, with a hidden algorithm implementation doing its magic behind the scenes. More importantly, the algorithm animation system was almost always a separate entity from the textbook and they were developed respectively written by different people.

This lead to the idea of constructing a system where the student can control the flow of the algorithm in a debugger like fashion and what she or he sees is the actual code running, developed hand-in-hand with a course book explaining the ideas and the mathematics behind the algorithms. Our concept of course influenced our choices in problems and algorithms introduced.

We cover classical polynomial-time methods on graphs and networks from combinatorial optimization, suitable for a course for (advanced) undergraduate or graduate students. In most of the methods considered, linear programming duality plays a key role, sometimes more, sometimes less explicitly. In our belief some of the algorithmic concepts, in particular minimum-cost flow and weighted matching, cannot be fully understood without knowledge of linear programming duality. We decided to already present the simplest example from that point of view and interpret Kruskal's greedy algorithm to compute a minimum spanning tree as a primal-dual algorithm. For that purpose we have to introduce the basic ideas of polyhedral combinatorics quite early. This might be tedious for an audience mostly interested in shortest paths and maximum flows.

Material from this book has been used in full semester graduate courses and for sections of graduate courses and seminars by colleagues and us. For use in an undergraduate course, we suggest to skip the linear programming related chapters 4, 7 and 9 possibly augmenting them with material from other sources.

The animation system Gato is open source, licensed freely under the GNU Lesser General Public License (LGPL). We will collect algorithms which did not make it into the book and community contributions under a compatible license, to support teaching a wider range of graph algorithms in the future.

Acknowledgments

Many people have contributed at various stages to the book and software and were essential for the success. In particular we would like to thank our mentors and collaborators from the Center for Parallel Computing (ZPR) at the University of Cologne, Achim Bachem, Sándor Fekete, Christoph Moll and Rainer Schrader for their motivation and many helpful discussions. We would also like to thank Martin Vingron at the Max Planck institute for Molecular Genetics for support. Many thanks to Günther Rote for helpful suggestions and finding bugs in the manuscript and algorithm implementations. Benjamin Georgi, Ivan G. Costa, Wasinee Rungsarityotin, Ruben Schilling, Stephan Dominique Andres read the manuscript and helped to iron out the rough spots and reduce the number of errors.

The software Gato benefitted from experiences with a prior implementation provided by Bernd Stevens and Stefan Kromberg. Ramazan Buzdemir, Achim Gädke and Janne Grunau contributed greatly to Gato; Heidrun Krimmel provided the screen design. Torsten Pattberg provided implementations of many algorithm animations and helped to increase coherence between text and implementations. Many, many thanks for their great work and the fun time working together.

Many beta testers have been using CATBox and helped to find bugs and inconsistencies. In particular we would like to thank Ulrich Blasum, Mona K. Gavin, Dagmar Groth, Günther Rote, Wolfgang Lindner, Andrés Becerra Sandoval, Werner Nemecek, Philippe Fortemps, Martin Gruber, and Gerd Walther.

Last but not least, many thanks to Martin Peters, our editor at Springer and very likely the most patient man on earth, for his consistent support.

Hagen, Germany *Winfried Hochstättler*
Piscataway, NJ, USA *Alexander Schliep*
August 2009

Table of Contents

6 Maximal Flows

6.1 Introduction

The last chapter ended with a remark that one might consider the shortest path problem as the task to send one unit of "flow" from s to t through a network at minimum cost. In more realistic problems from transportation we usually have to deal with capacity constraints, limiting the amount of flow across arcs. Hence, the basic problem in a capacitated network is to send as much flow as possible between two designated vertices, more precisely from a source vertex to a sink vertex. This notion of a flow in a capacitated network is made more precise in the following.

Definition 16. *Let $D = (V, A, cap)$ be a directed network with a capacity function $cap\colon A \to \mathbb{R}_+$ on the arcs. We call this a* capacitated network. *An s-t-*flow *$f\colon A \to \mathbb{R}_+$ for two designated vertices $s, t \in V$ is a function satisfying*

Flow conservation: *the flow out of each vertex $v \in V \setminus \{s,t\}$ equals the flow into that vertex, that is:*

$$\sum_{\substack{w \in V \\ (v,w)\in A}} f(v,w) = \sum_{\substack{w \in V \\ (w,v)\in A}} f(w,v).$$

Capacity constraints: *The flow on each arc $(u,v) \in A$ respects the capacity constraints. That is,*

$$0 \le f(u,v) \le cap(u,v).$$

We define the *net flow from s* to be

$$|f| := \sum_{\substack{w \in V \\ (s,w)\in A}} f(s,w) - \sum_{\substack{w \in V \\ (w,s)\in A}} f(w,s).$$

The flow conservation rules immediately imply

$$\forall v \in V \setminus \{s,t\}: \sum_{\substack{w \in V \\ (v,w)\in A}} f(v,w) - \sum_{\substack{w \in V \\ (w,v)\in A}} f(w,v) = 0.$$

Adding all these equations to the net flow from s yields

$$
\begin{aligned}
|f| &:= \sum_{\substack{w \in V \\ (s,w) \in A}} f(s,w) - \sum_{\substack{w \in V \\ (w,s) \in A}} f(w,s) \\
&= \sum_{\substack{w \in V \setminus \{t\} \\ (v,w) \in A}} f(u,v) - \sum_{\substack{w \in V \setminus \{t\} \\ (w,v) \in A}} f(t,w) \\
&= \sum_{\substack{w \in V \\ (w,t) \in A}} f(w,t) - \sum_{\substack{w \in V \\ (t,w) \in A}} f(t,w),
\end{aligned}
$$

thus the *net flow from* s equals the net flow into t. We can state our problem now as:

Problem 8. Let $G = (V, A, cap)$ be a capacitated network, $s, t \in V$. Find an s-t-flow which maximizes the net flow from s.

It will make our discussion a bit easier if we restrict the capacity function to integer values for a while, thus we get:

Problem 9. Let $G = (V, A, cap)$ be a capacitated network with an integer capacity function $cap : A \to \mathbb{Z}_+$ and $s, t \in V$. Find an s-t-flow, maximizing the net flow from s.

6.2 The Algorithm of Ford and Fulkerson

Considering shortest path problems we realized that we might consider paths as flows of unit value. The first algorithm for the maximum flow problem that we will introduce, successively finds paths from s to t. It takes the already existing flow into account and augments it by sending flow along that path. Clearly, a flow reduces the remaining capacity. A priori, it is not clear that we can send flow freely without making any mistakes.

