Karlsruhe Institute of Technology

Simultaneous Integer Flows

Bachelor Thesis of

Emil Dohse

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Guido Brückner, M. Sc.

Time Period: 1th June 2019 – 30th September 2019

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, September 30, 2019

Abstract

Flow algorithms have important practical and theoretical applications. A recent trend in algorithmics is to look at simultaneous variants of well-known problems. In this thesis, simultaneous integer flows are examined. For a given family of graphs, some edges in common, and some individual ones, a simultaneous flow consists of a feasible flow for each graph of the family so that the flow is the same on all common edges.

We start by comparing simultaneous flows to standard flows, to see that most structural properties, like augmenting paths or min-cut max-flow duality, do not seem to translate to simultaneous flows. We then consider the decisions problem SimFlow, that asks whether a simultaneous integer flow with a certain value exists. We show that SimFlow is NP-complete even with heavy restrictions to the flow networks, like the common network only consisting of a path. In addition to that, we show that there is no polynomial $2^{n(1-\epsilon)}$-approximation algorithm for simultaneous flows. On the positive side, we prove that SimFlow is FPT in the number of common or individual edges.

Deutsche Zusammenfassung

Fluss-Algorithmen sind ein wichtiger Teil theoretischer, wie praktischer Anwendungen. Ein aktueller Trend in der Algorithmik ist es, simultane Varianten bekannter Probleme zu betrachten. In dieser Arbeit betrachten wir simultane ganzzahlige Flüsse. Gegeben eine Familie von Graphen, die die gleichen Knoten, einige gleiche und manche eigene Kanten besitzen, ist ein simultaner Fluss nun ein Fluss in jedem der Graphen, so dass der Fluss auf den gleichen Kanten der selbe ist. Dazu definieren wir das Entscheidungsproblem SimFlow, welches entscheidet, ob es einen simultanen ganzzahligen Fluss gibt, der einen bestimmten Flusswert hat.

Zu Beginn vergleichen wir simultane Flüsse mit den bekannten Flüssen. Dabei sehen wir, dass die meisten strukturellen Eigenschaften von Flüssen, wie die Dualität von Min-Cuts zu Max-Flows oder die Existenz von augmentierenden Pfaden, nicht auf den simultanen Fall übertragen können. Danach zeigen wir, dass SimFlow stark NP-vollständig ist, selbst wenn man die einzelnen Flussnetzwerke von der Struktur her stark einschränkt. Es gibt also keinen effizienten Algorithmus für SimFlow. Zusätzlich zeigen wir, dass es keinen polynomiellen $2^{n(1-\epsilon)}$-Approximationsalgorithmus für simultane Flüsse gibt. Abschließend wird gezeigt, dass SimFlow FPT in der Anzahl der gemeinsamen Kanten oder der eigenen Kanten ist.

Contents

1. Introduction

Flow algorithms have important practical and theoretical applications. Be it distributed computer networks, planing or physical networks of any sort, flow algorithms are utilized. From bipartite matchings ([AMO88]) to solving scheduling tasks for airlines ([CLRS01]), flows are used. Historically maximum flows were developed for application in railway systems, or to be precise, the U.S. military wanted to know how much the Soviet railway system could transport and where to cut the network for it to stop working([Sch02]). Where to cut the network, can be solved with maximum flows because there is a duality between maximum flow and minimum cuts. [AMO88] is an in-depth work on flows. It covers almost everything we'll use about flows in this thesis. Many efficient algorithms have been found to compute maximum flows. And it is still a research topic with i.e. a near-linear algorithm for multiple-source multiple-sink maximum flow in planar graphs in 2017 ([BKM$^+$17]), a linear time algorithm for maximum flows in st-planar graphs ([HKRS97]) and a $O(nm)$ for maximum flows in general graphs in 2013 ([Orl13]).

The maximum flow problem has the neat property that for a maximum solution, there always exists an integer solution with the same flow value as well, if all capacities were integers (again [AMO88]).

In this thesis, we are going to look at simultaneous integer flows. That means, given a family of graphs with the same vertices, some edges in common, called the shared network and some individual ones, called non-shared networks. We want to decide if there is an integer flow with a certain flow value in each graph so that the flow on the shared edges is the same? The problem will be called SimFlow and formally defined in the Preliminaries, Definition 2.4. We'll only look at the integer case, real simultaneous flows can be computed efficiently, for example with an LP.

It is a recent trend in algorithmics to look at simultaneous problems. We give a couple of examples to motivate this trend. Often we don't only have one instance of a graph, but multiple ones, e.g., for different states or time periods. Often, they share some, if not all vertices and only differ in a few edges. If we want to layout and display these graphs over time, having the same vertices in the same places helps a lot for a better understanding of what has changed. [BKR12] gives some more criteria you might be looking for in simultaneous drawing and gives a good overview of different classes of simultaneous planar embeddings and what that means for their lay-outing. The difficulty of deciding whether graphs have a simultaneous planar embedding varies highly for different classes, e.g., it is NP-complete to decide, if 3 planar graphs have a simultaneous embedding with fixed edges,

for two graphs the complexity is unknown and if they are biconnected it can be decided in linear time.

