Contents

1 Introduction **2**
1.1 Basic Notation and Definitions . . . 3
1.1.1 Graph Theory Basics . . . 3
1.1.2 Combinatorial Optimization . . . 4
1.2 Motivation . . . 7
1.2.1 Related Work . . . 7
1.3 The Acyclic Flowbound Problem . . . 9
1.3.1 Relation To Other Flow Problems/Relaxations . . . 12
1.3.2 Computational Complexity . . . 15

2 Models and Solving Approaches **18**
2.1 MIP Formulations . . . 18
2.1.1 Model 1: Node Potentials . . . 18
2.1.2 Model 2: Acyclicity Constraints On All Cycles . . . 21
2.2 The Number of Cycles we have to forbid . . . 22
2.3 The Constrained Path Augmentation Model . . . 27
2.4 A Simple Heuristic Approach . . . 33

3 Implementation and Computational Results **39**
3.1 Implemented Algorithms . . . 39
3.1.1 Cyclic Flow Bounds . . . 39
3.1.2 MIP with Node Potentials and Binary Direction Variables . . . 40
3.1.3 Relaxation of the Node Potential Model . . . 41
3.1.4 MIP with Binary Direction Variables and separated Acyclicity Constraints . . . 41
3.1.5 Relaxations of the Exact Separation Model . . . 42
3.1.6 Acyclicity Constraints on a Cycle Base . . . 42
3.2 Tables . . . 42
3.2.1 Bound Improvements . . . 42
3.2.2 Model building impact . . . 44

4 Summary **47**

Bibliography **48**

1 Introduction

Today there are various real-world problems that are (tried to be) solved by computational methods. One of them (which is currently attacked at the Konrad Zuse Zentrum für Informationstechnik Berlin (ZIB)) is the Problem of transporting natural gas through a network of pipelines. Due to the physics of gas flows this turns out to be a nonlinear, nonconvex mixed-integer optimization problem, which is is quite hard to solve.

On first glance it may seem that solving a normal Min-Cost-Flow algorithm could give an appropriate bound on an arc for this problem. If we set the weight w_e on an arc e to 1 or -1 and all other weights on zero, we get a valid bound on the maximum or minimum possible flow on the arc. However, this bound is for most of the arcs far from reality. The reason is, that a Min-Cost-Flow could have cycles within it, and the algorithm could maximize the flow by sending as much flow as possible on a cycle. In natural gas networks this can not happen since the flow is always induced by pressure differences. The only exception are active compressure stations. While they increase pressure it is possible that they cause cyclic flow. However, in real gas networks a compressor station would normally be placed at a point where it has the highest impact. This is of course where it is unlikely to waste energy by sending flow on a cycle. The pressure would rather be controlled by resistors, valves and control valves in order to avoid cycing. So we will assume that flow is always acyclic in gas networks. So (ignoring that compressor stations could increase pressure) there is no cyclic flow. The intention of this thesis now is to investigate the problem of flow bounds in a network where the flow is assumed to be acyclic.

We take this gas network problem as motivation but do not use gas physics properties other than acyclicity. Instead we are only interested in the abstract network flow problem with the additional condition of flows being acyclic, and in the bounds on flow value in such a network.

In this thesis we first define the mathematical/algorithmical problem and examine connections to known problems and relaxations of the acyclic flowbound problem. We show that the minimum cost flow problem with general (possibly negative) weights becomes $\mathcal{NP}$-hard if we demand acyclic flow. Justified by this we model the problem as a mixed integer program. We present two different mixed integer programs that model our exact problem. One of the MIP models relies on constraints which each forbid cyclic flow on a cycle of the network. A network can have an exponential number of cycles. We show a condition under which an acyclicity constraint becomes redundant. Depending on the network we are given this could reduce the number of these constraints to be linear in the input size. Still on some networks the condition is not applicable. We furthermore discuss the idea of an approach that tries to modify and adjust path based algorithms for maximum flow or minimum cost flow in order to solve this problem. We present examples to show that this can only produce a heuristic algorithm or else requires to solve another hard problem as subroutine. A heuristic algorithm close to the one we would get her is described and analysed.

The algorithms/models we discussed were also implemented. in the end we give details on the algorithms that were implemented and compare results achieved by them.

1.1 Basic Notation and Definitions

1.1.1 Graph Theory Basics

Since there are many definitions in graph theory which may differ slightly, we want to introduce now the basic notation and definitions that we use throughout this thesis. The definitions in this chapter are mainly taken from the textbook about combinatorial optimization by Korte and Vygen [14].

Definition 1.1. An undirected graph is a triple (V, E, Ψ), where V and E are finite sets and $\Psi : E \to \{X \subseteq V : |X| = 2\}$.
A directed graph or digraph is a triple (V, E, Ψ), where V and E are finite sets and $\Psi : E \to \{(v, w) \in V \times V : v \neq w\}$.

The elements of V are called vertices, the elements of E are the edges. Ψ is the incidence function giving the relation between elements of V and elements of$E$.
In this thesis by a graph we normally mean the directed graph. If we talk about undirected graphs it will be stated explicitly. Edges of directed graphs will also be called arcs to make clear that they are directed.

Two edges e, e' with $\Psi(e) = \Psi(e')$ are called parallel. Graphs without parallel edges are called simple. For simple graphs we usually identify an edge e with its image $\psi(e)$ and write $G = (V(G), E(G))$ or $G = (V, E)$, where $E(G) \subseteq \{X \subseteq V(G) : |X| = 2\}$ or $E(G) \subseteq V(G) \times V(G)$. In this thesis all graphs are considered simple so we use this notation.

Definition 1.2. $|E(G)| = m$ denotes the number of edges and $|V(G)| = n$ denotes the number of vertices of a graph.

Definition 1.3. We say that an edge $e = \{v, w\}$ or $e = (v, w)$ joins v and w. In this case, v and w are adjacent. v is a neighbour of w (and vice versa). v and w are the endpoints of e. If v is an endpoint of an edge e, we say that v is incident with e. In the directed case we say that $e = (v, w)$ leaves v and enters w, v is the tail and w is the head of the arc e.

Definition 1.4. For a digraph G the underlying undirected graph is the undirected graph G' on the same vertex set which contains an edge $\{v, w\}$ for each edge (v, w) of G. We also say that G is an orientation of G'.

Definition 1.5. A subgraph of a graph $G = (V(G), E(G))$ is a graph $H = (V(H), E(H))$ with $V(H) \subset V(G)$ and $E(H) \subset E(G)$. We also say that G contains H. H is an induced subgraph of G if it is a subgraph of G and $E(H) = \{\{x, y\}$ resp. $(x, y) \in E(G) : x, y \in V(H)\}$. Here H is the subgraph of G induced by $V(H)$.

Definition 1.6. For $e \in E(G)$, we define $G - e := (V(G), E(G) \setminus \{e\})$. Furthermore, the addition of a new edge e is abbreviated by $G + e := (V(G), E(G) \cup e)$.

Definition 1.7. For undirected graphs G and a set of vertices $X \subseteq V(G)$ we define the cut induced by X

$$\delta(X) := \{\{x, y\} \in E(G) : x \in X, y \in V(G) \setminus X\}$$

For digraphs G and $X \subseteq V(G)$ we define

$$\delta^+(X) := \{(x, y) \in E(G) : x \in X, y \in V(G) \setminus X\}$$

and

$$\delta^-(X) := \{(y,x) \in E(G) : x \in X, y \in V(G) \setminus X\}$$

For one-element vertex sets $\{v\}$ (with $v \in V(G)$) we write $\delta(v) := \delta(\{v\})$ for the set of incident edges of v, and in the directed case $\delta^+(v) := \delta^+(\{v\})$ for the outgoing arcs and $\delta^-(v) := \delta^-(\{v\})$ for the set of ingoing arcs of v

We use subscripts (e.g. $\delta_G(X)$) to specify the graph G if necessary.

Definition 1.8. The degree of a vertex v is $|\delta(v)|$, the number of edges incident to v. In the directed case, the in-degree is $|\delta^-(v)|$, the out-degree is $|\delta^+(v)|$, and the degree is $|\delta^+(v)| + |\delta^-(v)|$.

A vertex v with zero degree is called isolated. A graph where all vertices have degree k is called k-regular.

Definition 1.9. An edge progression W in G is a sequence $v_1, e_1, v_2, \ldots, v_k, e_k, v_{k+1}$ such that $k \geq 0$, and $e_i = (v_i, v_{i+1}) \in E(G)$ resp. $e_i = \{v_i, v_{i+1}\} \in E(G)$ for $i = 1, \ldots, k$. If in addition $e_i \neq e_j \, \forall \, 1 \leq i < j \leq k$, W is called a walk in G. W is closed if $v_1 = v_{k+1}$.
A path is a graph $P = (\{v_1, ..., v_{k+1}\}, \{e_1, ..., e_k\})$ such that $v_i \neq v_j$ for $1 \leq i < j \leq k+1$ and the sequence $v_1, e_1, v_2, \ldots, v_k, e_k, v_{k+1}$ is a walk. P is called a path from v_1 to v_{k+1} or a $v_1 - v_{k+1}$-path. v_1 and v_{k+1} are the endpoints of P.

Evidently, there is an edge progression from a vertex v to another vertex w if and only if there is a $v - w$-path.

Definition 1.10. A cycle is a graph $(\{v_1, \ldots, v_k\}, \{e_1, \ldots, e_k\})$ such that the sequence $v_1, e_1, v_2, \ldots, v_k, e_k, v_1$ is a (closed) walk and $v_i \neq v_j$ for $1 \leq i < j \leq k$.

An easy induction argument shows that the edge set of a closed walk can be partitioned into edge sets of cycles.

Definition 1.11. The length of a path or cycle is the number of its edges. If it is a subgraph of G, we speak of a path or cycle in G.

Definition 1.12. A spanning path (i.e. a path containing all vertices) in G is called a Hamiltonian path while a spanning cycle in G is called a Hamiltonian cycle. A graph containing a Hamiltonian cycle is a Hamiltonian graph.

1.1.2 Combinatorial Optimization

Optimization of mathematical problems including to make certain discrete decisions is an important field of application for modern mathematics. The acyclic flow problem we describe in this thesis is a typical example of a combinatorial, graph theoretical optimization problem. Here we define such problems in general as well as the basics of integer programming which is a standard approach to solve such problems.
The definitions are taken from the scriptum of the course ADM I by Martin Grötschel held in 2012/2013.

Definition 1.13. Given a finite set $\mathcal{I}$ and a function $f : \mathcal{I} \to \mathbb{R}$. The problem of finding the element $i \in \mathcal{I}$ such that $f(i)$ is maximum or minimum is a general combinatorial optimization problem.

Of course it is hard to solve such a problem without having any structure on it. But the problems we have to deal with normally have a good structure on them: The set $\mathcal{I}$ is defined implicitly by some conditions on its elements and the function f is defined by formulas.

Definition 1.14. Given some finite set E , a set $\mathcal{I} \subset \mathcal{P}(E)$ of feasible solutions and a function $c : E \to \mathbb{R}$. Define for a set $S \in \mathcal{P}(E)$ the total costs $c' : \mathcal{P}(E) \to \mathbb{R}$, $c'(S) = \sum_{s \in S} c(s)$.
The problem of finding an element $i_m \in \mathcal{I}$ with $c'(i_m) \geq c'(i) \forall i \in \mathcal{I}$ (or $c'(i_m) \leq c'(i) \forall i \in \mathcal{I}$) we call a combinatorial maximization (minimization) problem with linear objective function.

Often we can model such combinatorial optimization problems as linear, mixed-integer, quadratic or constraint program depending on the definition of the feasible set $\mathcal{I}$.
As we will make use of such programs we want to define the basics here. We use the framework SCIP ("Solving Constraint Integer Programs") for solving our problem. For details of the solver setup we refer to the PhD Thesis of Tobias Achterberg [1] where he describes the main design of his Constraint Integer Programming solver SCIP. The following definitions also follow the work of Achterberg [1].

Definition 1.15. A *constraint program* is a triple $(\mathcal{C}, \mathcal{D}, f)$ and consists of solving

$$f^* = \min\{f(x) | x \in \mathcal{D}, \mathcal{C}(x)\}$$

with $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ the domain set, $f : \mathcal{D} \to \mathbb{R}$ the objective function and $\mathcal{C} = \{\mathcal{C}_1 \ldots \mathcal{C}_m\}$ the constraint set.

As a special case of constraint programs that can be handled by computers and can be used to model many different combinatorial optimization problems we have mixed integer programs (or shortly referred to as MIP). Here the constraints and the objective function can be given as linear inequalities and in addition the domains of some variables can be restricted to integer values. Often variable domains are even more specifically restricted to $\{0, 1\}$. The regarding variables are called binary (decision) variables.

Definition 1.16. A *mixed integer program* or *MIP* is a constraint program with the following representation:
Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ and a set $\mathcal{I} \subseteq \{1 \ldots n\}$ we have to solve

$$c^* = \min\{c^T x | Ax \leq b, x \in \mathbb{R}^n, x_j \in \mathbb{Z} \text{ for all } j \in \mathcal{I}\}$$

We call all the vectors x in the set

$$X_{MIP} = \{x \in \mathbb{R}^n | Ax \leq b, x \in \mathbb{R}^n, x_j \in \mathbb{Z} \text{ for all } j \in \mathcal{I}\}$$

i.e. vectors fulfilling the constraints of the problem, the *feasible solutions* of the MIP. A vector $x' \in X_{MIP}$ with $c^* = c^T x'$ is called an *optimal solution* of the MIP.

There are important special cases of mixed integer programs: The set $\mathcal{I}$ of integer variables could be empty so we really just optimize over a polyhedron. This kind of MIP with $\mathcal{I} = \emptyset$ is called a *linear program* or *LP*.
It can also happen that every variable is integer, i.e. $\mathcal{I} = \{1, \ldots, n\}$. We call this MIP an *integer program* or *IP* in this case.

Solving linear programs can be done in polynomial time. Algorithms to achieve this were described by Khachiyan[13] and Karmarkar [11]. Integer programs and by this also their generalization mixed integer programs can be (and mainly are) used for modeling $\mathcal{NP}$ hard problems. So unless $\mathcal{P} = \mathcal{NP}$ there is no polynomial algorithm to solve them.
Thus the LP relaxation of a MIP is a very important problem.

Definition 1.17. The LP-relaxation of a mixed integer program

$$\min\{c^T x | Ax \leq b, x \in \mathbb{R}^n, x_j \in \mathbb{Z} \text{ for all } j \in \mathcal{I}\}$$

is the MIP without any integrality constraints:

$$\min\{c^T x | Ax \leq b, x \in \mathbb{R}^n\}$$

It can be solved efficiently since it is a linear program. The solution of the LP relaxation of a problem is giving a bound on the solution of the mixed integer problem. It is used to get a starting point for a branching process or cutting planes that might yield an optimal or nearly optimal solution. Also the bounds from LP relaxations are needed to cut off subtrees while branching on the variables of the problem.