Software Exercise 41. Start the software and open the graph `FordFulkerson4.cat`. If you point on the edges you can check their capacity in the info line at the bottom of the window. Vertices s and t in these examples are always those with smallest respectively largest number. The first idea might be to send flow on the three edges that make up a path from 1 to 8. The bottlenecks of the capacity of these edges are the first and the last edge with a capacity of 150. Assume we send 150 units of flow over this path. If we just subtract these numbers from the capacities the graph becomes disconnected and there no longer exists a path from s to t. Nevertheless, this flow is not maximal. We can achieve a larger flow by sending 141 units of flow on each of the paths 1,3,4,7,8 and 1,2,5,6,8 and 9 units right through the middle.

This problem of not being able to revert incorrect previous choices is overcome by the idea of the backward arc. When sending flow through the network we add an artificial backward arc for each arc of nonzero flow, indicating that this flow "can be sent back", thus allowing to reduce the flow on the corresponding forward arc. Formally we get

Definition 17. *Let $D = (V, A, cap)$ be a capacitated network and $f : A \to \mathbb{R}_+$ an s-t-flow. The* residual network *$RN(D, f)$ with respect to f is the network $(V, \tilde{A}, rescap)$. For an arc $a \in A$ we denote by $-a$ a copy of a where head and tail have been interchanged and we write $-A := \{-a \mid a \in A\}$ for all such arcs. The residual capacity we define as*

$$rescap(a) = \begin{cases} cap(a) - f(a) & \textit{for } a \in A \\ f(-a) & \textit{for } a \in -A \end{cases}$$

and finally $\tilde{A} := \{\tilde{a} \in A \cup -A \mid rescap(a) > 0\}$.

With this definition it is easy to design our first algorithm. We iteratively search for a path P in the residual network and send as much flow as possible through this path. For $P \subseteq A$ let $\chi(P) \in \{0, \pm 1\}^A$ denote the *signed characteristic function* of P, i.e. the vector where we have a 1 in the indices of forward arcs from A in P, a -1 in the A-index if the corresponding arc in $-A$ is used by the P and zeroes elsewhere, that is

$$\chi(P)_a := \begin{cases} 1 & \text{if } a \text{ is a forward arc of } P \\ -1 & \text{if } a \text{ is a backward arc of } P \\ 0 & \text{if } a \text{ is a not an arc of } P \end{cases} \tag{6.1}$$

In the algorithm we search for an s-t path in the residual network, compute its bottleneck `delta` and update the flow $f = f + \texttt{delta}\,\chi(P)$. The latter is done by recursively backtracking the path in a for loop and computing `flow[(u,v)]=flow[(u,v)]` $\pm$ `delta` where the sign is chosen according to whether the current arc is an original arc of the network, a `ForwardEdge`, or an artificial backward arc.

In our implementation we compute the path by a BFS in `Shortest-Path(s,t)` and find a path using as few edges as possible. We will see later on that this is a clever choice, but for the following discussion we allow arbitrary paths.

ALGORITHM `FordFulkerson`

```
def UpdateFlow(Path):
    delta = MinResCap(Path)
    for (u,v) in Edges(Path):
        if ForwardEdge(u,v):
            flow[(u,v)] = flow[(u,v)] + delta
        else:
            flow[(v,u)] = flow[(v,u)] - delta

s = PickSource()
t = PickSink()

while not maximal:
    Path = ShortestPath(s,t)
    if Path:
        UpdateFlow(Path)
    else:
        maximal = true

ShowCut(s)
```

Software Exercise 42. Continuing our last experiment we recommend you to choose the "Window Layout" with "Two graph windows". If you start the software it will indicate the capacity of the arcs by their thickness in the upper window. If you choose "1" to be the source and "8" to be the sink, the algorithm will stop at the first breakpoint where the s-t-path using 3 edges has been found by BFS. Here the vertices not on the path are displayed in grey, if they have been processed in the BFS, blue if they are visited and green if they have not been put into the queue. If we now "Trace" into the "UpdateFlow"-Procedure, first we find the last edge to be the bottleneck and update the flow backtracking along the path accordingly. Simultaneously with the flow, we update the residual network, i.e. we introduce backward arcs and update the capacity if necessary.

The first and the last arc on the path disappear since their capacity has been completely used. If you point on the forward respectively backward edge of the middle arc you will find that the capacity of the forward arc has been reduced by 150 which is exactly the capacity of the backward arc.

The second path we find uses the backward edge and UpdateFlow "repairs" the mistake we made in the first step. The flow on the edge $(3,6)$ is reduced by the capacity of the bottleneck $(7,8)$ on the path, i.e. by 141 and we end with the flow described in our previous discussion.

After the update there is no longer any directed s-t-path in the residual network, which is proven by an s-t-cut (to be defined immediately) in the residual network that contains backward arcs only.

6.3 Max-flow-Min-cut

We have a simple argument that this algorithm will terminate as long as the capacities are integers. With every path augmentation the flow is incremented by at least one and there are natural upper bounds on the maximal value of the flow, which are called *cuts*.

Definition 18. *Let $D = (V,A)$ be a digraph, $s,t \in V$ and $\emptyset \neq S \subset V$. The* cut $[S, V \setminus S]$ induced by S *consists of those edges that have one end in each of the two non-empty sets of the partition $S, V \setminus S$:*

$$[S,V\setminus S] := \{(i,j) \in A \mid \{i,j\} \not\subseteq S \text{ and } \{i,j\} \not\subseteq V\setminus S\}.$$

The forward edges $(S, V \setminus S)$ *of a cut are defined as*

$$(S,V\setminus S) := \{(i,j) \in A \mid i \in S, j \in V\setminus S\}.$$

The backward arcs *of a cut $[S,V\setminus S]$ are denoted as $(V\setminus S, S)$. A cut $[S,V\setminus S]$ is an s-t-*cut, *if $s \in S$ and $t \notin S$.*

If $cap\colon A \to \mathbb{R}_+$ is a capacity function, the capacity $cap[S]$ of an s-t-cut *is defined as $cap[S] := cap(S,V\setminus S) := \sum_{a\in(S,V\setminus S)} cap(a)$, the total capacity of the forward edges.*

The capacity of any cut is an upper bound on the flow value:

Lemma 12 (weak duality). *Let $D = (V, A, cap)$ be a capacitated network, f a flow and $[S, V \setminus S]$ an s-t-cut. Then $|f| \leq cap[S]$.*

At the end of the proof we use the abbreviation $f(\tilde{E}) := \sum_{e \in \tilde{E}} f(e)$ if $\tilde{E} \subseteq E$ is a set of edges (c.f. Theorem 9).