Similarly, everywhere, were multiple instances of a problem appear that have some part in common the inspection of the simultaneous problem can be interesting. The problem of ordering a set of elements is part of a lot of problems in computer science. Often we don't need a total order for all elements though, but there is some degree of freedom. For example, some subsets might not have to be ordered at all, leading to multiple feasible orderings. Here as well, ordering multiple sets with certain subsets having the same ordering is of interest. *PQ-trees* are trees with nodes of type P and Q, that allow specific orderings for their child nodes, in the embedding of the tree. Looking at the leaves, *PQ-trees* yield such an ordering called *PQ-ordering*. [BR16] examines the problem of finding simultaneous *PQ-orderings*. They show that is NP-complete and solve it efficiently for certain instance classes.

Keeping in mind the duality of maximum flows and min-cuts, [Kri09] which looks at simultaneous min-cuts is not only another example of a simultaneous problem but also goes in the direction of flows. He looks at two different kinds of simultaneous min-cuts, ones that are the same in all graphs and ones that may differ slightly but might thereby have a lower cut value. All algorithms discussed and found in the thesis for optimal solutions run in exponential time, therefor some heuristics are presented as well. Though duality of min-cuts and max-flow, simultaneous min-cuts don't directly solve SimFlow, as we'll see later on in this thesis.

When looking at simultaneous solution problems, it is often helpful to first look at the partial solution problem. That means, we have got an instance of a problem where a partial solution is already given. We now want to solve the instance without changing the partial solution. Let's have a look at simultaneous lay-outing of a graph again. We could, instead of trying to choose locations for the vertices and curves for the edges, all at the same time, take one graph, find a layout for it, and then solve the lay-outing problem on all other graphs, restricted for the shared vertices and edges to have the same locations or curves respectively. This partial problem can be solved efficiently, as shown in [ABF$^+$10]. For *PQ-orderings* the partial problem, can be solved efficiently as well, as shown in ([KKV11]. Partial min-cuts can be extended efficiently by removing the edges of the partial solution and looking for a min-cut then. Coming back to flows, partial maximum flow is, that we set the flow on certain edges to be a certain value and want to get a maximum flow then. This can be solved by (integer) min-cost-max-flow with demands for each vertex and lower bounds for the flow on edges, allowing us to ensure the flow on a certain edge to have a specific value. This problem has multiple strongly polynomial solutions to be seen for example in [CC01]. This looks quite promising already, but the simultaneous problem turns out to be significantly harder. Quite similar to simultaneous flows, there is IntegerEqualFlow. IntegerEqualFlow is a generalization of maximum flows, where there are disjoint sets of edges, that must have the same flow. [MS09] gives an overview of what has been shown for IntegerEqualFlow and shows, that there is no polynomial $2^{n(1-\epsilon)}$-approximation algorithms for the optimization variant MaxIntegerEqualFlow. In this thesis, we'll see that this is the case for MaxSimFlow, the optimization variant of SimFlow as well. At least, from [MS09] follows that IntegerEqualFlow is fixed-parameter tractable. In [Sah74] it is shown that equal flow is NP-complete. It is even shown, that this holds, if the edge sets with equal capacity only consist of two edges. In this thesis, we'll refer to this as HomArcFlow. Concerning heuristics for solving HomArcFlow, an overview is given in [LL98]. There and in [AKS88], heuristics are given. According to [LL98], their heuristic can be extended to handle MaxIntegerEqualFlow involving more than two equal edges per set as well.

We are going to see, that instances of MaxSimFlow and MaxIntegerEqualFlow can be linearly transformed to one an other, preserving approximation properties. Thus, the $2^{n(1-\epsilon)}$-inapproximability is passed on to MaxSimFlow. But so are the heuristics for HomArcFlow that is quite similar to SimFlow with a graph family of two graphs. Looking at efficiently solvable instances of simultaneous flows, to my knowledge only [EH06] looked at simultaneous flows so far, resulting in an algorithm to find a simultaneous flow if the incidence matrix is a consecutive-ones-matrix.

Contribution and outline

In this thesis, we are going to start in Chapter 2 by defining SimFlow and its variants, then in Chapter 3 comparing simultaneous flows to normal flows, to see what properties and dualities still exist when going into the simultaneous case. After looking at some general properties of simultaneous flows, in Chapter 4 we then show that it is NP-complete to find a simultaneous flow, even with heavy restrictions to the graphs like the shared network only being a path and the non-shared connecting vertices along this path. Chapter 5 then looks at approximability of the optimization variant MaxSimFlow and see that it is $2^{n(1-\epsilon)}$-inapproximable. In Chapter 6, we look at fixed-parameter tractability (FPT). As already mentioned, SimFlow is quite similar to IntegerEqualFlow which is FPT. We'll end by proving SimFlow is FPT in the number of intersections of the common edges with the individual edges of the family of graphs. In the conclusion, Chapter 7, we look back at what was shown in this thesis.