1.2 Motivation

Natural gas is one of the most common ressources of energy in germany and makes up more than 20 percent of energy consumption. Most of this gas is produced in ressource-rich countries like Russia or Norway and transported to Germany through special pipelines. This thesis evolved from a joint research project of the Konrad Zuse Zentrum fuer Informationstechnik Berlin(ZIB) with Open Grid Europe (OGE) who operate the largest network of gas pipelines in germany. A general overview over the work on the gas network planning problem is given in [6]. The problem of finding settings for all active components of the network such that given demands and supplies can be sent through the network with respect to all constraints is the nomination validation problem. A description of the model used and work done for the nomination validation problem can be found in [16].
The flow of gas through a pipeline network is determined by physical conditions such as pressure and temperature. Pressure can be increased in compressor stations and controlled by elements like valves. Pressure loss along a pipe is described by ordinary differential equations.
The computation of the flow is difficult due to numerical and algorithmical reasons. Good preprocessing routines can help a lot by reducing the domains and model sizes before the main computation is started. Since the flow of gas in the network is always determined by pressure differences and flow is always going from higher pressure to lower pressure we know that flow in the network in general has to be acyclic. The only exception of this is a compressor station. An active compressor station can increase the pressure and by this could indeed cause cyclic flow. However we will simplify the model and assume acyclic flow. We can compute a flow with maximum value on a specified arc in the network with a standard Minimum Cost Flow algorithm. However a Minimum Cost Flow Algorithm usually is allowed to have cyclic flows and thus will maximize flow by shifting it around a cycle containing the maximized arc as long as the flow on the cycle can be increased on every arc.
The main question leading to this thesis was how we could improve bounds and compute flows if we assume the flow to have no cycles. Normal and multicommodity network flows are well known algorithmical problems, but (as far as the author knows) there is no work on acyclic network flows and how to compute them. Whereever we have a network with nonpositive lower capacity bounds, we can make any flow acyclic by shifting flow back on the cycles until an edge has zero flow. But if we are interested in the maximum possible flow on an edge under the condition of acyclicity things get more complicated.
We simplified the problem by assuming that there are no compressor stations. At the same time we just ignore the physical properties of the gas. We do not compute or take care of any gas density, pressure or possible pressure loss. Pressure is only used to justify the assumption of acyclicity. This abstraction is justified by the fact that it is weaker than the original problem which also has to fulfill the flow conditions we use. The bounds are only needed for preprocessing and hoped to improve the model size in the end.

1.2.1 Related Work

The diploma thesis of Thea Göllner [9] describes preprocessing techniques for stationary gas network optimization models. This is the same model as the one we use here. In her thesis she also describes an algorithm for determining flow directions on the arcs

of the network and determines cases in which we can fix the flow direction even on arcs contained in cycles of a certain kind. To achieve this she makes use of the acyclicity property of gas flow as well. Fixing a flow is basically saying that the lower flow bound in one direction is zero. Thus her work is a special case of the much more general approach on preprocessing acyclic bounds we have in here. The fixing of flow directions that she proposes works on induced paths of the network and on cycles where flow can only be coming in from one side. She also shows the restrictions of this approach and says that for many cases it is necessary to take pressure or flow quantity into account in order to see if one can fix the direction variable.
Here we aim at computing more general acyclic flow bounds. So we can not only fix the direction of flow in some special cases but also give upper bounds on possible flow and bounds in cases were it is not clear which direction flow has to take. It still yields flow directions in the cases described. But in practice a combination of our models with the flow direction fixing Göllner did could improve runtime and quality, because for big networks we can only use heuristic algorithms in practice.

1.3 The Acyclic Flowbound Problem

We already described the gas flow problem, where the motivation for this thesis came from. Here we want to define the problem as a general combinatorial flow problem with specific contraints. We will also give the formulation as a Mixed Integer Program (MIP) and as well some natural relaxations, which might be easier to solve.
We will represent our originally undirected graph by a directed graph where flow is allowed to go over edges backward and forward as well. This allows us to specify directions forward and backward on every edge in a consistant way.

Definition 1.18. Let $G = (V, A)$ be a directed Graph and $e \in A$ a specific arc of G. For every vertex $v \in V$ let there be a prescribed range for the amount of flow demand $[d_l(v), d_u(v)] \subset \mathbb{R} \setminus \{0\}$. Vertices with demand values greater than 0 are sinks, vertices with negative demand value are sources. In a feasible solution flow demands are assumed to be balanced on their (absolute) higher bound values

$$\sum_{v \in V \text{ sink}} d_u(v) - \sum_{v \in V \text{ source}} d_l(v) = 0$$

Let there be a capacity function

$$c : A \to \mathcal{P}(\mathbb{R}),\ c(a) = [c_l(a), c_u(a)],\ c_l(a) \leq 0 \leq c_u(a)\, \forall a \in A$$

and a flow $f : A \to \mathbb{R}$ with

$$c_l(a) \leq f(a) \leq c_u(a)\, \forall a \in A$$

and

$$\sum_{a \in \delta^-(v)} f(a) - \sum_{a \in \delta^+(v)} f(a) - d(v) = 0\, \forall v \in V$$

We call this a *feasible network flow on G*, i.e. a flow with balances of 0 and respected capacity bounds. We also call

$$\sum_{a \in \delta^-(v)} f(a) - \sum_{a \in \delta^+(v)} f(a) - d(v)$$

the flow balance, which might be nonzero in incomplete flows during a solving algorithm.

The standard definition of Flow Problems has fixed demand values $d(v) = const$ instead of an interval. These two definitions are equivalent in the way that you can transform them into each other easily in polynomial time. Obviously we can see fixed demands $d(v)$ as intervals with $d_l(v) = d_u(v) = d(v)$ consisting of only one point $[d(v), d(v)]$. The other way around we use a construction that adds only one node and $|\{v \in V | d_u(v) < 0\}| + |\{v \in V | d_l(v) > 0\}|$ arcs to the network.

Proposition 1.19
There is a (linear time) transformation from the flow problem with demand intervals on the vertices to the standard flow problem with fixed demands.

Proof. Given a graph $G = (V, A)$ with demand intervals at the entries and exits, we construct a graph $G' = (V', A')$ as follows:

$$V' := V \cup \{z\}, \text{ and } A' := A \cup \{(v, z) | d_u(v) < 0\} \cup \{(z, v) | d_l(v) > 0\}$$

On G' we then have to redefine the arc capacity and node demands:

$$c' : A \to \mathcal{P}(\mathbb{R}), c'(a) = c'(u, v) = \begin{cases} \{x | x \in [0, d_u(u) - d_l(u)]\} & \text{if } v = z \\ \{x | x \in [0, d_u(v) - d_l(v)]\} & \text{if } u = z \\ \{x | x \in [c_l(a), c_u(a)]\} & \text{else} \end{cases}$$

The node demands now are

$$d'(v) = \begin{cases} d_l(v) & \text{if v is source, i.e. } d_u(v) < 0 \\ d_u(v) & \text{if v is sink, i.e. } d_u(v) > 0 \end{cases}$$

If we compute a solution of the flow problem on G' and restrict the flow to the arcs of G, we get the solution of the problem on G that respects the demand intervals:
We call the demands in the solution restricted to arcs of G d^s and have $d^s(u) = d'(u) + f'((u, z))$. The flow balance constraints in normal nodes (nodes with demand 0) do not change at all, since the only new node z is not in their neighborhood. All vertices that are sources or sinks (per definition thy cannot be both at the same time) have z as neighbor. The flow balance on a source u before and after restriction to G differs only by the flow on the artificial arc (u, z). This flow $f'((u, z))$ is in the capacity range $[0, d_u(u) - d_l(u)]$. Hence for sources u ($d'(u) < 0$) with $d^s(u) = d'(u) + f'((u, z))$ we can conclude for the demands $d^s(u)$ of the solution restricted to G

$$d_l(u) \leq d^s(u) = d'(u) + f'((u, z)) = d_l(u) + \underbrace{f'((u, z))}_{0 \leq f'((u,z)) \leq d_u(u) - d_l(u)} \leq d_u(u)$$

and as well for sinks ($d'(v) > 0$)

$$d_l(v) \leq d_u(v) - \underbrace{f'((z, v))}_{0 \leq f'((z,v)) \leq d_u(v) - d_l(v)} = d'(v) - f'((z, v)) = d^s(v) \leq d_u(v)$$

□

Definition 1.20. A *cyclic flow* within a feasible network flow f is a maximal flow $f' : C \to \mathbb{R}$ where $C \subseteq A$ is a cycle,

$$\sum_{a \in \delta^+(v) \cap C} f(a) - \sum_{a \in \delta^-(v) \cap C} f(a) = 0 \, \forall v \in C$$

for all arcs the flow directions in f' and f are the same, i.e.

$$f(a) \geq 0 \Rightarrow f'(a) \geq 0, \; f(a) \leq 0 \Rightarrow f'(a) \leq 0$$

and there is really a nonzero flow, i.e.

$$f'(a) \neq 0 \, \forall a \in C$$

.

If we can find any cyclic flow f' in a network flow f we say that f contains a flow cycle. If there is no cyclic flow, we say f is *acyclic* or an *acyclic flow* on G.

The conditions imply, that a cyclic flow is one that has no sources or sinks and the same value of flow on each arc.
The *size of a cyclic flow* we call the maximal absolute amount of flow contributing to the cycle. This means it is the value $|f'(a)|$ a cyclic flow f' has on an arc and the amount of flow that is really cycling. If we reduce flow on each arc of the cycle by the size of the flow there should be at least one arc with zero flow resulting since we said that it is maximal.

Definition 1.21. Given a graph $G = (V, A)$ like above, with $e \in A$ a specific arc of G. We call the problem of finding an acyclic flow

$$f : A \to \mathbb{R} \text{ with } f(e) \geq f'(e) \, \forall f' : A \to \mathbb{R} \text{ s.t.} f' \text{ is an acyclic flow on } G$$

the *Acyclic Flowbound Problem*.

For describing the algorithms and manipulations of network flows we will need further definitions.

Definition 1.22. Assume we are in the situation that we have have a current flow f in the given network G and we have a set $S \subseteq A$ of arcs with a direction forward or backwards for each - for example a path from a source to a sink. We call the act of setting flow to

$$f'(a) = f(a) + x \, \forall a \in S : \text{ a is forward}, \, f'(a) = f(a) - x \, \forall a \in S : \text{ a is backward}$$

in a way that f' again is a feasible flow for a (maybe different) set of flow demands $d' : V \to \mathbb{R}$ *augmenting a value x on S*.
If the demand values for which the flow is acyclic do not change, the set of arcs we augment have to be a cycle. In this case we also call this shifting flow on a cycle in G.

Proposition 1.23
The maximum flow value on the maximized arc e in the Edge-Maximizing Acyclic Flow Problem is the same as the maximal flow value if only cyclic flow on cycles containing e is forbidden.

This proposition is stating that only a small fraction of the cycles in the network are important. It is not necessary to add the acyclicity constraints on all cycles - they do only make a difference if the arc we maximize is contained in the cycle. If we have any feasible flow in a network, then we also have an acyclic flow there. We can just shift flow along cycles until one arc has zero flow until the flow is acyclic. The proof does the same with cycles not containing the maximized arc, reasoning these cycles do not impact the optimal solution on the maximized arc.

Proof. Let F_{acyc} be the set of flows that are completely acyclic and F^e_{cyc} the flows that can have cyclic flow on cycles not containing e. Since $F_{acyc} \subseteq F^e_{cyc}$ (i.e. the partially cyclic flow problem is a relaxation of the acyclic problem) we always have $f_{cyc}(e) \geq f_{acyc}(e)$ for all $f_{cyc} \in F^e_{cyc}, f_{acyc} \in F_{acyc}$ that are maximizing flow on e. We have to show that also $f_{cyc}(e) \leq f_{acyc}(e)$).
We show how we can produce a completely acyclic flow $f' \in F_{acyc}$ from any given flow $f_{cyc} \in F^e_{cyc}$.

This flow will have the same flow value on e: $f'(e) = f_{cyc}(e)$, so we prove for each flow $f \in F^e_{cyc}$ there is an acyclic flow with at least the same flow value on e.

Take a given flow $f \in F^e_{cyc} \setminus F_{acyc}$. Consider a cycle C where f has cyclic flow. On C all flow is going in the same direction. Choose the minimum flow value $\alpha(C) = \min(f(a)|a \in C)$ on an edge of C and shift flow by $-\alpha$ (this means decrease flow on every edge of C by α). On no arc the direction of flow was reversed, so there are no new cyclic flows created by this. But on at least one arc there is no flow at all now, so on the cycle C we have no cyclic flow anymore. The resulting flow is still a valid flow: On each node ingoing and outgoing flow is decreased by the same value α so flow balances still hold. The capacity constraints are still fulfilled because we know that no minimum flow bound is higher than 0 in our problem. The flow on the special arc e is unchanged, since e must not be contained in any cyclic flow.
We can continue this until there is no cyclic flow left and we transformed our flow into a flow $f' \in F_{acyc}$ with the same flow value on e as before. The algorithm terminates after less than $|A(G)|$ repetitions because every arc can be set to zero only once.
$\Rightarrow$ We conclude that

$$\max(f(e)|f \in F^e_{cyc}) \leq \max(f'(e)|f' \in F_{acyc})$$

and together with $F_{acyc} \subseteq F^e_{cyc}$ this gives us

$$\max(f(e)|f \in F^e_{cyc}) = \max(f'(e)|f' \in F_{acyc})$$

□

Now we can investigate the problem deeper. The Acyclic Flowbound Problem is obviously closely related to other flow problems though the special constraints we built make it more complicated. One way to deal with this is to look at related problems, which could give bounds on our problem or even the same solution in special cases.

1.3.1 Relation To Other Flow Problems/Relaxations

If we look for related problems, it is natural to just drop the constraints of our problem. In this case we could for example easily forget the contraint that flows have to be acyclic. If we do this, our problem becomes a special instance of a Min-Cost-Flow where we allow edge weights to be negative:

Definition 1.24. The *Minimum Cost Flow Problem* is the problem to determine a feasible network flow with the least possible cost c_f. The cost function $c : A \to \mathbb{R}$ (or often $c : A \to \mathbb{R}_{\geq 0}$) is defined on every arc, the cost of the flow is

$$c_f := \sum_{a \in A} |f(a)| \cdot c(a)$$

So for a given network and cost function we look for a feasible network flow f with

$$c_f \leq c_{f'} \, \forall f' : A \to \mathbb{R}$$

i.e. with minimum overall costs.

This problem is normally defined with nonnegative cost functions, like in [4] where Jack Edmonds and Richard Karp present their well known $O(V \cdot E^2)$ flow algorithm. At least cycles of negative weight should be excluded, since they lead to problems in the algorithm's subroutines like shortest path. Nevertheless there are algorithms that can handle negative cycles without a problem - but they will of course give a solution where as much flow as possible is just going around these negative cycles.
In our application we want to find upper and lower bounds on the possible flow over each arc, or equivalently the maximum possible flow on this arc in forward and backward direction. For this we set the weight on the arc e where we want to maximize flow to -1 and compute a Minimum Cost Flow in the graph. If e is contained in a cycle of G whose arcs all have a very high upper capacity bound, e will get a very high bound as well. The reason is that any cyclic flow over e can reduce the cost of the flow. Hence as much cyclic flow as possible will be used to obtain a Minimum Cost Flow.
So in most cases the bound computed with a Min-Cost Flow algorithm is not sharp. Still it gives an upper bound for the possible flow. One could ask how good this bound will be, and whether we can use it as an approximation. It turns out that the gap between the amount of flow on an arc e in an acyclic flow and in a possibly cyclic flow can be arbitrarily large:

Proposition 1.25
The gap between the maximum flow values $f(a)$ on an arc $a \in A$ in a normal and in an acyclic flow problem is unbounded.