Proof.

$$\begin{aligned}
|f| &= \sum_{\substack{w \in V \\ (s,w) \in A}} f(s,w) - \sum_{\substack{w \in V \\ (w,s) \in A}} f(w,s) \\
&= \sum_{\substack{w \in V \\ (s,w) \in A}} f(s,w) - \sum_{\substack{w \in V \\ (w,s) \in A}} f(w,s) \\
&\quad + \sum_{v \in S \setminus s} \left(\sum_{\substack{w \in V \\ (v,w) \in A}} f(v,w) - \sum_{\substack{w \in V \\ (w,v) \in A}} f(w,v) \right) \\
&= \sum_{v \in S} \left(\sum_{\substack{w \in V \\ (v,w) \in A}} f(v,w) - \sum_{\substack{w \in V \\ (w,v) \in A}} f(w,v) \right) \\
&= f((S, V \setminus S)) - f((V \setminus S, S)) \\
&\leq f((S, V \setminus S)) \\
&\leq cap[S].
\end{aligned}$$

□

This proves finiteness of the above procedure if the capacities are integers. Furthermore, when the algorithm terminates source s and destination t are separated in the residual network $RN(D, f)$. Thus if we set

$$S^* := \{v \in V \mid \text{ there is some } s\text{-}v\text{-path in } RN(D, f)\},$$

then $[S^*, V \setminus S^*]$ is an s-t-cut. We compute $|f| = f((S^*, V \setminus S^*)) - f((V \setminus S^*, S^*))$. Since there is no edge emanating from S in $RN(D, f)$ there cannot be any nonzero flow on backward edges of the cut and $f((V \setminus S^*, S^*)) = 0$. Thus $[S^*, V^*]$ is a cut whose capacity equals the value of the flow. Lemma 12 now implies.

Theorem 15 (Max-flow-Min-cut). *Let $D = (V, A, cap)$ be a capacitated network and $s, t \in V$. Then*

$$\max_{f \text{ is } s\text{-}t\text{-flow}} |f| = \min_{[S, V \setminus S] \text{ is } s\text{-}t\text{-cut}} cap[S].$$

If, furthermore, the capacities are integer then a maximal flow can be chosen to be integer on all edges.

Exercise 43. Let $G(V,E)$ be a graph and $s,t \in V$ two vertices.

(i) Prove that the number of pairwise edge disjoint s-t-paths equals the minimal number of edges that must be removed in order to destroy all s-t-paths.
(ii) Prove that the number of pairwise internally vertex disjoint s-t-paths equals the minimal number of vertices that must be removed in order to destroy all s-t-paths.

6.4 Pathologies

In this section we will learn that the augmenting path algorithm with a clumsy implementation can have an exponential running time behavior. Before we do so, we will present an example with irrational data, where the algorithm not only does not terminate but also converges towards a non-optimal value.

Example 4. Consider the network in Figure 6.1. The most important edges are the three innermost edges, the upper and the lower run from left to right with capacities of 1 respectively $\sqrt{2}$. The edge in the middle runs backwards and has an infinite capacity as well as all other edges except of the direct arc (s,t) which has a capacity of 1. The latter edge—as well as several backward arcs—will be omitted in the following pictures since it is not used until the very end. After the first four indicated augmenting steps we get into a situation where the capacities of all arcs—except for the direct arc—are a scalar multiple of the data in the beginning. Clearly an infinite repetition of these steps will thus converge towards a flow of value $1+\sqrt{2}$, while a flow of value $2+\sqrt{2}$ exists.

Even with integer data a bad implementation of the above procedure may result in an exponential run time behavior.

Example 5. Consider the network depicted in Figure 6.2. If each augmentation uses the "bottleneck edge", clearly we will need 2^{k+1} augmentations in order to compute a maximal flow. The size of the data though is dominated by the coding length of the numbers and is $O(k)$. Thus the number of augmentations is exponential in the size of the data.

Software Exercise 44. The interested reader may ask why we did not animate these examples with our software. While this is quite obvious for the example with irrational data, you may try it yourself. Load the file `FordFulkersonWC.cat` and run `FordFulkerson.alg` on it. Isn't it disappointing, it terminates in two iterations. We consider it difficult to find some reasonable implementation that would chose the augmenting paths as in the above examples. This does not mean that there is no problem. The difficulties may occur in larger examples less obviously and hidden somewhere, but actually not with the implementation in our package. The reason will become clear in the next section.