Proof. In order to show this we look at the two examples:

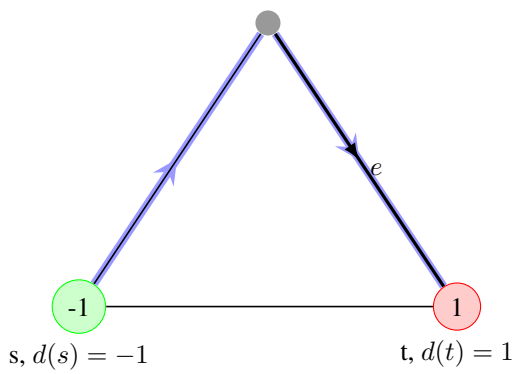

Figure 1: A simple network with acyclic flow

In figure 1 we see that the flow on the maximized arc e in the acyclic case has to be 1. If we allow cyclic flow as in figure 2 it could be any number - it is the minimum we choose as capacity for an arc in the network. Hence the relative gap is

$$\frac{f_{cyc}(e)}{f_{acyclic}(e)} = \frac{\text{min capacity on the cycle}}{1}$$

which is unbounded.
Figure 3 shows a graph where the flow on e in the acyclic case is 0. If we do not forbid cyclic flows, the flow on e will always be $\min_{a \in \text{ cycle } C} c_u(a)$ like in the first picture.

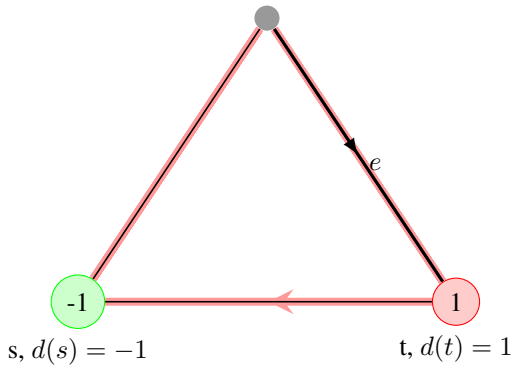

Figure 2: The same simple network with cycling flow that can pass e several times

So the gap in relative numbers

$$\frac{f_{cyc}(e)}{f_{acyclic}(e)} = \frac{\min_{a \in \text{ cycle } C}(c_u(a))}{0}$$

is not even defined!

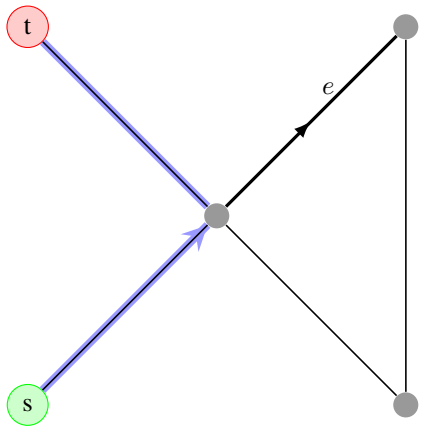

Figure 3: In the acyclic case there is no flow at all on the arc e in this network

□

As a different relaxation we could weaken the capacity constraints, but still insist on an acyclic flow that is maximum on an arc e. In the normal Max-Flow Problem dropping the capacity constraints reduces the problem to finding a shortest path between the source and the sink.
In our case, if there is only one source and one sink, we have to decide whether there is a simple path from the source to the sink that contains the maximized arc e. If there are more sources and sinks, we already have the problem of finding a set of simple paths containing e. Since the general constrained shortest path problem is $\mathcal{NP}$ hard this seems not to be promising and won't be disscussed here.

1.3.2 Computational Complexity

In this chapter we will discuss the computational complexity of the *Acyclic Flow Problem*. We show that Cycle-free Min-Cost-Flow with negative weights is $\mathcal{NP}$-complete, and that our problem is just a special case of this problem.

$\mathcal{NP}$-completeness is maybe the most important concept in the classification of combinatorial optimization problems (we will define it in a second). The $\mathcal{P} - \mathcal{NP}$-Problem is an open question for many years now. Many computer scientists conjecture that in fact $\mathcal{P} \neq \mathcal{NP}$ and the $\mathcal{NP}$-complete algorithmical problems are hard to solve.
The classical formal definition of the class $\mathcal{NP}$ in complexity theory uses formal languages and automata theory (namely Turing Machines). For this formal definition we refer to the textbook of Korte and Vygen [14]. If we accept to be a bit less formal we can define it the following way:

Definition 1.26. A combinatorial decision problem (a problem with the possible solutions "yes" and "no") is in the complexity class $\mathcal{NP}$ iff there exists an algorithm that, given the problem and an additional input (the certificate), computes "yes" in a polynomial bounded number of steps if and only if the instance is a "yes"-instance of the problem *and* the right certificate is given, otherwise false.
The set of problems where there exists an algorithm that always finds the correct answer in a polynomial (in the input size) bounded number of steps is called the complexity class $\mathcal{P}$.

Obviously all problems for which polynomial algorithms are known are in $\mathcal{NP}$, so $\mathcal{P} \subseteq \mathcal{NP}$. We do not know if polynomial algorithms (without the "certificate") exist for all problems in $\mathcal{NP}$. There are problems in $\mathcal{NP}$ for which no subexponential algorithm is known so far, for example the SAT problem. For some of them one can show that a polynomial algorithm for this problem already would imply a polynomial algorithm for all others.

Definition 1.27. A problem *P* is called $\mathcal{NP}$-complete if *P*$\in \mathcal{NP}$ and for every problem *P'* $\in \mathcal{NP}$ there exists an algorithm with polynomial bounded running time if it can call an oracle for *P*, i.e. a blackbox algorithm solving *P* in a single step.

Stephen Cook proved that every problem in $\mathcal{NP}$ can be solved by a polynomial algorithm if the SAT (boolean satisfiability) problem can be solved polynomially [2]. Richard Karp presented the result that the hamiltonian cycle problem (and by this also the hamiltonian path problem) can be reduced to SAT ([12]). Thus these problems are $\mathcal{NP}$-complete.
We want to show that the acyclic minimum cost flow problem, of which the acyclic flowbound problem is a special case, is an $\mathcal{NP}$-complete problem. This justifies to deal with algorithms that either are not provable to have polynomial worst case running time or else are no exact algorithms but rather heuristics to find a weaker bound.

Definition 1.28. We call the problem of finding a Minimum Cost Flow as in Definition 1.24 in a given network G with the additional constraint that there is no cyclic flow the *Acyclic Minimum Cost Flow* Problem.
The corresponding decision problem is to decide the question if there can be any acyclic flow with total cost of at most x in the network. With binary search on the values of x we can solve any optimization problem by iteratively solving the corresponding decision problem - so since both have the same complexity, we can speak of $\mathcal{NP}$ completeness of the optimization problem as well .

Theorem 1.29
The *Acyclic Minimum Cost Flow Problem* with arbitrary arc weights is $\mathcal{NP}$ complete.

Proof. We have to show two things: The problem is in the complexity class $\mathcal{NP}$, and the problem is indeed $\mathcal{NP}$ hard. We do this via two lemmas:

Lemma 1.30
The Acyclic Minimum Cost Flow Problem is in the complexity class $\mathcal{NP}$

Proof. It is easy to see that the problem is in $\mathcal{NP}$. Given a valid solution to the decision problem we can check if it is a feasible network flow by just checking the flow balances on all nodes, which needs only linear time $\mathcal{O}(|V| + |A|)$. We can also easily check if the total cost is smaller or equal than demanded and that all capacities are respected by running over all arcs and checking them, thus this part is also linear. We can also check the acyclicity in polynomial time: On each of the arcs we can start a graph search, considering arcs only in their flow direction. If this graph search comes back to the original arc again, there is a cycle. If graph search does not find a way to the tail of the original arc, there is no cyclic flow on this arc. The graph search is also a linear algorithm. If we do this for each arc, the running time is still quadratic. □

Lemma 1.31
The Acyclic Minimum Cost Flow Problem is $\mathcal{NP}$-hard.

Proof. To show the $\mathcal{NP}$ hardness, we reduce the problem to the *undirected $s - t$-Hamiltonian Path Problem* which is well known as a standard example for $\mathcal{NP}$ completeness. Hamiltonian cycle is $\mathcal{NP}$ complete both in the directed and undirected case, see [12]. To find an $s - t$-hamiltonian path in a graph G for some vertices $s, t \in V(G)$ we add one extra vertex x and the edges (t, x) and (x, u) to G. This new graph has a hamiltonian cycle if and only if G has a hamiltonian path from s to t.

Given an instance of the $s - t$-Hamiltonian Path problem, we have to transform it into an instance of our problem. Then we show that the reduced problem is a yes-instance of the Acyclic Min Cost Flow decision problem if and only if the original instance of the Hamiltonian Path problem is a yes-instance. Further the reduction must be computable in polynomial time.

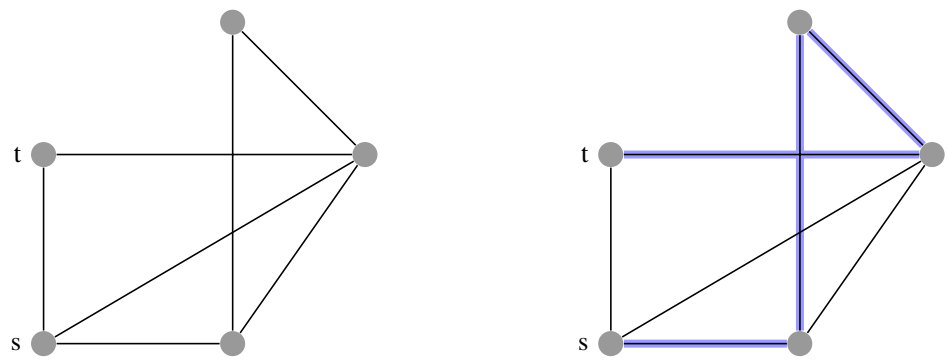

Figure 4: This graph has an $s-t$-hamilton path contained, marked in blue in the second picture

An $s-t$-Hamiltonian Path is a path in a network starting at node s and ending at node t that is visiting every node of G exactly once like in the example in figure 4.
This can only work in connected graphs, so we know that if n is the number of nodes in the graph, the Hamiltonian Path has to have size $n-1$. Our reduction is the following: Transform the $s-t$-Hamiltonian Path instance into an instance of a flow network where for each edge $\{uv\}$ there are two directed arcs $(u,v),(v,u)$, the arcs $a \in A$ all have capacities $c(a) = [0,1]$ and costs of -1 with the same underlying graph as before. The nodes s and t become the source and sink of the flow. The decision we ask for is whether there is a flow with total costs of at most $-(n-1)$ or not. (Figure 5 shows an example of such a transformation)

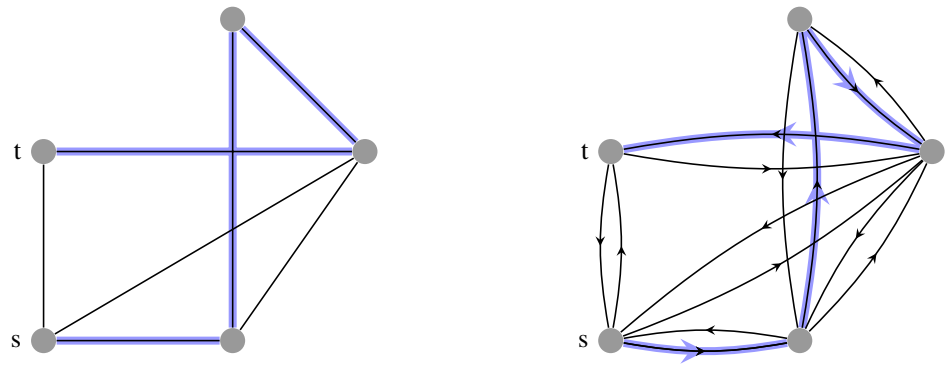

Figure 5: A graph with an $s-t$-hamilton path is transformed to an acyclic minimum cost flow instance with costs of -1 per arc. Hence in this example we find a minimum acyclic flow of cost $-4 = -n+1$

It is obviously a polynomial time transformation because only a linear number of flags and arcs is added to the network, while the edges are deleted when they get replaced with the new arcs.
It yields the same decision as the original Hamiltonian Path instance:

- $\Rightarrow$: Given an instance of the Acyclic Min Cost Flow which has a acyclic flow from s to t with cost $-n+1$. Flow can have a value of at most 1 on each arc, so each arc the flow is going through can only contribute up to -1 costs to the cost function. The flow could be split up into different paths from s to t, but we know that there are no cycles. That means on every of these paths every node is visited only once - if we see it twice we have closed a cycle. The cost of the flow is minus the average length of the paths of flow. So we can conclude there is at least one path with length $n-1$ that is seeing no node twice - hence it has to see every node exactly once, so we found (at least) one $s-t$-Hamiltonian Path and know it is a yes-instance of Hamiltonian Path too.

- $\Leftarrow$: Given a graph together with an $s-t$-Hamiltonian Path. In the transformed instance of Min Cost Flow we get immediately a feasible $s-t$-Flow by setting the flow value on all forward arcs on the given hamiltonian path equals 1. This flow also has total costs of $-(n-1)$.

$\square$

These two lemmas together show that our problem of finding an acyclic flow with minimum costs and arbitrary weight function is in general $\mathcal{NP}$-complete. $\square$

2 Models and Solving Approaches

In this section we describe different mathematical models that can be used to obtain bounds on the acyclic flow going over a specified arc of the given flow network (assumed there is any feasible network flow). We present two different MIP models of the acyclic flowbound problem. The first model uses node potentials and constraints to force flow to go in direction from higher to lower potentials. The second model uses decision variables for the direction of the flow and special acyclicity constraints on them for each cycle. They are different in size as the second MIP model explicitly uses constraints on the cycles to exclude cyclic flow from the solution. We show that it is not sufficient to have these constraints just on a cycle base (which would be a linearly small number). But we can prove conditions under which the acyclicity constraint of a cycle becomes redundant. Still there can be exponential many cycles in a network and the condition could be fulfilled on *cycle* of the network.
Furthermore we discuss an idea to solve the problem with a direct algorithm augmenting paths. We show by examples that path augmentation alone can not solve the problem, but the idea inspires a relaxation/heuristic algorithm which runs in polynomial time.

2.1 MIP Formulations

For many combinatorial Problems it is the best practical solution to formulate them as a Mixed Integer Problem and just solve this problem with modern MIP Solvers such as CPlex, Gurobi, XPRESS, SCIP etc. Often there are different possible formulations of the problem as MIP, which might result in very different running times due to numerical or algorithmical reasons. In our problem, it is easy to model the flow conservation and the ingoing and outgoing flow on vertices. Like described before, we can assign a negative weight to an arc variable and this way maximize flow over this arc by minimizing the overall cost. This would be the typical MIP formulation of a min cost flow:

$$\begin{aligned}
&\min \sum_{a \in A} w_a \cdot q_a \\
s.t. \quad &\sum_{a \in \delta^+(v)} q_a - \sum_{a \in \delta^-(v)} q_a = d_v && \forall v \in V \\
&c_l(a) \leq q_a \leq c_u(a) && \forall a \in A
\end{aligned}$$

This model still allows cyclic flow, so we have to find constraints to avoid cyclicity.

2.1.1 Model 1: Node Potentials

One idea (that is unfortunately not a linear problem formulation anymore) would be to set potentials on the nodes and allow only flow from higher to lower potential. This is quite close to the application of our problem in gas flow computation, where gas will only flow from places with high pressure to places with lower pressure in the network. So each vertex v would get a variable π_v for its potential, each arc a as before a variable q_a for the amount of flow. Then we add the constraint

$$q_a \cdot (\pi_v - \pi_w) \geq 0 \, \forall a = (v, w) \in A$$

We have to ensure that the values of the potentials are all different, or that there is no flow between nodes of the same potential. Otherwise any solution where all potentials are set to the same value would fulfill this constraint, regardless if it is acyclic or not. So in practice we could set a constant $\varepsilon > 0$ to describe the minimum distance between the potentials. A feasible solution now has to fulfill the constraint above on every arc. If we set up the constraint as above, we get the Mixed Integer Nonlinear Program

$$
\begin{aligned}
&\min \sum_{a \in A} w_a \cdot q_a \\
s.t. \quad & \sum_{a \in \delta^+(v)} q_a - \sum_{a \in \delta^-(v)} q_a && = d_v && \forall v \in V \\
& q_a && \leq c_u(a) && \forall a \in A \\
& -q_a && \leq c_l(a) && \forall a \in A \\
& -q_a \cdot (\pi_v - \pi_w) && \leq 0 && \forall a = (v, w) \in A \\
& (\pi_v - \pi_w)^2 && \geq \varepsilon && \forall v, w \in V \\
& q_a \in \mathbb{R} && && \forall a \in A \\
& \pi_v \in \mathbb{R} && && \forall v \in V \\
& d_a \in \{0, 1\} && && \forall a \in A
\end{aligned}
$$

We show that this flow indeed is an acyclic one:

Proposition 2.1
A flow which fulfills the constraints of the above mixed integer nonlinear program is always acyclic.

Proof. Assume a flow fulfilling the above constraints would have cyclic flow on a cycle C. Without loss of generality we label the vertices on the cycle from 1 to n and assume all arcs are directed forward on this cycle. This means there are arcs $a_1 = (v_1, v_2), a_2 = (v_2, v_3), \dots a_n = (v_n, v_1) \in C$ such that $q_{a_i} > 0 \, \forall i = 1, \dots, n$. Also we know from the constraints $q_a \cdot (\pi_v - \pi_w) \geq 0$ and $\pi_v - \pi_w \neq 0$ (Hence $\Rightarrow \pi_v \neq \pi_w) \, \forall v, w \in V$. So we conclude an ordering of the vertices potentials: $q_{a_1} \cdot (\pi_{v_1} - \pi_{v_2}) \geq 0 \Rightarrow \pi_{v_1} > \pi_{v_2}$ and so on. This yields a sequence $\pi_{v_1} > \pi_{v_2} > \dots > \pi_{v_n} > \pi_{v_1}$, which is a contradiction. $\lightning$ $\square$

However, solving nonlinear mixed integer programs comes with many difficulties, so we will use indicator constraints or an equivalent Big-M constraint to get a mixed integer formulation avoiding these difficulties. For this we replace

$$q_a \cdot (\pi_w - \pi_v) \leq 0 \, \forall a = (v, w) \in A$$

and

$$(\pi_v - \pi_w)^2 \geq \varepsilon \, \forall v, w \in V$$

Instead we introduce a new binary variable ρ_{vw} to indicate the flow direction on $a = (v, w)$, and say this variable has to be 1 if flow is from v to w and 0 if it is from w to v. Also if ρ_{vw} is 1 the potential π_v of v has to be greater than the potential π_w of w and the other way around. We can explicitly formulate the implications of these variables as indicator constraints. An indicator constraint in a mixed integer program consistst

of a binary variable and a linear constraint. The linear constraint is only active if the binary variables value is 1, it is ignored if it is 0. Technically it is no problem to take the negated variable to change when the constraint is turned on and off, so depending on the solution value of the binary variable different constraints can be turned on or off. The implications we want to model here are the following:

$$\begin{aligned}
&\rho_{vw} = 1 && \Rightarrow \pi_v \geq \pi_w + 1 \\
&\rho_{vw} = 0 && \Rightarrow \pi_w \geq \pi_v + 1 \\
&\rho_{vw} = 1 && \Rightarrow q((v,w)) \geq 0 \\
&\rho_{vw} = 0 && \Rightarrow q((v,w)) \leq 0 \\
&\rho_{vw} \in \{0,1\}
\end{aligned}$$

In order to get a pure mixed integer program we can reformulate these indicator constraints as so called Big-M-constraints. These constraints are constructed the following way: Each linear inequality can be transformed into a "≤ 0" -inequality. Then we add or subtract the binary variable (or the negation) on the right hand side of this transformed inequality multiplied by a big constant. This constant has to be just big enough to make sure the inequality automatically always holds if the binary variable used is 1. If on the other hand the binary variable is 0 then the inequality really has to be fulfilled. This way we can model the implications above as actual MIP constraints:

$$\begin{aligned}
&\pi_v - \pi_w && \geq N(\rho_{vw} - 1) + 1 \\
&\pi_v - \pi_w && \leq N\rho_{vw} - 1 \\
&q((v,w)) && \geq M(\rho_{vw} - 1) \\
&q((v,w)) && \leq M\rho_{vw} \\
&\rho_{vw} \in \{0,1\}
\end{aligned}$$

where $N \in \mathbb{R}$ and $M \in \mathbb{R}$ are sufficient big constants. Sufficient big in this case could mean we choose the values

$$N = |V|$$

$$M = \sum_{v \in V} |d(v)|$$

With all this the complete Mixed Integer Program is:

$$
\begin{aligned}
&\min \sum_{a \in A} w_a \cdot q_a \\
s.t. \quad & \sum_{a \in \delta^+(v)} q_a - \sum_{a \in \delta^-(v)} q_a && = d_v && \forall v \in V \\
& q_a && \leq c_u(a) && \forall a \in A \\
& q_a && \geq c_l(a) && \forall a \in A \\
& \pi_v - \pi_w && \geq N(\rho_{vw} - 1) + 1 && \forall a = (v, w) \in A \\
& \pi_v - \pi_w && \leq N\rho_{vw} - 1 && \forall a = (v, w) \in A \\
& q_a && \geq M(\rho_{vw} - 1) && \forall a = (v, w) \in A \\
& q_a && \leq M\rho_{vw} && \forall a = (v, w) \in A \\
& q_a \in \mathbb{R} && && \forall a \in A \\
& \pi_v \in \mathbb{R} && && \forall v \in V \\
& d_a \in \{0, 1\} && && \forall a \in A \\
& \rho_{vw} \in \{0, 1\} && && \forall a = (v, w) \in A \\
& N && = |V| && \\
& M && = \sum_{v \in V} |d(v)| &&
\end{aligned}
$$

2.1.2 Model 2: Acyclicity Constraints On All Cycles

Another idea to solve the Acyclic Flow Bound Problem is the following: Every arc has a direction and the sign of the flow on this arc tells us in which direction flow is send over this arc. Again we introduce variables $\rho_a \in \{0, 1\}$ for each arc $a \in A$ that indicate the direction of flow and are coupled with the flow variables:

$$
\begin{aligned}
\rho_a = 1 \Rightarrow q_a \geq 0 \\
\rho_a = 0 \Rightarrow q_a \leq 0
\end{aligned}
$$

These indicator constraints can be handled the way they are by standard MIP solvers, but to make them real MIP constraints we may formulate them as follows (with M a constant sufficient big, e.g. $M = \sum_{v \in V} |d_v|$) :

$$
\begin{aligned}
q_a + M \cdot (1 - \rho_a) \geq 0 \\
q_a - M \cdot \rho_a \leq 0
\end{aligned}
$$

In addition to decision variables for the flow direction, we have to add constraints to avoid cycles. If the flow is acyclic, each cycle should have at least two arcs with opposite directions. In other words, there has to be one arc in forward direction and one in backward direction, so with $n := |C|$ we get as constraint:

$$
1 \leq \sum_{a \in C \text{ forward}} \rho_a + \sum_{a \in C \text{ backward}} 1 - \rho_a \leq n - 1
$$

Let us formulate this in a slightly different way to get only one sum: For each cycle of size $C_n = C_l + C_m$ let C_m be the number of arcs directed forward and C_l the number

of arcs directed backward. We define that always $C_l \leq C_m$, so forward is defined as the direction that more arcs are pointing towards (left or right would only make sense in a planar embedding). Then we get the constraint

$$1 - l \leq \sum_{a \in C} \rho_a \leq n - (l+1)$$

that forbids any cyclic flow on C. We will show later under which conditions such constraints also forbid cyclic flow on other cycles.
So our MIP formulation of the model is finally

$$\begin{aligned}
&\min \sum_{a \in A} w_a \cdot q_a \\
s.t. &\sum_{a \in \delta^+(v)} q_a - \sum_{a \in \delta^-(v)} q_a = d_v && \forall v \in V \\
&c_l(a) \leq q_a \leq c_u(a) && \forall a \in A \\
&q_a + M \cdot (1 - \rho_a) \geq 0 && \forall a \in A \\
&q_a - M \cdot \rho_a \leq 0 && \forall a \in A \\
&1 - l \leq \sum_{a \in C} \rho_a \leq n - (l+1) && \forall \text{ cycle } C \in G \\
&q_a \in \mathbb{R} && \forall a \in A \\
&\rho_a \in \{0, 1\} && \forall a \in A
\end{aligned}$$

or if we normalize the MIP to only $\leq$ inequalities:

$$\begin{aligned}
&\min \sum_{a \in A} w_a \cdot q_a && && (1) \\
s.t. &\sum_{a \in \delta^+(v)} q_a - \sum_{a \in \delta^-(v)} q_a && = b_v && \forall v \in V && (2) \\
&q_a && \leq c_u(a) && \forall a \in A && (3) \\
&-q_a && \leq c_l(a) && \forall a \in A && (4) \\
&-q_a - M \cdot (1 - \rho_a) && \leq 0 && \forall a \in A && (5) \\
&q_a - M \cdot \rho_a && \leq 0 && \forall a \in A && (6) \\
&1 - l - \sum_{a \in C} \rho_a && \leq 0 && \forall \text{ cycle } C \in G && (7) \\
&\sum_{a \in C} \rho_a + (l+1) - n && \leq 0 && \forall \text{ cycle } C \in G && (8) \\
&q_a \in \mathbb{R} && && \forall a \in A && (9) \\
&\rho_a \in \{0, 1\} && && \forall a \in A && (10)
\end{aligned}$$

2.2 The Number of Cycles we have to forbid

The model above has one obvious problem we have to deal with: Since it has a constraint for every cycle of G and the number of cycles in a graph might be exponential, even setting up the model would need exponential running time.
So do we really need every cycle? Or is it enough to just forbid cyclic flow on a small subset (e.g. a cycle base) of the cycles of G?

The figures 6 and 7 show that a cycle base works on the easiest possible example, but fails if the problem gets just a bit more complicated:

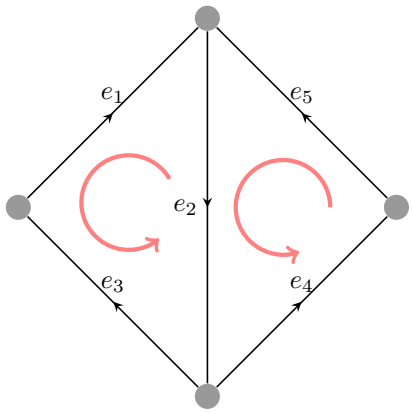

Figure 6: Here it suffices to forbid the 2 interior cycles marked with red in order to obtain acyclicity

Why are the constraints of a cycle base sufficient in this case of figure 6?
We see this immediately when we look at all acyclicity constraints for the 3 cycles contained in the graph (x_i being the binary direction variables of e_i, $i \in \{1 \dots 5\}$):

$$\begin{aligned} 1 \leq x_1 + x_4 + x_5 \leq 2 \\ 1 \leq x_2 + x_3 + x_5 \leq 2 \\ -1 \leq x_1 - x_2 - x_3 + x_4 \leq 1 \end{aligned} \qquad \iff -2 \leq -x_2 - x_3 - x_5 \leq -1$$

The last inequality is exactly the sum of the two inequalities above. Hence if two are fulfilled, the last one is automatically induced by them, it is a redundant constraint.

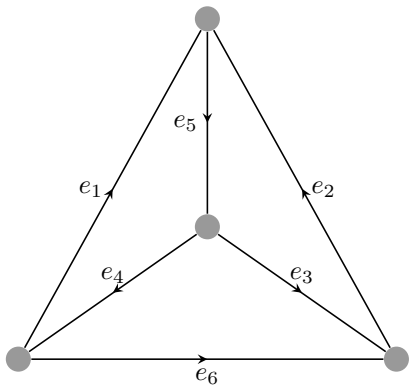

Figure 7: In this tetrahedron graph **no** cycle bases' acyclicity constraints can enforce acyclic flow on the whole graph

For figure 7 the situation is more complex, though there is only one more arc in the graph. But the number of cycles grows exponentially, so we get 7 cycles in this case.

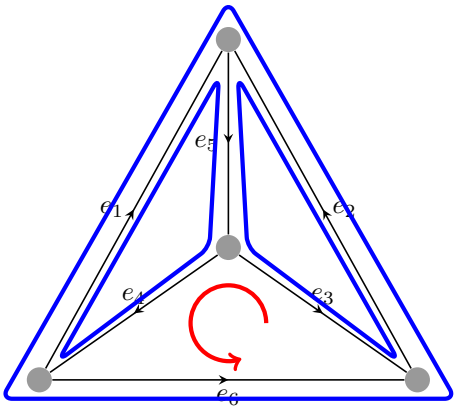

Figure 8: The example tetrahedron graph with the blue marked cycles where acyclicity constraints can not forbid cyclic flow on the marked red cycle

Their inequalities are:

$$
\begin{aligned}
1 \leq x_1 + x_4 + x_5 \leq 2 \\
1 \leq x_2 + x_3 + x_5 \leq 2 \\
-1 \leq x_3 - x_4 - x_6 \leq 0 \\
-1 \leq x_1 - x_2 - x_6 \leq 0 \\
-1 \leq x_1 - x_2 - x_3 + x_4 \leq 1 \\
0 \leq x_1 + x_3 + x_5 - x_6 \leq 2 \\
1 \leq x_2 + x_4 + x_5 + x_6 \leq 3
\end{aligned}
$$

We could choose the three cycles $(e_1 e_5 e_4)$, $(e_2 e_5 e_3)$, $(e_3 \overline{e_6}\, \overline{e_4})$ as a cycle base, these cycles induce the first three constraints. By combining the first and second, we directly get the fifth inequality $-1 \leq x_1 - x_2 - x_3 + x_4 \leq 1$ as above (this one is of course still redundant). But if we add inequality 3, the result is

$$
\begin{array}{cccccc}
 & -1 \leq & x_1 - x_2 & -x_3 + x_4 & \leq 1 \\
+ & -1 \leq & & x_3 - x_4 & -x_6 \leq 0 \\
= & -2 \leq & x_1 - x_2 & & -x_6 \leq 1
\end{array}
$$

But the acylicity constraint of the cycle $(e_1 \overline{e_2}\, \overline{e_6})$ is $-1 \leq x_1 - x_2 - x_6 \leq 0$ instead. The resulting constraints are not sharp, because the left hand side and the right hand side depend on the length of the cycles, which changes while we combine them.
Due to symmetry in this complete graph it does not matter which cycles we choose for the cycle base. In fact there is no combination of three cycles that could completely exclude cyclic flow solutions from the solution space.