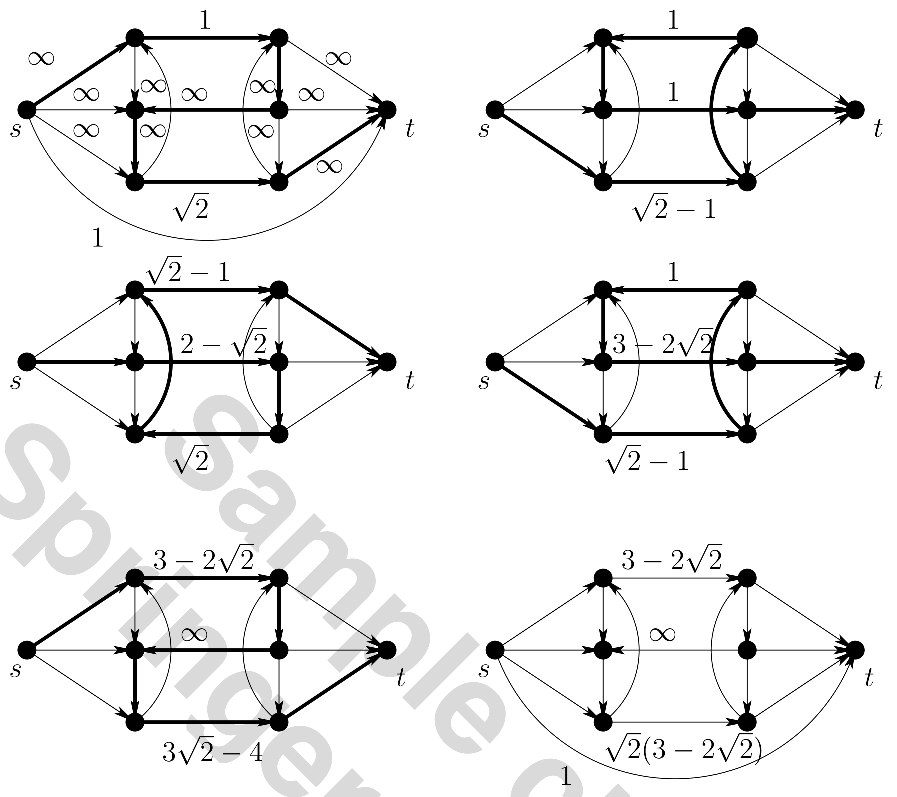

Fig. 6.1. Pathological irrational

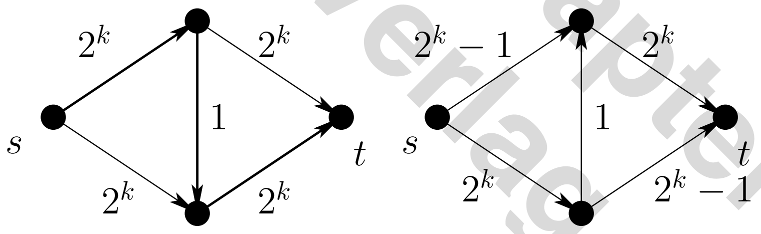

Fig. 6.2. A smallest worst case example

6.5 Edmonds-Karp Implementation

The bad example in the last section suggests two possible approaches to designing a good implementation. Either by shortest paths, i.e., paths using as few edges as possible, or by paths of maximal capacity. The latter can be implemented, but is "unlikely to be efficient" [1].

The former is quite natural and it is the way the augmenting paths are chosen in our implementation of Ford and Fulkerson's algorithm. We run a BFS in the residual network, starting at s until we reach t, i.e. we search for a path from s to t that uses as few edges as possible. The advantage of this implementation is, that we have additional control over the flow of the algorithm as the length of the shortest paths from any vertex v to t must be monotonically increasing:

Lemma 13. *Let $D = (V, A, cap)$ be a capacitated network (cap not necessarily rational), f a flow, P a directed s-t-path in $RN(D, f)$ using as few edges as possible and $\widetilde{f}$ the flow that arises from f by augmentation along P, $\widetilde{f} = f + \Delta\chi(P)$. Let $dist(v)$ denote the distance from v to t in $RN(D, f)$, with respect to to unit weights, that is the number of edges, and $\widetilde{dist}(v)$ the same in v in $RN(D, \widetilde{f})$, then*

$$\forall v \in V : \widetilde{dist}(v) \geq dist(v).$$

Proof. $RN(D, f)$ and $RN(D, \widetilde{f})$ differ only along P. Edges of that path which had the residual capacity of the bottleneck of this path will disappear. Possibly we will be facing newly introduced backward arcs for edges that had no flow in f. Let (i, j) be such an edge. Since (i, j) belongs to a shortest s-t-path in $RN(D, f)$ we necessarily have $dist(i) = dist(j) + 1$.

We now show the assertion by induction on $\widetilde{dist}(v)$. If $\widetilde{dist}(v) = 0$ then $v = t$ and there is nothing to prove. Thus, let $\widetilde{dist}(v) = k$ and $v = v_0, \ldots, v_k = t$ be a shortest v-t-path in $RN(D, \widetilde{f})$. If edge (v, v_1) is an edge in $RN(D, f)$ as well, we have $dist(v) \leq dist(v_1) + 1 \leq \widetilde{dist}(v_1) + 1 = \widetilde{dist}(v)$ by inductive assumption. Otherwise, (v_1, v) was on a shortest s-t-path in $RN(D, f)$ and we get $dist(v) = dist(v_1) - 1 \leq \widetilde{dist}(v_1) - 1 = \widetilde{dist}(v) - 2$. □

As an immediate corollary we get

Corollary 3. *Each edge is saturated at most $\frac{|V|}{2}$ times in each direction, i.e. the flow on that edge is set to maximal capacity, or to zero respectively.*

Proof. Whenever an edge (i, j) is saturated then $dist(i) = dist(j) + 1$. In order to get saturated again in the same direction, it has to occur in a shortest path in the other direction, first. The distances $\widetilde{dist}$, valid at that moment, satisfy $\widetilde{dist}(j) = \widetilde{dist}(i) + 1 \geq dist(i) + 1 = dist(j) + 2$. As the distances are bounded by $|V|$, the claim follows. □

Theorem 16. *The implementation of Edmonds and Karp computes a maximal flow in $O(|A|^2|V|)$ steps (even if the data is irrational).*

Proof. According to the last Corollary we need at most $O(|A||V|)$ augmentations. The bound follows, since we can find a shortest path in $RN(D, f)$ and update the residual network in $O(|A|)$. □

Remark 9. A complete recomputation of the distances from scratch in each augmentation is not efficient. By a clever update strategy on the distance it is possible to achieve an overall complexity of $O(|A||V|^2)$ (see again [1] for details).

6.6 Max-flow-Min-cut as Linear Programming Duality

The definition of our network flow problem has a strong algebraic flavor and it is already quite close to its linear program, which is as follows:

$$\begin{array}{rll} \max & (-d(s))^\top x & \\ \text{subject to} & (-B)x & = 0 \\ & x & \leq w \\ & x & \geq 0 \end{array}$$

where $d(s) \in \{0, \pm 1\}^A$ is the directed incidence vector of s, that is

$$d(s)_e = \begin{cases} -1 & \text{if } e = (s, i) \\ 1 & \text{if } e = (i, s) \\ 0 & \text{if } e = (i, j),\ i \neq s \neq j \end{cases}.$$

B is the vertex-arc incidence matrix, where the rows indexed by s and t have been deleted.

Going over to the dual we derive the following program:

$$\begin{array}{rl} \min & u^\top w \\ \text{subject to} & u - p^\top B \geq -d(s) \\ & u \qquad \geq 0 \end{array}$$

where u are edge variables and p vertex variables. We interpret the value of a p-variable p_v as the *potential* of vertex v. How does such a dually feasible potential change if we follow its values along an s-t-path $s = v_0, \ldots, v_k = t$ with edge set U? The inequalities involving variables corresponding to edges from U are

$$\begin{aligned} -p_{v_1} + u_{s,v_1} &\geq 1, \\ p_{v_i} - p_{v_{i+1}} + u_{i,i+1} &\geq 0, \\ p_{v_{k-1}} + u_{v_{k-1},t} &\geq 0. \end{aligned}$$

Adding these inequalities yields $\sum_{e \in U} u_e \geq 1$.