Proposition 2.2
Given two acyclicity constraints $cons_1$ and $cons_2$ of cycles C_1 and C_2 in G. The acyclicity constraint of the joint cycle $C = C_1 \cup C_2 \setminus \{a \in A | a \in C_1 \wedge a \in C_2\}$ is the sum of $cons_1$ amd $cons_2$ if and only if C_1 and C_2 have exactly one arc in common (and this arc is directed differently in both cycles/has different sign in both constraints).

This proposition shows that many of the acyclicity constraints in our model may be redundant and could be omitted. But still the remaining constraints are exponential in the size of G.

Proof. Assume we are given two cycles C_1, $C_2 \in G = (V, A)$ with exactly one arc $g \in A$ contained in both cycles, and the direction of the cycles such that g is forward in one cycle and backward in the other cycle. Without loss of generality we assume it is forward in C_1 and backward in C_2, i.e. x_g has a positive sign in $cons_1 : lhs_1 \leq \cdots \leq rhs_1$ and a negative sign in $cons_2 : lhs_2 \leq \cdots \leq rhs_2$, which are the acyclicity constraints of C_1 and C_2.
$\Rightarrow$: The acyclicity constraints $lhs \leq \cdots \leq rhs$ are constructed such that always $rhs - lhs = l - 2$, l being the length of the cycle the constraint belongs to. Consider two cycles C_1 and C_2 and let $J \subset A = \{a \in A | a \in C_1 \wedge a \in C_2\}$ be the set of arcs that are contained in both cycles, $j := |J|$.
If we sum up the constraints $cons_1$ and $cons_2$ of the cycles C_1 and C_2 we obviously get another constraint, say $cons_3 : lhs = lhs_1 + lhs_2 \leq \cdots \leq rhs_1 + rhs_2 = rhs$. The length of the joint cycle C is $l = l_1 + l_2 - 2 \cdot j$. We have

$$rhs - lhs = rhs_1 + rhs_2 - lhs_1 - lhs_2 = \underbrace{rhs_1 - lhs_1}_{l_1 - 2} + \underbrace{rhs_2 - lhs_2}_{l_2 - 2} = l_1 + l_2 - 4$$

It holds

$$l_1 + l_2 - 4 = l_1 + l_2 - 2j - 2 = l - 2 \iff j = 1$$

This means that only if C_1 and C_2 have exactly one arc in common (i.e. $j = 1$), the necessary condition $rhs - lhs = l - 2$ is fulfilled to get the new acyclicity constraint.
$\Leftarrow$: Now we have to show that we get a sharp acyclicity constraint every time we sum up two acyclicity constraints that have exactly one variable in common, and this has different sign in both constraints (so it cancels out). Let $cons_1$ and $cons_2$ be chosen this way and $C_3 = C_1 \cup C_2 \setminus (C_1 \cap C_2)$ be the resulting cycle, then we get:

$$\begin{array}{rrl}
 & 1 \leq & \sum_{a \text{ forward in } C_1} q_a + \sum_{a \text{ backward in } C_1} (1 - q_a) \leq l_1 - 1 \\
+ & 1 \leq & \sum_{a \text{ forward in } C_2} q_a + \sum_{a \text{ backward in } C_2} (1 - q_a) \leq l_1 - 1 \\
= & 2 \leq & \sum_{a \text{ forward in } C_3} q_a + \sum_{a \text{ backward in } C_3} (1 - q_a) + \underbrace{q_a + 1 - q_a}_{a \in C_1 \cap C_2} \leq l_1 + l_2 - 2 \\
\iff & 1 \leq & \sum_{a \text{ forward in } C_3} q_a + \sum_{a \text{ backward in } C_3} (1 - q_a) \leq \underbrace{l_1 + l_2 - 2}_{l_3} - 1
\end{array}$$

which is exactly the acyclicity constraint of the cycle C_3. $\square$

There are still more redundant acyclicity constraints than these. Under certain conditions the acyclicity of smaller cycles implies the acyclicity of combined cycles even if there is more than one arc contained in both cycles.

Proposition 2.3
Given two cycles C_1, C_2 with a set of shared arcs $J := \{a \in C_1 | a \in C_2\}$ which we assume to have size greater than one ($j := |J| > 1$). If J is a path in the graph with no

sink or source vertex in the interior, the acyclicity of C_1, C_2 together implies acyclicity of the cycle $C_3 := C_1 \cup C_2 \setminus C_1 \cap C_2$.

Proof. We prove this via contradiction. If the common arcs J are a path like above, they have flow always in the same direction. Suppose C_1 and C_2 are acyclic, but C_3 is not. This means in both C_1 and C_2 there are arcs with different directions on the cycle, while they are all in the same direction on C_3. This means, the arcs with different direction have to be in the path J of shared arcs of C_1 and C_2. Hence within the path J there must be arcs which are directed oppositely to each other. This can only happen, if one of the vertices in the interior of J is a sink or a source. ↯ □

We can prove something more general by giving conditions that apply in more cases:

Proposition 2.4
Given a graph $G = (V, A)$, without loss of generality assumed to be connected, and a cycle $C \subset G$ with the following properties:

- $G \setminus C$ is disconnected
- Let G' be a connected component of $G \setminus C$
- G' has no source or sink nodes
- $G' \cup C$ consists of cycles whose direct sum is C

Then any acyclic orientation of the other cycles than C in $G' \cup C$ implies already the acyclicity of C. This means the acyclicity constraint of C is redundant.

Proof. Consider only the subgraph $G' \cup C$ and assume C had a cyclic orientation. In C there has to be a chordal path through G'. Since there are no sources or sinks in G', in fact there must be a *directed* path from one node $v_1 \in C$ to another node $v_k \in C$ going through G': starting from any arc incident to a node on C and one in G', we will always find another arc from the endnode in the same direction unless we run into a cycle or a source/sink node. But since neither exists in G', after a finite number of arcs we come to a node on C again. Thus we can always find a directed path in our setting. But there are two smaller cycles C_1 and C_2 if we divide C along this path. C is a directed cycle and the path is also directed $\Rightarrow$ so one of the two cycles has to be directed, which contradicts our assumption. ↯
Hence we conclude that acyclicity and absence of sources/sinks in the interior of a cycle make its acyclicity constraint redundant. □

There are some special cases where the above result can be very helpful and indeed reduce the number of needed acylicity constraints to polynomial size. This is not only relevant in theory, but also in practice: a real world gas network might have (at least partially) a planar graph representation.
Planar graphs fulfill the condition to find a cycle which is dividing the graph into different parts easily - each cycle in a planar graph embedding divides the graph into interior and exterior. If we have only one source and one sink and find an embedding where both are on the outer (unbounded) face of the graph, it is sufficient to have the acyclicity constraints only on the face cycles of the graph. The same holds for all areas or subgraphs without sources/sinks that are enclosed by a cycle: we only need small simple cycles and all other acyclicity constraints are redundant.
Still we have a problem with the sources and sinks. If there are many of them we still run into exponential numbers of acyclicity constraints.

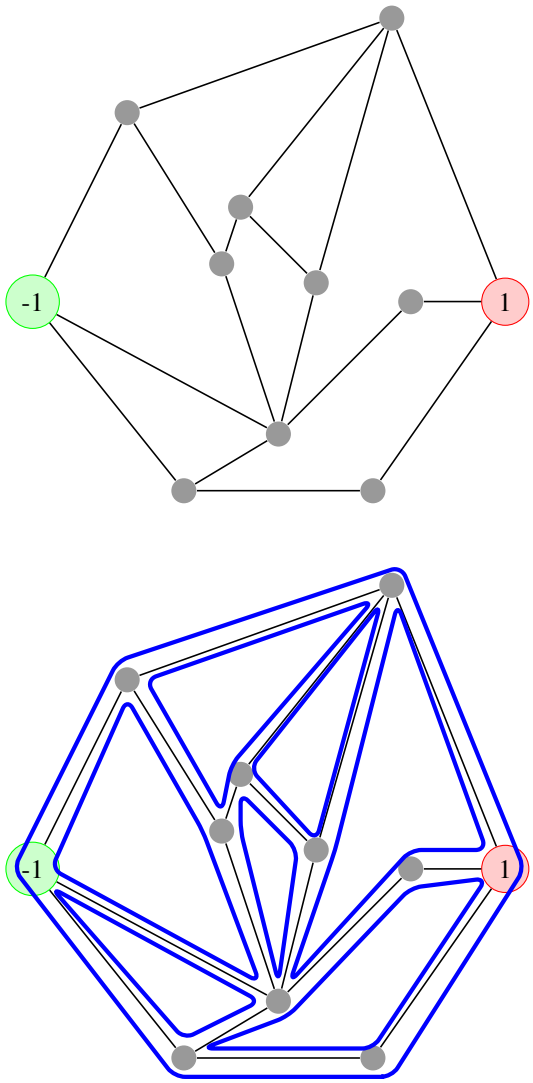

Figure 9: Flow on this example network can be forced to become acyclic by adding only the acyclicity constraints on the blue marked cycles

2.3 The Constrained Path Augmentation Model

The main idea in this chapter was to design an algorithm that uses path augmentation to saturate the given flow demands. The algorithm should try to push as much flow value as possible over the arc where flow is maximized, while having the constraint that the resulting flow is acyclic. Algorithms that use path augmentation to compute flows are well known and among the standard flow algorithms. The algorithm of Ford and Fulkerson [5] arising from their Max-Flow-Min-Cut theorem is based on source-sink-paths in the network as well as the improved algorithms of Edmonds and Karp [4] or Dinic [3].

This idea of path augmentation is conveniant and nice looking in the beginning. It is the base for a direct heuristic approach to the problem that can produce valid bounds

very fast. We also show the difficulties of choosing a path and illustrate with examples that only a relaxed version or either an algorithm with edge progressions instead of simple paths can work for the acyclic flowbound problem.

If we want to maximize flow over arc $e = (u, v)$ the first idea might be to just search a path from v to the sink t and another path from the sink s to u. Finding such a path can be done by graph search in the graph of augmentable arcs in linear time. Also the combined path of the two would always use the arc e. But it is not sufficient to just combine a path from source s to node u and one from node v to a sink t. Such a combination could produce cyclic flow, see for example figure 10.

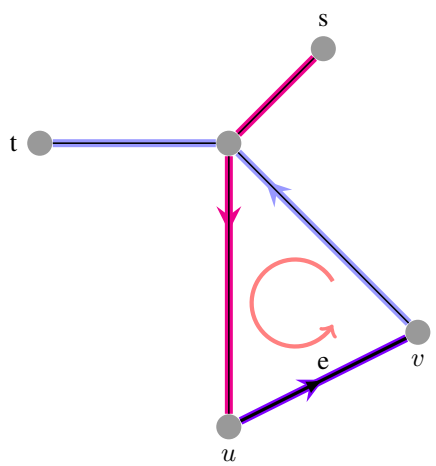

Figure 10: This small network shows that combining two independent paths (blue, magenta) from both endes of the maximized arc unto source and sink can lead to cyclic flow

The next straightforward idea is to use simple paths containing the arc we want to maximize. This means we would just forbid that the two paths we found have vertices in common. But it is also not sufficient to just use simple paths: it might happen that two different paths intersect in such a way that the combination of these paths produces a flow cycle (see figure 11).
In order to design an algorithm that uses such path augmentations, it is thus necessary to put acyclicity into the path choosing procedure as a constraint. The condition that flow on an arc e is maximized still has to be the objective. However, if we want to successively search and augment paths in the network it might be necessary to augment flow backwards - i.e. to change previously chosen paths due to an acyclicity issue of the current path. This is a problem for easy path searching methods. If we have to choose a next node for the path locally and do not know if we have to change a previous path we might have to branch on the decisions. It is even worse:
If the algorithm chooses to augment the wrong path it can happen that *no* simple $s - t$-path can be augmented - although there exists an acyclic flow with a higher flow value $f(e)$. Figure 12 shows such a situation where path augmentation is useless because in the first step a wrong path was chosen.
This example clearly shows the problems of a path based augmentation approach. A flow could block the wrong arcs while the next augmentation step possibly needs these

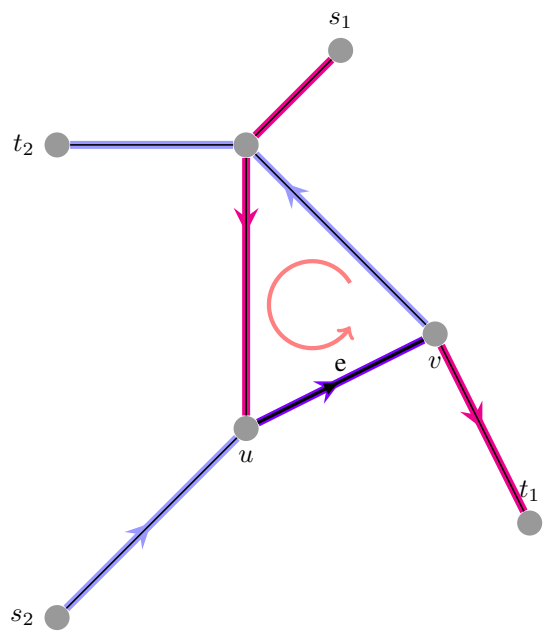

Figure 11: If we combine two simple $s - t$-paths (blue and magenta) going over e we might get a crossing of both, leading to cyclic flow