If we introduce a potential $p_s = -1$ for s and $p_t = 0$ for t, then all of the above inequalities look alike. Each time we have an increase of Δ in potential from v_i to v_{i+1}, then this increase has to be balanced by the edge variable $u_{i,i+1} \geq \Delta$. Since our objective is to minimize $\sum_{e \in E} w_e u_e$ and there has to be a total increase in potential of 1 along each s-t-path, this is achieved at minimum cost only if we realize it across a minimum cut:

Lemma 14. *Let $D = (V, A, cap)$ be a capacitated network and $s, t \in V$, such that every vertex $v \in V$ lies on some directed s-t-path. Let*

$$MC := \{u \in \mathbb{R}^A \mid \exists p \in \mathbb{R}^V : (p, u) \textit{ is feasible for the dual program}\}.$$

Then $u \in MC$ if and only if there exist (directed) s-t-cuts $C_i = (S_i, V \setminus S_i)$ and coefficients $\lambda_i \in [0, 1], i = 1, \ldots, k$ such that $\sum_{i=1}^{k} \lambda_i = 1$ and $u \geq \sum_{i=1}^{k} \lambda_i \chi(C_i)$. The latter means that MC is the sum of the convex hull of all s-t-cuts plus the cone of the positive orthant:

$$MC = \textit{conv}\,\{\chi(C) \mid C \textit{ st } s\text{-}t\textit{-cut}\} + \mathbb{R}^A_+.$$

Proof. Let $C = (S, V \setminus S)$ be a directed s-t-cut. Setting

$$p_v = \begin{cases} -1 & \text{if } v \in S \\ 0 & \text{if } v \notin S, \end{cases}$$

we see that $(p, \chi(C))$ is feasible for the dual program. Hence, by Lemma 7 conv $\{\chi(C) \mid C$ is s-t-cut$\} \subseteq MC$. Clearly, we can add any positive vector without leaving MC and thus one inclusion follows.

For the other inclusion let (p, u) be feasible for the dual program. We proceed by induction on the number of non-zeroes in u. Let S denote the set of vertices that are reachable from s on a directed path that uses only edges a satisfying $u_a = 0$. Then $t \notin S$ and thus $C_1 = (S, V \setminus S)$ is an s-t-cut. Let $\lambda_1 := \min_{a \in (S, V \setminus S)} u_a$. If $\lambda_1 \geq 1$, the claim follows and founds our induction. Otherwise, set $\tilde{u} = \frac{1}{1-\lambda_1}(u - \lambda_1 \chi(C_1))$ and furthermore

$$\tilde{p}_v = \begin{cases} -1 & \text{if } v \in S \\ \frac{1}{1-\lambda_1} p_v & \text{if } v \notin S. \end{cases}$$

We claim that $(\tilde{p}, \tilde{u})$ is feasible for the dual program. Clearly, $\tilde{u}$ has at least one nonzero less than u. Inequalities that correspond to non-cut edges from $A \setminus C$ are immediately seen to be satisfied. Consider an edge $a = (i, j) \in C$. Note, that by definition of S all vertices $i \in S$ are reached on paths using only arcs $\tilde{i}$ where $u_{\tilde{a}} = 0$ and hence, necessarily, $p_i \leq -1$ must hold. Using this we compute

$$\begin{aligned} \tilde{p}_i - \tilde{p}_j + \tilde{u}_a &= -1 - \frac{1}{1-\lambda_1} p_j + \frac{1}{1-\lambda_1}(u_a - \lambda_1) \\ &= \frac{1}{1-\lambda_1}(\lambda_1 - 1 - p_j + u_a - \lambda_1) \\ &\geq \frac{1}{1-\lambda_1}(p_i - p_j + u_a) \\ &\geq 0. \end{aligned}$$

By inductive assumption there exist directed cuts $C_2, \dots, C_k$ and coefficients $\mu_2, \dots \mu_k$ such that $\tilde{u} \geq \sum_{i=2}^{k} \mu_i \chi(C_i)$ and $\sum_{i=2}^{k} \mu_i = 1$. Putting $\lambda_i = (1 - \lambda_1)\mu_i$ for $i = 2, \dots, k$ yields the desired combination. □

Summarizing, the optimal solution of the dual LP is the incidence vector of a minimal s-t-cut, or at least a convex combination of minimal cuts, and therefore MaxFlow-MinCut is an example of LP-duality.

6.7 Preflow Push

In the algorithm of Ford and Fulkerson we have a feasible flow at any time and thus a feasible solution of the linear program. The minimal cut at the end shows up at sudden. We could try to approach this problem from a dual point of view. A disadvantage of the above method is that the computation of an augmenting path always takes $O(|V|)$ steps, because we compute it from scratch. How bad this may be is best visualized by an example.

Software Exercise 45. Run our Edmonds Karp implementation of the Ford Fulkerson algorithm on the graph `EdmondsKarpBad.cat`. It consists of a path of large capacity and a mesh of several paths of length two and capacity one. Each time we increase the flow by a unit we have to recompute the unique path again and again.

The last example illustrates one basic idea of the preflow push algorithms due to Goldberg and Tarjan: Push as much flow onward as possible. Unfortunately, it is in general not clear a priori what “onward” means. In order to overcome this, we interpret it the following way:

- The (primal) balancing conditions are violated at several vertices, we have an excess of flow into a vertex and we try to fix this by pushing flow, if possible towards the sink.
- Simultaneously we keep a “reference cut” of non-increasing capacity that is used completely by the flow.

The procedure terminates, when the “preflow” becomes a flow, i.e. primally feasible. The main problem is to determine the direction into which the flow shall be pushed. The idea to solve this problem stems from the following physical model. Consider a network of pipelines of the given capacity, where the nodes of the network may be lifted. Naturally, flow goes downhill. Our intention is to send as much flow as possible from s to the sink t. The sink is fixed at ground zero and s at a height of $|V|$. In a preprocessing step we determine for each vertex $v \in V \setminus \{s, t\}$ its minimum height, namely, $dist(v, t)$, the length of a shortest v-t-path. This is the minimal descent that is required for any flow passing v to have a chance to arrive at t. If, now, we let the flow drip from s, due to capacity constraints, an excess might get stuck at some points. These are lifted in order to diminish the excess by allowing some flow back towards the source. Note, that this way, some flow may now go uphill. We will see that, since the height of s and t is fixed, this procedure is finite and in the end some of the initially sent flow will arrive at t and the remaining be returned to s.