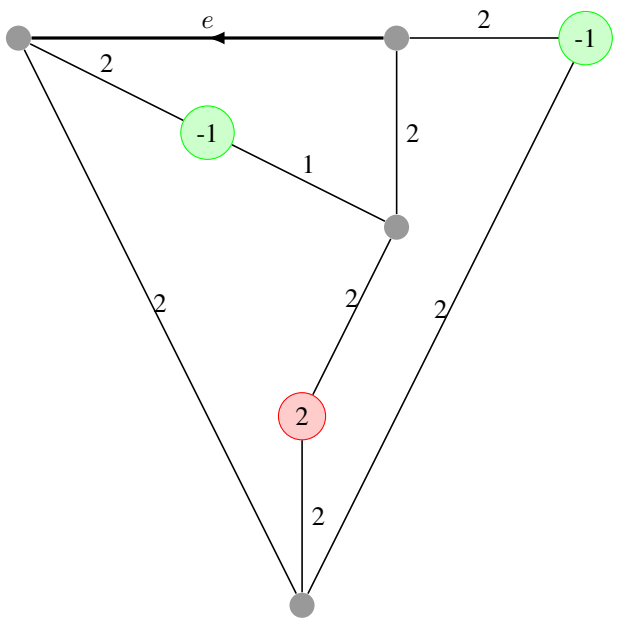

Figure 12: Example graph with capacities on all arcs and demand values at sources and sinks

arcs. We must give up the idea of finding simple paths and relax them to edge progressions. An edge progression can contain arcs and vertices more than once. Thus it can

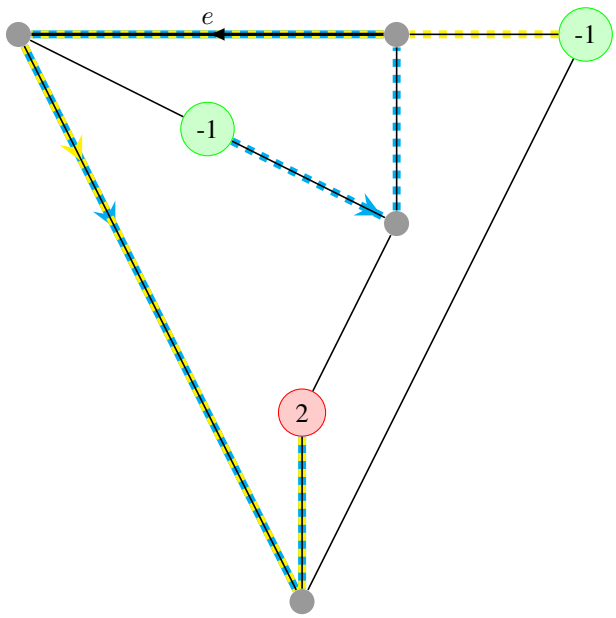

Figure 13: The optimum - the whole amount of flow is going over e while the flow is still acyclic in the whole network

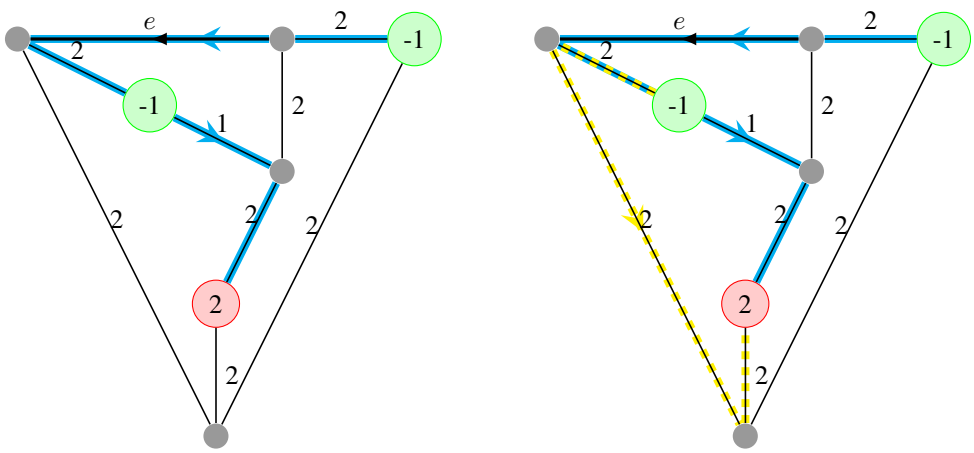

Figure 14: With the given capacities and the already augmented blue flow (left picture) there exists no simple path from the middle source to the sink such that flow on e could be increased. The best possibility is to augment the dashed yellow path which yields a flow value $f(e) = 1$ on the maximized arc. This is not optimal as figure 13 showed

do some kind of cyclic shifting on the flow. Since two flows with the same balances in the same network only differ by cyclic flow we can change the flow network to the state we desire. Whenever needed the optimal edge progression can do a cyclic shift and afterwards augment the actual path.

Here we will not try to deal with the problem of finding such an edge progression. It

is likely to be as difficult as the acyclic flowbound problem itself. Nevertheless we describe the algorithm in pseudocode (1) to show how such an algorithm would look like if we could find such paths or edge progressions and to use it as basis for a heuristic approach.

Algorithm 1 path based heuristic flow bound algorithm

```
function HEURISTIC(G = (V, A), e = (u, v) ∈ A, b : V → ℝ)
    define b' : V → ℝ, b'(v) ← 0∀v ∈ V                    ▷ temporary balances
    define f : A → ℝ, f(v) ← 0∀a ∈ A
    while ¬(b + b' ≡ 0) do                ▷ while there are active source-sink-pairs
        if c(e) > f(e) then
            choose active source v ∈ V
            P ←ACYCLICEDGEPROGOVERE(v, e)
            if P ≠ ∅ then
                MAXAUGMENT(P)
                continue
            end if
        end if
        choose s ∈ V with b(s) + b'(s) < 0                ▷ source that is still active
        c_alt(e) ← c(e)                                    ▷ save old capacity value
        c(e) ← f(e)                                  ▷ make e blocked for a short time
        P ←ACYCLICEDGEPROG(s)                         ▷ try finding a path that avoids e
        c(e) ← c_alt(e)
        if P ≠ ∅ then
            MAXAUGMENT(P)
            continue
        else
            P ← ACYCLICEDGEPROG(s)    ▷ try to find any augmentable path, at
this point of time it will be decreasing the flow on e
            if P ≠ ∅ then
                MAXAUGMENT(P)
                continue
            else
                return problem infeasible
            end if
        end if
    end while
    return f(e)
end function
```

Proposition 2.5
The described algorithm 1 together with a function *acyclicEdgeProg()* resp. *acyclicEdgeProgOverE* that finds an augmentable edge progression (acyclicEdgeProgOverE using a specified arc e) whose augmentation is not causing any flow cycles computes the optimal bounds to the acyclic flowbound problem.

Proof. We assume that the blackbox algorithm for finding an acyclic edge progression is correct. An edge progression can use arcs more than once $\Rightarrow$ it can do cyclic shifting on every cycle in the same connected component: For every cyclic shift that

```
function MAXAUGMENT(P)
    s ← source node of P
    t ← sink node of P
    augval ← min(|s|, |t|)
    for all a ∈ P do
        arcMaxAugval ← max augmentable flow in direction of P on a
        augval ← min(augval, arcMaxAugval)
    end for
    for all a ∈ P do
        increase flow on a in direction of P by augval
    end for
    b'(s) ← b'(s)+augval
    b'(t) ← b'(t)−augval
end function
```

is necessary it chooses an augmentable path towards the cycle, then goes around the cycle and on the same way back. If there is no augmentable path unto a cycle in the same connected component there has to be a directed cut that is at its capacity bounds. But in this case this cycle (and everything behind the cut) cannot influence the acyclic path at all. If the algorithm can change the flow on every cycle, it can produce every acyclic flow that is possible (before augmenting). There is always a path if there is any acyclic flow with this flow value on e because every acyclic flow can be decomposed into flow on simple paths. We conclude that the optimal edge progression could detect the right $s - t$-path, produce the acyclic flow without this path and then augment. So if the algorithm does not find any edge progression, there is also no acyclic flow with a higher flow value on e. □

We have seen that augmentation algorithms could only work with a too powerful algorithm that shifts flow before augmenting on a path. The main problem is that a flow can unnecessarily block an arc that is needed in a later stage of the algorithm.
So the following idea for a heuristic algorithm comes in: Can we increase the capacity bounds of the arcs in a way such that they can only be blocked in the case when the acyclic flow bound is reached?

Yes: if we double the capacities there will be no more flows blocked if not necessary: Even in flows that do not require acyclicity an upper bound for flow on an arc e is the flow that can be sent from sources to one end of the maximized arc e as well as the flow that can be sent from the other end of e to sinks. This flow can be computed by the standard maximum flow algorithms where path augmentation works well. But on each part of the problem - maximum flow from sources to one end of e and maximum flow from the other end vertex of e to sinks - the capacity of each arc in the network can only be used one time. So combining these two would give twice the capacity at most. Thus we can conclude that a more restricted problem (like finding the acyclic bound) can not need more capacity than this on a single arc.

We will describe an equivalent variant of this idea in the next section more detailed.

2.4 A Simple Heuristic Approach

As we have seen, the computations of the MIP formulation with separation of the Acyclicity Constraints could be expensive in running time and ressource consumption. The solved problem is a relaxation of the real-world problem anyway, so we can as well think about different relaxations and heuristic approaches. This chapter will introduce a simple and easy to implement heuristic approach inspired by the idea of the former section, where we showed why path augmentation models do not work for the acyclic flowbound problem. This approach uses a graph transformation and computes a Minimum Cost Flow on the transformed graph. We show that we get a valid bound for the Acyclic Flowbound Problem from this transformed graph.

The main idea of this heuristic approach is to avoid that the same flow is cycling over and over again until it reaches the capacity bounds. The path augmentation of the section before could achieve this goal. But we need to relax the capacity bounds and by this we relax also the problem. The proof that it suffices to double the capacities relies on the idea of distinguishing the flow before and after the maximized arc, because in every part of this the capacities have to be respected. If we double capacities and compute flow in this network it could happen that flow in one part uses 1.5 times of an arcs capacity and the other only half of it. So the bounds get in fact tighter if we separate the two parts.
In order to achieve this we make two copies of the original graph and make the arc we maximize the bridge between the two. On one part of the graph we only have sources, on the other part we only have sinks. In order to make the problem feasible we introduce artificial arcs between the sinks and their counterparts in the other copy of the graph. These artificial arcs get a detention cost so they are only used when there is no other way. On this modified graph we obtain minimum costs by sending as much flow as possible over the maximized arc. We can add the flow in both parts in the end, obtaining a flow for the original graph that might be not acyclic and not respecting the capacities but has maximum flow value on the arc we maximize.
Let us describe the approach in a formal way.

The Graph Transformation

The following algorithm 2 describes how the graph is transformed to the new graph. It mainly splits the graph into two copies X and Y, sets costs and introduces links between both parts and the supersource and supersink.
Figure 15 shows as an example how the graph from the last section would be transformed by this algorithm
With this transformation and any algorithm for Minimum Cost Flow we can set up an algorithm for our problem. For the Maximum Flow and Minimum Cost Flow Problem in graphs there are many standard algorithms we could use. The classical Maximum Flow algorithm of Ford and Fulkerson [5] arising from their Max-Flow-Min-Cut theorem is based on augmenting flow on source-sink-paths in the network as well as the improved algorithms of Edmonds and Karp [4] or Dinic [3]. Algorithms for Maximum Flow can be used to first check if there is any feasible flow in the network. If there is no feasible flow we can give up at this point, while the flow problem on the transformed graph is always feasible by construction.
Deriving from the Max-Flow Algorithms there are many algorithms solving the Min-Cost-Flow Problem. Edmonds and Karp described a Successive Shortest Path Algo-

Algorithm 2 graph transformation

```
function MakeTransformedGraph(G = (V, A), e ∈ A)
    create empty graph G' := (V', A'), V' = A' = ∅
    create labels capacity A' → ℝ, cost A' → ℝ
    create s, t ∈ V                                   ▷ supersource and supersink
    create a_0 := (s, t) ∈ A'
    capacity(a_0) ← ∞
    cost(a_0) ← 2                                     ▷ highest arc cost in the network
    for all v ∈ V do                                  ▷ make two copies of each node
        create v_X, v_Y ∈ V'
        if v is source of G then
            create arc a := (s, v_X) ∈ A'
            capacity(a) ← supply(v)
        else if v is sink of G then
            create arc a := (v_X, t) ∈ A'
            create arc b := (v_Y, t) ∈ A'
            cost(a) ← 1
        end if
    end for
    for all a = (u, v) ∈ A do
        if a = e then    ▷ for the arc to maximize we only make a bridge, no copy
            create arc e' := (u_X, v_Y) ∈ A'
            capacity(e') ← capacity(e)
        else
            create arcs a_X := (u_X, v_X), a_Y := (u_Y, v_Y) ∈ A'
            capacity(a_X) ← capacity(a), capacity(a_Y) ← capacity(a)
            cost(a_X) ← 0, cost(a_Y) ← 0
        end if
    end for
    return G'
end function
```

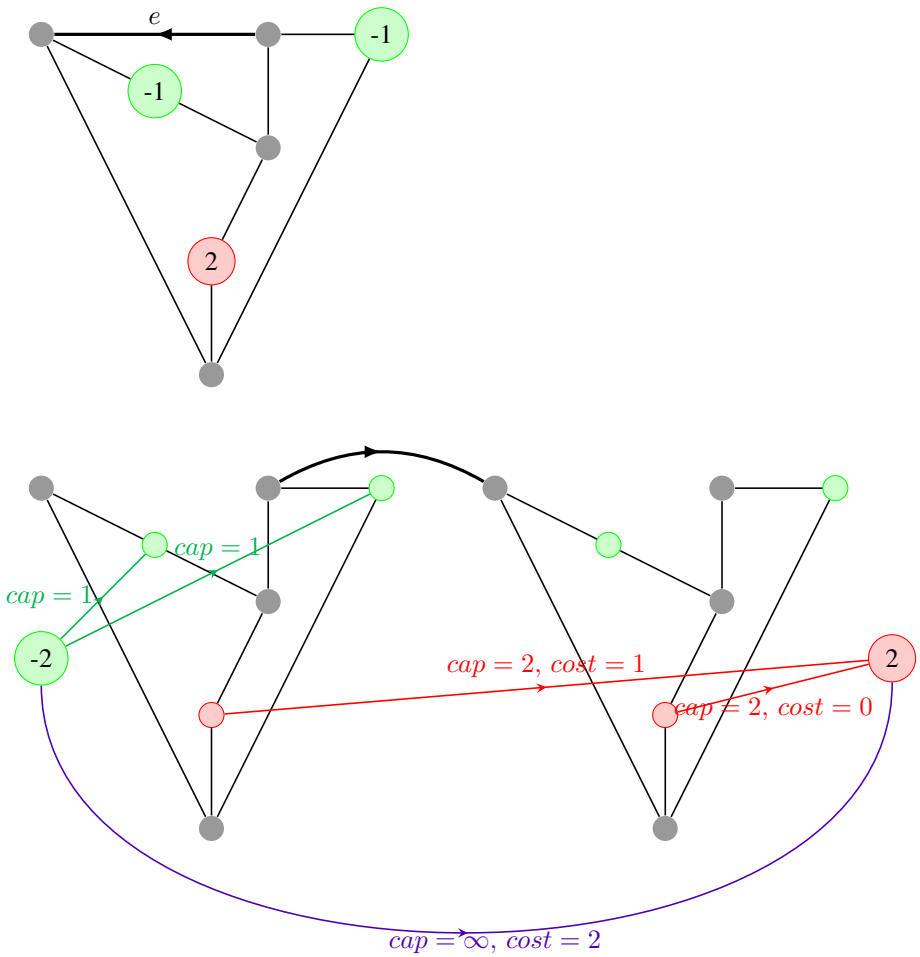

Figure 15: The transformation of the example network from the former section

rithm in [4]. Goldberg and Tarjan proposed the Minimum Mean Cycle Cancelling Algorithm [7] with runnning time of $\mathcal{O}(nm(logn)\min\{log(nC), mlogn\})$. Another algorithm is the Network Simplex by Orlin [15] (which was used in the computational study) There are a lot more algorithms available for Minimum Cost Flow.
Our algorithm mainly consists of a feasibility test, a transformation of the graph G and a Minimum Cost Flow computation on the transformed graph G', see algorithm 3.
We show the correctness of the algorithm:

Proposition 2.6
The heuristic algorithm 3 returns a valid upper bound for the possible acyclic flow on a given arc e in the network, or infeasible if there is no feasible network flow for the given flow balances at sources and sinks.