Putting this idea into an algebraic framework we define:

Definition 19. *Let $D = (V, A, cap)$ be a capacitated network and $s, t \in V$. An* s-t-preflow $f : A \to \mathbb{R}_+$ *is a function satisfying*

preflow condition: *For all vertices $v \in V \setminus \{s\}$ the preflow out of the vertex does not exceed the preflow into the vertex: $\sum_{(v,w)\in A} f(v,w) \leq \sum_{(w,v)\in A} f(w,v)$,*
capacity constraints: *For all arcs $(u,v) \in A$: $0 \leq f(u,v) \leq cap(u,v)$.*

Let f be a preflow. The residual network $RN(D, f)$ *is defined as in the case of a flow. If $v \in V \setminus \{s,t\}$ then the* excess $ex(f, v)$ of f in v *is defined as $\sum_{(w,v)\in A} f(w,v) - \sum_{(v,w)\in A} f(v,w)$. A vertex with positive excess is called* active.

We implement the idea described above as follows. The height of vertex v is stored in the variable `pot[v]`, which we call the potential of v. This potential should not be confused with the dual variables of the linear program. Now, as long as some vertex `v` has a positive `excess(v)`, we try to find a vertex `u` in `Neighborhood (v)` with `pot[u]` < `pot[v]` and push as much `flow` as possible, namely the minimum of the residual capacity `res((v,u))` and the `excess` of the vertex, along `(v,u)`. If such a vertex does not exist, we enforce its existence for the next iteration by updating `pot[v] = minResNeighborPot(v)+1` the potential of `v` to one more than the minimal potential of a neighbor reachable through a directed arc with a non-vanishing residual capacity.

ALGORITHM `PreflowPush`

```
InitPotential(s,t)
for v in G.Neighborhood(s):
    flow[(s,v)] = cap((s,v))

feasible = False
while not feasible:
    pushed = False
    v = FindExcessVertex()
    if not v:
        feasible = True
    else:
        for u in Neighborhood(v):
            if pot[v] > pot[u]:
                delta = Min(res((v,u)),excess(v))
                if ForwardEdge(v,u):
                    flow[(v,u)] = flow[(v,u)] + delta
                else:
                    flow[(u,v)] = flow[(u,v)] - delta
                pushed = True
                break
        if not pushed:
            pot[v] = minResNeighborPot(v)+1

ShowCut(s)
```

The line `pot[v]=minResNeighborPot(v)+1` is feasible which can be seen as follows. If $v \in V \setminus \{s,t\}$ satisfies $ex(f, v) > 0$ and hence

$$\sum_{(u,v)\in A} f(u,v) > \sum_{(v,u)\in A} f(v,u),$$

then it must have some neighbor w such that $f(w,v) > 0$ and as a consequence w is a neighbor in the residual network.

It is not clear a priori, that the algorithm is finite and we have to work on that. Interestingly, we will not directly prove that the flow must become feasible, but prove instead that the labels `pot` are bounded. For that purpose, we first show that the vertex labels can be considered as lower bounds for the distances in the residual network. This will be an immediate consequence of the following Proposition:

Proposition 5. *During execution of the while-loop at any time we have* $\texttt{pot[i]} \le \texttt{pot[j]} + 1$ *for all edges* $(i,j) \in RN(D,f)$.

Proof. In the beginning this is guaranteed by the preprocessing. Whenever additional flow is sent on an edge (i,j), by inductive assumption we have $\texttt{pot[i]} \le \texttt{pot[j]}+1$. Since we send flow along that arc on the other hand we necessarily have $\texttt{pot[i]} > \texttt{pot[j]}$, thus $\texttt{pot[i]} = \texttt{pot[j]} + 1$. In particular, if an arc (j,i) is newly introduced in the residual network it satisfies $\texttt{pot[j]} = \texttt{pot[i]} - 1 \le \texttt{pot[i]}+1$. If a label is increased, the assertion directly follows from the inductive assumption. □

Now, we can bound the minimum length of a path by the difference in potential

Corollary 4. *If* $\texttt{pot[v]} = k$ *and* $\texttt{pot[w]} = l > k$*, then any directed w-v-path in the residual network $RN(D,f)$ uses at least $l-k$ edges.*

Next, we consider the situation that for some vertex v there no longer exists a directed v-t-path, but it has a positive excess. What do we do then? We increase its label, but the backward arcs of the flow, that accumulates in v, bound the labels, since there always exists an s-v-path in this situation:

Proposition 6. *If $v \in V \setminus \{s,t\}$ and $ex(f,v) > 0$ then there exists a directed v-s-path in $RN(D,f)$.*

Proof. Let S denote the set of vertices v that are connected to s by a directed v-s-path. Clearly $s \in S$. By definition of $RN(D,f)$ we cannot have a non-zero flow on any edge $e \in (S, V\setminus S)$ leaving S. The preflow condition for $V \setminus S$ implies

$$\begin{aligned} 0 \le \sum_{v\in V\setminus S} ex(f,v) &= \sum_{v\in V\setminus S}\left(\sum_{(u,v)\in A} f(u,v) - \sum_{(v,w)\in A} f(v,w)\right) \\ &= \sum_{e\in(S,V\setminus S)} f(e) - \sum_{e\in(V\setminus S,S)} f(e). \end{aligned}$$

As there is no flow on any arc leaving S we get

$$0 = \sum_{e\in(S,V\setminus S)} f(e) \ge \sum_{e\in(V\setminus S,S)} f(e) \ge 0.$$

Hence all there is no flow on the arcs of $(V \setminus S, S)$ implying

$$\sum_{v \in V \setminus S} ex(f, v) = 0.$$

As the preflow condition guarantees a non-negative excess, we must have $ex(f, v) = 0$ for all $v \in V \setminus S$. □

Since the label of s is fixed to $|V|$, we get the following Corollary, which implies finiteness of the algorithm.

Corollary 5. *Throughout the algorithm and for all $v \in V$ we have:* `pot[v]` $\leq 2|V| - 1$.