Proof. **Feasibility Test:** We have to show that the ordinary feasibility test with maximum flow is sufficient even for acyclic flow: If the flow network nomination (in- and

Algorithm 3 simple heuristic

```
function FlowBoundHeur(G = (V, A), e ∈ A)
    mf ← MaxFlow(G)
    if mf < nominated ingoing flow then
        return infeasible
    end if
    G' = MakeTransformedGraph(G, e)
    f ← MinCostFlow(G')
    ub ← f(e')
    backwardflow ← f(a_0)
    return ub - backwardflow
end function
```

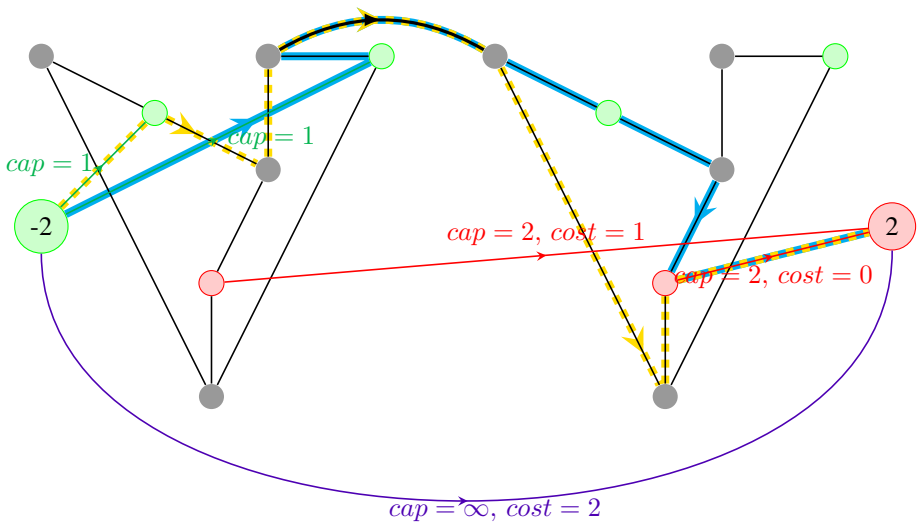

Figure 16: This flow has zero costs, is feasible in the heuristic and has the optimum value $f(e) = 2$

outflow) is infeasible, no algorithm for the maximum flow problem will find a flow that is fulfilling these balances completely. Then also for the stricter acyclic flow problem this is infeasible. The other direction is also true: if there is a feasible flow, there also is a feasible acyclic flow:
Take the maximum flow and choose one flow cycle within (If there is none we are done). On all arcs of the cycle the flow is going in the same direction in relation to the cycle. We can choose the minimum flow on the cycles arcs and reduce flow on each arc of the cycle by this amount. The resulting flow is still a feasible flow, but we removed the flow cycle. This can be done until the flow is acyclic. But if there is any acyclic flow, there also has to be an acyclic flow that is maximum on the arc we are interested in. So in order to test feasibility it is enough to run a ordinary maximum flow algorithm.

Relaxation: To prove that we get an upper bound with this heuristic we have to show

that the result we get is the maximum value of a relaxation of the problem.
If a flow is acyclic we can decompose it into flow on a set of simple paths. This means that the Acyclic Flowbound Problem could theoretically be solved by an algorithm augmenting simple paths (in the right order) where we put some extra constraints and objectives on these paths. These extra constraints have to forbid the closing of any flow cycle by the path as well as forcing the path to use arc e if possible. (Section 2.3 describes this idea and shows the difficulties of choosing the right path for augmentation.)
The heuristic algorithm is a relaxation of this constrained path augmentation model. It still forces all paths to go over e if possible (which might be much more since it is a problem with weaker conditions). We can reconstruct a flow in the original graph from the flow in the transformed graph by summing up the flow values on both copied arcs, assigning this flow to the original arc: $f(a) = f'(a_X) + f'(a_Y)$ and the assigned flow value of the bridge arc to the maximized arc of the original problem.
All the flow conservation constraints still hold after constructing the flow in the original graph from the two copies:

$$\begin{aligned}
&\sum_{a\in\delta^+(v)} q_a - \sum_{a\in\delta^-(v)} q_a && =d_v && \forall v \in V_X \\
+&\sum_{a\in\delta^+(v)} q_a - \sum_{a\in\delta^-(v)} q_a && =d_v && \forall v \in V_Y \\
=&\sum_{a\in\delta^+(v_X)\cup\delta^+(v_Y)} q_a - \sum_{a\in\delta^-(v_X)\cup\delta^-(v_Y)} q_a && =d_v && \forall v_X \in V_X, v_Y \in V_Y \\
=&\sum_{a\in\delta^+(v)} q_a - \sum_{a\in\delta^-(v)} q_a && =d_v && \forall v \in V
\end{aligned}$$

The flow balances at sources and sinks are set right by construction.

When no more flow can be sent over the maximized arc e the path augmentation model already gives a valid bound if the flow is feasible. But in fact this model can determine the amount of flow that is necessary to augment backwards on e in a path model.
We distinguish whether the flow is going over the bridge e, going to a sink node in already in part X of the transformed graph and thus having costs of 1 per unit, or is going from s to t directly with a cost of 2 per unit. The flow with costs of 2 cannot be sent over e in forward direction neither find any sink in part X of the graph. If there is a feasible flow in G (which we tested) the only possible way is that the remaining flow has to go backward over e. So we can reduce the flow on the bridge by this amount of flow directly going from s to t and the bound is still valid. (In algorithm 1 from the section before this would be the part in line 24 where flow is augmented backwards).

$\Rightarrow$ Since the problem is just a relaxation of the original problem/the constrained simple path model, the obtained bound is always weaker (in this case $\geq$) than the optimal bound for maximum acyclic flow. $\Rightarrow$ the heuristic is correct for finding upper bounds for the acyclic flow on an arc.

$\square$

So we have seen that we can compute an upper bound via the optimum for this relaxed problem with a fast standard algorithm for maximum flows that can be retranslated into the original problem. However, the capacity constraints of the original problem might

be violated after retranslation. We set the arcs capacities in part X and part Y both to the value of the original capacities. Thus it might happen that actual flow value in the original graph sums up to a higher value than the capacity allows (up to twice the capacity).

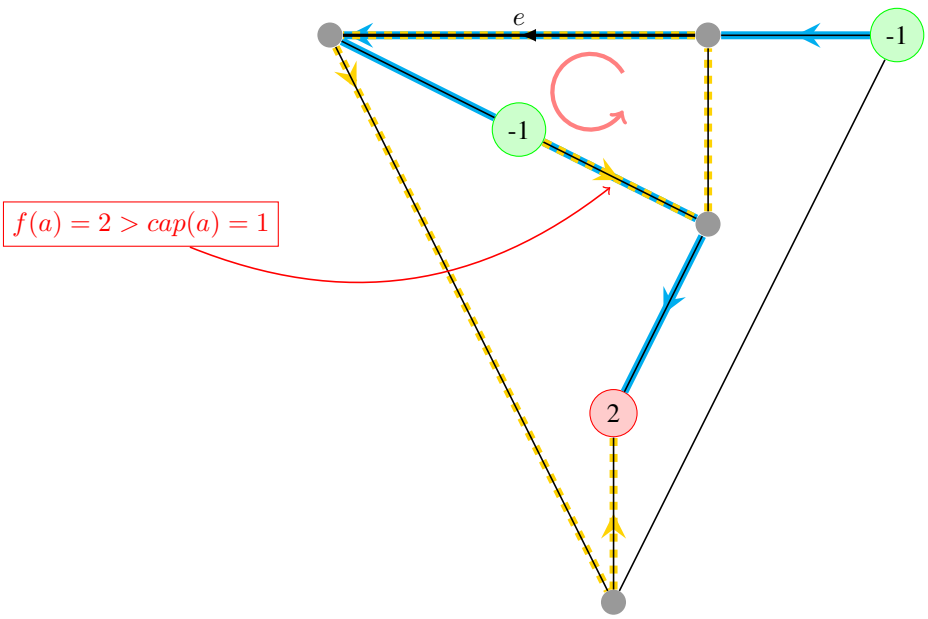

Figure 17: If we retransform the flow from figure 16 (which is feasible in the heuristic) to the original network we get a violated capacity constraint on the arc with $c = 1$. The flow is not acyclic anymore either.

Also we cannot guarantee the acyclicity anymore. It might happen that we have cyclic flow in our original graph G even if the flow in the transformed graph G' was acyclic. This happens when paths from the different parts X and Y of G' are crossing after they are brought back to the original graph G to build a solution there. For an example see figure 17 which is the retransformation of the example flow in figure 16.

3 Implementation and Computational Results

In this section we will discuss the practical implementations and present results of the computation. For relevant real-world problem sizes the exact models are too slow, even with separation of the acyclicity constraints. For this reason we also describe how to relax the problem in order to get useable (though in general not optimal) results for the flow bounds. We compare the bounds found by different approaches and with different settings (different networks and nominations) to see which approach is suited best to actually compute acyclic flow bounds in a network. We also describe in short how the standard bounds we used to compare the results to are computed.
At last we compare the impact of bounds and the number of fixed variables fed into the solver on the model sizes of the discretization.

3.1 Implemented Algorithms

We described the theoretical backgrounds of different algorithms to obtain flow bounds for our problem. The computational results of different implementations of them may vary widely, and the performance is influenced by lots of details regarding the machines, programming languages, data structures or just code quality. Nevertheless we want to give an overview what worked well in the implementation and what did not.

All algorithms where tested on a PC with instances of the *"gaslib"* gasnet test library. The library can be downloaded on `http://gaslib.zib.de/`. A detailed description of this problem library can be found in [10].

3.1.1 Cyclic Flow Bounds

To start a comparison we need a bound value to which we can compare. This value should be one that is fast and easy to compute, in some way sensible and not too far away from a realistic solution, and of course a valid upper bound. One of the first algorithmic problems that one might think of after seeing the flowbound problem is the normal minimum cost flow problem 1.24. Setting a negative weight on only one arc in the network will maximize flow on this specific arc. We described the connections with this problem in section 1.3.1.
We can set the maximal flow capacity on an arc to the natural upper bound "sum of all ingoing flow" if no tighter capacity bounds are given. This is a very easy condition derived from the acyclicity and can always be applied.
This model is also implemented in Lamatto. It is modeled as a mixed integer programm with only flow balance and capacity constraints. Binary directions are not contained in this model. Because this MIP model also served as the basis for the more specific MIP models it is a relaxation of all the others which just add some more constraints or variables.
We will refer to this model as "cyclic bounds" respectively "cyclic bounds + flowsum" in the tables below. We compare the bounds from the different algorithms to the bounds found with this approach.

3.1.2 MIP with Node Potentials and Binary Direction Variables

The MIP model described in 2.1.1 was implemented in Lamatto, using Gurobi as solver for this model. However,the two layers of indicator constraints seem to be a heavy problem of the model in practice. The following model was implemented:

$$
\begin{aligned}
&\min \sum_{a \in A} w_a \cdot q_a \\
s.t. \quad & \sum_{a \in \delta^+(v)} q_a - \sum_{a \in \delta^-(v)} q_a && = d_v && \forall v \in V \\
& q_a && \leq c_u(a) && \forall a \in A \\
& q_a && \geq c_l(a) && \forall a \in A \\
& \rho_{vw} = 1 && \Rightarrow \pi_v \geq \pi_w + 1 && \forall (v,w) \in A \\
& \rho_{vw} = 0 && \Rightarrow \pi_w \geq \pi_v + 1 && \forall (v,w) \in A \\
& \rho_{vw} = 1 && \Rightarrow q((v,w)) \geq 0 && \forall (v,w) \in A \\
& \rho_{vw} = 0 && \Rightarrow q((v,w)) \leq 0 && \forall (v,w) \in A \\
& q_a \in \mathbb{R} && && \forall a \in A \\
& \pi_v \in \mathbb{R} && && \forall v \in V \\
& d_a \in \{0,1\} && && \forall a \in A \\
& \rho_{vw} \in \{0,1\} && && \forall a = (v,w) \in A
\end{aligned}
$$

Integer node potentials imply the direction of the arc between the incident nodes. These binary direction variables again imply the sign of the actual flow on an arc. Indicator constraints are usually implemented via Big-M-constraints of the form

$$
\begin{aligned}
& \pi_v - \pi_w && \geq N(\rho_{vw} - 1) + 1 \\
& \pi_v - \pi_w && \leq N\rho_{vw} - 1 \\
& q((v,w)) && \geq M(\rho_{vw} - 1) \\
& q((v,w)) && \leq M\rho_{vw} \\
& \rho_{vw} \in \{0,1\}
\end{aligned}
$$

with sufficient big values for M and N. This model heavily relies on the Big-M constraints. For any MIP solver these indicator or Big-M constraints are difficult to handle, because the optimal vertex of the LP polyhedron can be quite far from the optimal mixed integer solution. Modern MIP solvers also allow to directly formulate and add indicator constraints for a problem and try to improve the branching on these variables constraints. This was done here: the indicator constraints were explicitly given to the solver which tries to set the M and N constants and handle the constraints as good as possible. Still the model suffers from the same problem: The vertices of the LP-relaxation polytope are quite far from the integral points which might be feasible in the MIP. After solving the (easy) linear program there there remains a lot of (computational hard) work for the branch and bound process left to do. So it is natural to not expect larger instances of the problem to be quickly solved by this model.

The model was tested on the smallest network named "gaslib-40" of the gaslib package. After running the algorithm for 5 seconds the optimal bounds for all arcs of the network were computed. The results were the same as with the Acyclicity Constraint (separation) Model, apart from numerical rounding differences.
On the medium size network "gaslib-135" of the gaslib package, this model already fails to deliver exact results. Even after running it on a desktop PC with 8 cores for more than a week just the first bound of the first arc was computed. The result for the even bigger gaslib582 network was similar. It was not possible to solve the problem with this model on a bigger network. To get some result and make the approach comparable we also look at a relaxation of the problem.

3.1.3 Relaxation of the Node Potential Model

For the included solver (Gurobi) one can set a limit to the processed branch-and-bound nodes. If no solution is found within this limit the dual bound at this stage still gives a bound on the maximum flow. This is a relaxation of the problem (the dual bound of a maximization problem is always greater or equal to the maximum $\Rightarrow$ the dual bound is a valid upper bound for the acyclic flowbound problem). In the tables below we refer to this relaxed model with "nodepot_x" if there is a nodelimit of x set.

3.1.4 MIP with Binary Direction Variables and separated Acyclicity Constraints

In section 2.1.2 we described a model using acyclicity constraints on the binary directions for each cycle that was found during the solving process. This model itself is independent of the implementation of the acyclicity constraints. But since a graph can have an exponential number of cycles, writing down all constraints of the model alone could take exponential time. The model redundancies we described in 2.2 are not enough to sufficiently reduce the model size - in fact it could happen that the conditions for 2.2 do not apply at all in certain networks (though we also did not proof that a clever generalization of the proposition couldn't do better). For example the condition of the proposition states that there have to be two cycles with exactly one arc in common. But there are of course example networks not containing any two cycles like this. (You could just subdivide every arc in a network. Then every two cycles have at least two arcs in common.)

So the implementation does not add all the acyclicity constraints explicitely. Instead, the MIP Solver gets a general constraint that demands that there is no cyclic flow in the network. When this constraint is checked and there is cyclic flow somewhere in the network, a new linear acyclicity constraint on the cycle with the detected cyclic flow is added. An Acyclicity Constraint Handler was implemented for this thesis in Lamatto with SCIP as MIP solver.

Like the other exact model with node potentials this model takes lots of time to compute the optimal solution. In a network there are huge numbers of arcs to be computed that yield similar subproblems, some of them being difficult and running very long. Like in the Node Potential Model we could only produce an exact solution for the "gaslib-40" network while the algorithm was running too long on the other gaslib test instances.

3.1.5 Relaxations of the Exact Separation Model

If we want to have fast answers or compute bigger networks the exact model with separation of acyclicity constraints does not work in practice. Nevertheless this model gives us an interesting relaxation of the acyclic flowbound problem for free:
We can limit the time or the number of nodes in the branch and bound tree for each arc in the network. If an optimal solution can be found with low effort, we get the optimum bound. If the situation is difficult and we run into the limit we just take the dual bound of the mixed integer program. This number is always an upper bound because it is always greater or equal to the optimum (which is the maximum flow).

With this algorithm we will always get some solution. This solution might be ∞. To get rid of the infinite values, we can combine it with a simple bound estimate. For example we can take the sum of all ingoing flow as upper bound for all arcs with a dual bound that is greater than this. Depending on the preset limit this dual solution could closer to or further from the optimum.
This relaxation was implemented with a limit on the nodes and is compared to others in the tables below. The tables refer to this relaxation as "sepa_X" for the exact MIP with separation of acyclicity constraints and a node limit of X in the branch and bound process of the MIP.

3.1.6 Acyclicity Constraints on a Cycle Base

This idea was the starting point of this thesis and implemented by Robert Schwarz some time before. It is like the former an incomplete application of the acyclicity constraints. It adds all its acyclicity constraints in the beginning. The advantages are easier implementation and hope for a faster running time since the solver does not need to add new constraints all the time.
We have seen in 7 that the acyclicity constraints of a small subset of the cycles (like a cycle base) do not suffice to forbid cyclic flow in the network. Still we can try to use them as a relaxation to the exact problem. This will already avoid cyclic flow on single cycles that are not touching each other. The tables show how the idea works out in practice.

3.2 Tables

3.2.1 Bound Improvements

gaslib-40	Time	Fixed Flows	Bounds tightened	avg improvement
cyclic bounds	< 1 s	21	[-48]	[-7825.0]
cyclic bounds+flowsum	< 1 s	21	-	-
cycle base	1 s	21	48	1372.4
heuristic	< 1 s	21	0	0
node potential	4 s	21	48	1372.4
sepa_all	112 s	21	48	1372.4

gaslib-582 warm_1111	Time	Fixed Flows	Bounds tightened	avg improvement
cyclic bounds+flowsum	1 s	365	-	-
cycle base				
heuristic	11 s	365	164	5047.2
node potential				
nodepot_60000	20 min			
nodepot_500000	132 min			
sepa_1000	50 min	382	107	3802.1
sepa_5000	159 min	382	180	3971.6
sepa_20000	281 min	382	264	4312.5
sepa_60000	702 min	382	278	4267.7

gaslib-582 cool_12	Time	Fixed Flows	Bounds tightened	avg improvement
cyclic bounds+flowsum	1 s	365	-	-
cycle base				
heuristic	11 s	365	164	4050.9
node potential				
nodepot_60000	19 min			
nodepot_500000	135 min			
sepa_1000	50 min	378	100	3025.2
sepa_5000	158 min	378	211	3525.6
sepa_20000	361 min	378	242	3648.3
sepa_60000	678 min	378	276	3390.3

gaslib-582 cold_100	Time	Fixed Flows	Bounds tightened	avg improvement
cyclic bounds+flowsum	$1s$	365	-	-
cycle base				
heuristic	$10s$	365	164	5157.4
nodepot_60000	19 min			
nodepot_500000	123 min			
sepa_1000	50 min	378	88	4265.5
sepa_5000	127 min	378	242	4508.9
sepa_20000	382 min	378	241	4526.3
sepa_60000	635 min	378	283	4200

gaslib-582 freezing_1	Time	Fixed Flows	Bounds tightened	avg improvement
cyclic bounds+flowsum	$1s$	365	-	-
cycle base				
heuristic	11s	365	148	6148.2
nodepot_5000	5 min/8 cores	365	0	0
nodepot_60000	18 min /8 cores	366	51	4123.2
nodepot_500000	127 min/8 cores	366	57	4224.7
sepa_1000	50 min	374	90	5405.4
sepa_5000	154 min	374	231	5104
sepa_20000	366 min	374	244	4929.8
sepa_60000	593 min	374	276	4777.6
sepa_300000	3135 min	374	295	4652.8

gaslib-582 mild_10	Time	Fixed Flows	Bounds tightened	avg improvement
cyclic bounds+flowsum	1 s	365	-	-
cycle base				
heuristic	11 s	365	164	4291.5
node potential				
nodepot_500000	135 min/8 cores			
sepa_1000	50 min	378	105	3271.3
sepa_5000	158 min	378	188	3329.5
sepa_20000	394 min	378	216	3144.6
sepa_60000	651 min	378	289	3593.3

The results show that in all cases the bounds get better with more time for the separation algorithm, while the exact models need too much time. The heuristics results are weak, but far better than setting the limit to only 1000 nodes. The bounds computed can be read in and used by the solver to improve building of other models like the discretization. The impact of the computed flow bounds on this model is shown in the tables below.

3.2.2 Model building impact

gaslib-40	Fixed Dirs(of 45)	nonlinear f.	variables (cont,discr)	constraints
cyclic bounds	24	57	1477 (948+529)	2009
cyclic bounds+flowsum	24	57	1477 (948+529)	2009
cycle base	26	57	955 (687+268)	1473
heuristic	24	57	1477 (948+529)	2009
node potential	25	57	955 (687+268)	1473
sepa	34	57	955 (687+268)	1473

If we compare the results on this small gaslib-40 network with its table of bound values there are two interesting points. In building the model there is no difference between the cyclic bounds that were computed with respect to the sum of ingoing flow and the ones were the sum of ingoing flow was not set as upper flow bound. Obviously these bounds are already used implicitely. Nevertheless it is worth using these natural bounds already in the bound computation, since the separation algorithms run much faster and give better results when they are given these bounds before.
The second point is that the number of fixed flow directions in the last two entries increases a lot, although the exact algorithms both found the same optimal solution. The reason is a single numerical difference. It might be useful to round values which are extremely close to zero (like this 1^{-10}) to get more fixations for the model.

gaslib-582 cold_100	Fixed Dirs(of 609)	nonlinear f.	variables (cont,discr)	constraints
cyclic bounds+flowsum	440	241	5727 (4113+1614)	8216
heuristic	440	241	5675 (4087+1588)	8162
sepa_1000	467	237	5471 (3977+1944)	7930
sepa_5000	467	237	5185 (3834+1351)	7641
sepa_20000	467	237	5185 (3834+1351)	7640
sepa_60000	469	237	4903 (3693+1210)	7339

gaslib-582 freezing_1	Fixed Dirs(of 609)	nonlinear f.	variables (cont,discr)	constraints
cyclic bounds+flowsum	438	241	5757 (4128+1629)	8246
heuristic	438	241	5701 (4100+1601)	8188
sepa_1000	466	237	5443 (3963+1480)	7905
sepa_5000	469	237	5171 (3827+1344)	7632
sepa_20000	466	237	5099 (3791+1308)	7553
sepa_60000	472	237	4747 (3615+1132)	7192

gaslib-582 cool_12	Fixed Dirs(of 609)	nonlinear f.	variables (cont,discr)	constraints
cyclic bounds+flowsum	450	241	5427 (3963+1464)	7926
heuristic	450	241	5371 (3935+1436)	7868
sepa_1000	480	237	4993 (3738+1255)	7464
sepa_5000	480	237	4805 (3644+1161)	7268
sepa_20000	480	237	4803 (3643+1160)	7266
sepa_60000	482	237	4655 (3569+1086)	7099

gaslib-582 mild_10	Fixed Dirs(of 609)	nonlinear f.	variables (cont,discr)	constraints
cyclic bounds+flowsum	458	241	5569 (4034+1535)	8077
heuristic	458	241	5491 (3995+1496)	7997
sepa_1000	488	237	4991 (3737+1254)	7469
sepa_5000	490	237	4871 (3677+1194)	7344
sepa_20000	492	237	4709 (3596+1113)	7163
sepa_60000	494	237	4675 (3579+1096)	7117

gaslib-582 warm_1111	Fixed Dirs(of 609)	nonlinear f.	variables (cont,discr)	constraints
cyclic bounds+flowsum	458	241	5673 (4086+1587)	8183
heuristic	458	241	5589 (4044+1545)	8097
sepa_1000	487	237	4865 (3674+1191)	7335
sepa_5000	492	237	4745 (3614+1131)	7212
sepa_20000	494	237	4433 (3458+975)	6886
sepa_60000	496	237	4371 (3427+944)	6823

For the big networks we are only able to compare the results of heuristics because the exact algorithms take far too long for solving. The tables show that improved bounds yield a smaller model. The more nodes we allow in the separation process, the tighter the bounds. With tighter bounds we get a smaller model. Hence it could in fact make sense to compute such bounds to accelerate the solving process in the end. For this there are even more improvements one could try to make: In a graph there can be induced paths (i.e. neighboring vertices with degree 2). On a path the flow value has to be the same on each edge, so also the bounds have to be the same.
Hence it is sufficient to compute the bound on one arc of a path and automatically set the same bound on the other arcs of this path.
Another improvement could be to immediately write the computed bounds into the network for all further computations. Currently every arcs bounds are computed on the same network with default flow bounds. It might improve the algorithm if the already computed bounds are set directly into the network and used when another arc is computed. It could also work to compute bounds only on a small selection of interesting arcs and propagate these bounds. In an incomplete acyclicity constraint separation algorithm like what we used for the gaslib-582 networks there are sometimes arcs where

the MIP solver did find a good solution, while it did not find one on neighboring arcs. Since the heuristic in a way propagates flow bounds it even could improve the bounds to let the heuristic run again after setting the bounds already found. There might also be different models or algorithms than the ones described in this thesis. The results show that in practice some relevant improvements on the flow bounds can be achieved by taking into account that gas flow has to be acyclic. It also shows the differences between cyclic and acyclic flow bounds can be huge not only in theory (the gap can be unbounded as we have shown) but also in practice.

4 Summary

References

[1] Tobias Achterberg, *Constraint integer programming*, Ph.D. thesis, 2007.

[2] Stephen A. Cook, *The complexity of theorem-proving procedures*, Proceedings of the Third Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '71, ACM, 1971, pp. 151–158.

[3] E. A. Dinic, *Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation*, Soviet Math Doklady **11** (1970), 1277–1280.

[4] Jack Edmonds and Richard M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. ACM **19** (1972), no. 2, 248–264.

[5] L. R. Ford and D. R. Fulkerson, *Maximal Flow through a Network.*, Canadian Journal of Mathematics **8**, 399–404.

[6] Armin Fügenschuh, Björn Geißler, Ralf Gollmer, Christine Hayn, Rene Henrion, Benjamin Hiller, Jesco Humpola, Thorsten Koch, Thomas Lehmann, Alexander Martin, Radoslava Mirkov, Antonio Morsi, Werner Römisch, Jessica Rövekamp, Lars Schewe, Martin Schmidt, Rüdiger Schultz, Robert Schwarz, Jonas Schweiger, Claudia Stangl, Marc C. Steinbach, and Bernhard M. Willert, *Mathematical optimization for challenging network planning problems in unbundled liberalized gas markets*, Energy Systems **5** (2013), no. 3, 449 – 473.

[7] Andrew V. Goldberg and Robert E. Tarjan, *Finding minimum-cost circulations by canceling negative cycles*, J. ACM **36** (1989), no. 4, 873–886.

[8] Martin Grötschel, Michael Jünger, and Gerhard Reinelt, *On the acyclic subgraph polytope*, Mathematical Programming **33** (1985), no. 1, 28–42 (English).

[9] Thea Göllner, *Preprocessingtechniken für die optimierung stationärer gasnetzwerke*, 2010.

[10] Jesco Humpola, Imke Joormann, Djamal Oucherif, Marc E. Pfetsch, Lars Schewe, Martin Schmidt, and Robert Schwarz, *GasLib – A Library of Gas Network Instances*, Tech. report, November 2015.

[11] N. Karmarkar, *A new polynomial-time algorithm for linear programming*, Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '84, ACM, 1984, pp. 302–311.

[12] Richard M. Karp, *Complexity of computer computations: Proceedings of a symposium on the complexity of computer computations, held march 20–22, 1972, at the ibm thomas j. watson research center, yorktown heights, new york, and sponsored by the office of naval research, mathematics program, ibm world trade corporation, and the ibm research mathematical sciences department*, ch. Reducibility among Combinatorial Problems, pp. 85–103, Springer US, Boston, MA, 1972.

[13] L.G. Khachiyan, *Polynomial algorithms in linear programming*, USSR Computational Mathematics and Mathematical Physics **20** (1980), no. 1, 53 – 72.

[14] Bernhard Korte and Jens Vygen, *Combinatorial optimization: Theory and algorithms*, 4th ed., Springer Publishing Company, Incorporated, 2007.

[15] James B. Orlin, *A polynomial time primal network simplex algorithm for minimum cost flows*, Mathematical Programming **78** (1997), no. 2, 109–129 (English).

[16] Marc E. Pfetsch, Armin Fügenschuh, Björn Geißler, Nina Geißler, Ralf Gollmer, Benjamin Hiller, Jesco Humpola, Thorsten Koch, Thomas Lehmann, Alexander Martin, Antonio Morsi, Jessica Rövekamp, Lars Schewe, Martin Schmidt, Rüdiger Schultz, Robert Schwarz, Jonas Schweiger, Claudia Stangl, Marc C. Steinbach, Stefan Vigerske, and Bernhard M. Willert, *Validation of nominations in gas network optimization: Models, methods, and solutions*, Tech. Report 12-41, ZIB, Takustr.7, 14195 Berlin, 2012.