Proof. The label of a vertex is increased only if it has a positive excess. But then there exists a directed path to s in the residual network and the claim follows from Corollary 4. □

As now the finiteness of the algorithm has become clear, we examine its time complexity. In each iteration of the while loop we either increase the potential of a vertex or we perform a push on some edge. By Corollary 5 the number of relabel operations is of order $O(|V|^2)$. The same considerations we made for the Edmonds-Karp implementation of the Ford-Fulkerson algorithm in Corollary 3 yields an upper bound of $O(|V||A|)$ on the number of saturating pushes. In order to bound the number of non-saturating pushes we use the trick of a "potential function".

Lemma 15. *The number of non-saturating pushes in a run of the preflow push algorithm is $O(|V|^2|A|)$.*

Proof. We consider the sum of the labels of the active vertices

$$\Phi := \sum_{i \text{ is active}} \texttt{pot[i]}$$

as a *potential function*. By Corollary 5 this number is bounded by $2|V|^2$. When the algorithm terminates we have $\Phi = 0$. Consider a non-saturating push along (i, j). Afterwards i is no longer active but j is active. As `pot[j]` $=$ `pot[i]` $- 1$ this implies:

A non-saturating push decrements Φ by at least 1.

The claim now follows if we can show that the total increase of Φ is of order $O(|V|^2|A|)$. We consider the two possible cases of an increase of Φ.

Case 1: A saturating push is performed. This may result in a new active vertex. Therefore the potential function may grow, but at most by $2|V| - 1$. As there are $O(|V||A|)$ saturating pushes we can bound the total increase of Φ caused by saturating pushes by $O(|V|^2|A|)$.

Case 2: A vertex is relabeled. Since the labels never decrease and are bounded by $2|V| - 1$, the total increase caused by relabeling operations is of order $O(|V|^2)$. □

Summarizing we have the following bound on the running time of our first version of the preflow push algorithm.

Theorem 17. *The running time of the preflow push algorithm is of order* $O(|V|^2|A|)$.

You may have realized that we did not make any comment on the correctness of the algorithm, yet. All we know by now is that it computes a flow, since there are no active vertices left at the time of termination. We will conclude correctness directly from the existence of a cut of the same size using linear programming duality.

6.8 Preflow Push Considered as a Dual Algorithm

As seen above the initial preflow is infeasible. We will show that it is optimal as soon as it becomes feasible. This is done by providing a saturated cut for the preflow at each stage. This saturated cut is *complementary* to the preflow as its variables are zero if the corresponding inequalities of the capacity constraints are strict; i.e., we always have $(f_a - cap_a)u_a = 0$. Furthermore, the potentials as defined in the proof of Lemma 14 guarantee that $(p_i - p_j + u_{ij})f_{ij} = 0$. To see this, note that the left coefficient in that product is nonzero only on backward arcs of the saturated cut which will have a zero flow.

Recall from Theorem 5 that in general a complementary pair of primally and dually feasible solutions is optimal for both programs.

Recall the following equation about "conservation of mass" that holds for any function f on the arcs if $[S, V \setminus S]$ is an s-t-cut:

$$f((S, V \setminus S)) - f((V \setminus S, S)) = \sum_{v \in V \setminus S} ex(f, v). \tag{6.2}$$

For our dual solution we define the cuts iteratively. First, we set $S_0 = \{s\}$. Thus, in the beginning of the while loop of the algorithm `PreflowPush` on page 80 $(S_0, V \setminus S_0)$ is a saturated s-t-cut. Now, if $(S_i, V \setminus S_i)$ is a saturated s-t-cut for some steps in the algorithm we define S_{i+1} when it becomes necessary—because $(S_i, V \setminus S_i)$ is no longer saturated—as follows. As soon as some edge $(u, v_0) \in (S_i, V \setminus S_i)$ becomes an edge of the residual network $RN(D, f)$ we set

$$S_{i+1} := S_i \cup \{v \in V \mid \text{ there exists a directed } v_0\text{-}v \text{ path in } RN(D, f)\}.$$

Then the following holds

Lemma 16. *S_i is an s-t-cut for all i and*

(i) $S_i \subseteq S_{i+1}$,
(ii) $cap[S_i] = \sum_{v \in V \setminus S_i} ex(f, v)$,
(iii) $cap[S_i] \geq cap[S_{i+1}]$.

Proof. In order to show that $t \notin S_i$ it suffices to show that for all i and all $v \in S_i$ at any time `pot[v]` $> |V \setminus S_i|$, which implies the assertion as `pot[t]` $= 0$. By definition this holds for S_0. We proceed by induction on i. Let $u = v_0, v = v_k$ and $v_0 v_1 \dots v_k$ be a directed path in $RN(D, f)$, where $u \in S_i$ and $\{v_1, \dots, v_k\} \cap S_i = \emptyset$. If `pot[v]` $\geq$ `pot[u]` we are done. Otherwise by Corollary 4, we have `pot[u]` $-$ `pot[v]` $\leq k$ and thus `pot[v]` $\geq$ `pot[u]` $- k > |V \setminus S_i| - k \geq |V \setminus S_{i+1}|$. Now, assertion 1 is clear by definition.

As by construction, when S_{i+1} is defined, all forward edges of $[S_{i+1}, V \setminus S_{i+1}]$ are saturated and all backward edges have zero flow, the second assertion follows from (6.2). Since the excesses are nonnegative and the sets S_i grow monotonically the third assertion follows from the second. □

Now, we get the correctness of the algorithm as a corollary.

Corollary 6. *The preflow push algorithm computes a maximal flow.*

Proof. As mentioned above the preflow has become a flow when the algorithm terminates, i.e. t is the only active node and the net flow into t equals the value of the cut S_i. □

6.9 FIFO-Implementation

We can get an improved bound on the running time by a clever implementation of the choice of active vertices. Furthermore, it seems quite natural to work on an active vertex until it is no longer active or has to be relabeled. We say that we *examine a node*. If we have to relabel a node we append it to the end of the queue.

The choice of the vertex can be done by maximal label or maximal excess to achieve the best bounds of $O(\sqrt{|A|}|V|^2)$ respectively $O(|V||A| + |V|^2 \log(U))$, where U denotes an upper bound on the (integer) capacities (see again [1] for details). For a recent overview on implementations and their complexities we recommend [38].

Here we will discuss the simpler alternative of a First-In-First-Out queue that we already met several times during our course. This means that we always choose the oldest active node, which is at the beginning of the queue. A vertex that becomes active is added to the end of the queue.

The running time is analyzed as follows: We provide a time stamp for each active node. The vertices v that become active in the initialization receive a time stamp of $t(v) = 1$. If vertex v becomes active because edge (u, v) is pushed it receives the time stamp $t(v) = t(u) + 1$. Furthermore the time stamp is incremented if the vertex is relabeled. We will show that using this rule $t \leq 4|V|^2 + |V|$. Recall that the number of non-saturating pushes dominated the running time of our naive implementation and that while examining a node we perform at most one non-saturating push. Hence, the above inequality improves the result of our running time analysis in Section 6.7 to $O(|V|^3)$.

As potential function for our analysis here we consider

$$\Phi = \max\{0, \{\texttt{pot[i]} \mid i \text{ is active}\}\}.$$

By Corollary 5, the potential of each vertex is bounded by $2|V|-1$ and hence the total increase the of the potential function caused by relabel operations is bounded by $|V|(2|V|-1) \leq 2|V|^2$.

If during the examination of vertices with time stamp k, we call this a *round*, no vertex is relabeled, then the excess of all active vertices has been passed to vertices with smaller label and Φ has decreased by at least 1. Otherwise, a vertex `v` has been relabeled. In this case, the potential function may increase by at most `pot[v ]`$< 2|V|-1$. Since the total increase of the sum of all labels (active or non-active) caused by relabel operations is bounded by $2|V|^2$ this can happen in at most $2|V|^2$ rounds and lead to an increase of Φ of at most $2|V|^2$.

As $\Phi < |V|$ in the beginning and since in all other rounds without a relabeling operation we decrease the potential by at least one, there can be at most $|V|+2|V|^2$ such rounds. We conclude that our algorithms terminates after at most $4|V|^2+|V|$ rounds and hence all time stamps have a value of at most $4|V|^2+|V|$. Since no vertex will receive the same time stamp twice and while examining a node we perform at most one non-saturating push the number of non-saturating pushes is bounded by $O(|V|^3)$. Using the analysis of the last section we get the following theorem:

Theorem 18. *The FIFO-implementation of the preflow push computes a maximal flow in* $O(|V|^3)$.

A final remark on “real implementation” of the preflow push.

Software Exercise 46. For an efficient implementation of the preflow push it is mandatory to avoid the clumsy explicit computation of the way that the excess of the preflow is sent back to s. How costly this may be is illustrated by running the algorithm on the example graph `PreflowPushWC.cat`.

One possibility to overcome this problem is to heuristically search for small saturated cuts. The s-side of this cut can be neglected in the following computations.

Exits

The dual pair of problems Max-Flow and Min-Cut is tractable, since in the case of flows we consider only one kind of flow or good that is distributed through the network and two vertices which have to be separated. If we have several goods and pairs of origin and destination, where we want to distinguish the flow we get a “Multiflow”-Problem which is tractable only in a few special cases (see [38]). If we want to separate more than two terminals we have a multiterminal-cut problem which is intractable already for three terminals in general networks [13].

One can generalize the notion of a cut from our definition by dropping the sources and sinks and rather ask about finding minimum cuts in a graph, minimum edge sets which lead to 2 or k connected components when removed. For fixed k, Goldschmidt and Hochbaum [22] give a polynomial time algorithm for this problem, which arises for example in the context of analyzing data by clustering, that is identifying groups of similar objects. In this context the vertices represent objects—e.g., web pages, patients, or pixels in an image—and the edge weight corresponds to the similarity—shared words, common genetic variations, like color and intensities—between objects. Sometimes G is then called a similarity graph and the connected components in the graph without the cut are the clusters. Hence, the k-way cut problem can be viewed as identifying a minimal set of similarities which we ignore to arrive at k clusters. More involved cost functions for cuts, normalized and weighted [8], have been introduced to compensate for artifacts such as unbalanced cluster sizes. Algorithmically, spectral algorithms which rely on the eigenvectors and eigenvalues of the Laplacian of the adjacency matrix, are the state of the art in clustering in similarity graphs [27].

Exercises

Exercise 47. Let $D = (V, A, cap)$ be a capacitated directed network and $S, T \subseteq V$ two disjoint vertex sets. An $S - T$-flow is a function that satisfies flow conservation for all vertices in $V \setminus (S \cup T)$. We define the *value* of the flow

$$\begin{aligned} |f| &:= \sum_{s \in S} \left(\sum_{(s,w) \in A} f(s,w) - \sum_{(w,s) \in A} f(w,s) \right) \\ &= \sum_{t \in T} \left(\sum_{(w,t) \in A} f(w,t) - \sum_{(t,w) \in A} f(t,w) \right). \end{aligned}$$

Give a reduction from the problem to find a maximal $S - T$-flow to the task of maximizing a flow from a single source to a single sink.

Exercise 48. Consider a directed network $D(V, A, cap, low)$ with upper and lower bounds on the flow of an arc. An s-t-flow is feasible if it respect these bounds at all arcs and Kirchhoff's law at all vertices different from s and t.

(i) Give a reduction of the problem of the existence of a feasible flow to an ordinary MaxFlow-Problem.
(ii) Modify the algorithm of Ford and Fulkerson in order to compute a maximum flow in a network with upper and lower bounds.

Exercise 49. Write an algorithm to compute an s-t-path of maximum residual capacity in a residual network and analyze its complexity.

Exercise 50. The following is an abstract model of a reload problem where we have to decide to make a delivery directly or via one of two *hubs*. Given a graph $G = (V, E)$, a nonnegative integer weight function $w_0 : E \to \mathbb{Z}^+$ on the edges and two nonnegative integer weight functions on the vertices $w_1, w_2 : V \to \mathbb{Z}^+$. We will say $V' \subseteq V$ *satisfies* an edge $e = (u, v)$ if $\{u, v\} \subseteq V'$. The objective is to find a set of edges $F \subseteq E$ and 2 subsets of the vertices $V_1, V_2 \subseteq V$ such that for all $e = (u, v) \in E$ either $e \in F$ or V_1 or V_2 satisfies e, and

$$\sum_{e \in F} w_0(e) + \sum_{v \in V_1} w_1(v) + \sum_{v \in V_1} w_1(v)$$

is minimized. Model this problem as a MinCut-Problem in a capacitated network.