2. Preliminaries

This chapter provides the formal background for the proofs in this thesis. First, we define the basics about flows and what a simultaneous flow is. Then, the decision problems SimFlow, MinCostSimFlow and the optimization problem MaxSimFlow are introduced.

Definition 2.1. flow*: Given a directed graph $G := (V, E)$ with specific vertices s and t, (*source *and* sink*) and a* capacity function $c : E \to \mathbb{N} \cup \infty$*. A* flow *is a function $f : E \to \mathbb{N}_0$ with $0 \leq f(e) \leq c(e)$ $(e \in E)$ and each vertex, except for source and sink, having the same amount of flow entering and leaving. That is, if we denote v^- and v^+ as the sets of edges leaving or entering a vertex $v \in V$, respectively, $\sum_{e \in v^-} f(e) = \sum_{e \in v^+} f(e)$ for all vertices $v \in V \setminus \{s, t\}$. This is called* flow conservation*.*

Definition 2.2. simultaneous flow*: Given m directed graphs $G_i := (V, E_i \cup E_{shared})$, with specific vertices s_i and t_i, (*source *and* sink*) and a* capacity function $c_i : E \cup E_{shared} \to \mathbb{N} \cup \infty$ *while $c_i(e) = c_j(e)$ $(e \in E_{shared}$ and $1 \leq i, j \leq m)$, a* simultaneous flow *is a flow $f_i : E \cup E_{shared} \to \mathbb{N}_0$ where $f_i(e) = f_j(e)$ for all $e \in E_{shared}$ and for $1 \leq i \leq m$.*

The sum of flow leaving s_i is also called the flow value or s_i-t_i flow value of the graph G_i $(1 \leq i \leq m)$.

Definition 2.3. simultaneous property*: $f_i(e) = f_j(e)$ $\forall e \in E_{shared}$. That is, the flow on the shared edges is the same in all G_i.*

Since this is the property, that separates simultaneous flows form multiple standard flows, we'll often refer to the *simultaneous property*.

We denote the graphs (V_{shared}, E_{shared}) as the *shared network* with $V_{shared} := \{u, v \in V | (u, v)$ or (v, u) in $E_{shared}\}$ and the graphs (V_i, E_i) as the *non-shared networks of the graphs G_i* with $V_i := \{u, v \in V | (u, v)$ or (v, u) in $E_i\}$. If we say a vertex is part of in the shared or of *the* non-shared network it means that it is start or end of an edge in the shared or in all non-shared networks. This leads to the intersection of both networks being the set $V_{shared} \cap \bigcap_{1 \leq i \leq m} V_i$.

Talking about the *residual graph* of a graph, we denote a graph that has an additional reverse edge $\hat{e}$ for each edge e that has capacity $c(\hat{e}) = c(e) - f(e)$. If the original edge was a shared edge, the reverse edge is a shared edge as well, $e \in E_{shared} \Rightarrow \hat{e} \in E_{shared}$.

As shown in the definition, we are looking at integer flows. Unless stated otherwise, all flows from now on are integer flows. Real simultaneous flows can be solved in polynomial

time via an LP ([EH06] for the LP) which is well known to be solvable in polynomial time ([KHA79]).

Often in this thesis we'll only have two graphs G_1 and G_2. Figures will use the colors red for non-shared edges of G_1, blue for non-shared edges of G_2 and black for shared edges.

2.1. Decision and Optimization Problems Using Simultaneous Flows

The definition of simultaneous flows leads to the following decision problem:

Decision Problem 2.4. SimFlow*: Given the m graphs G_i from Definition 2.2 and m values $h_i \in \mathbb{N}_0$.* SimFlow *asks: Is there a simultaneous flow so that the s_i-t_i flow values in the graphs G_i are equal to h_i?*

In later sections we are going to have a look at a generalization of SimFlow called MinCostSimFlow.

Decision Problem 2.5. MinCostSimFlow*: Given a* SimFlow *instance and a value $b \in \mathbb{Z}$. Additionally we have a demand $d_i \in \mathbb{Z}$ for each node, weakening the flow conservation of node i to $\sum_{j:e_{ij}\in E} f(e_{ij}) - \sum_{j:e_{ji}\in E} f(e_{ji}) = d_i$. We also add a cost $b(e_{ij}) \in \mathbb{Z}$ for each unit of flow on an edge. We want to decide if there exists a simultaneous flow satisfying the new flow conservation constraint in each node with a total cost $\sum_{e_{ij}\in E} f(e_{ij}) \cdot b(e_{ij})$ equal to b.*

This is a generalization of the original SimFlow, since for cost 0 on each edge we get a total cost of 0 and we can set d_j for j being a source or sink to h_i and $-h_i$, respectively, and $d_j = 0$ otherwise. Like this, we can decide the original SimFlow.

The optimization variant of SimFlow is defined as follows:

Optimization Problem 2.6. MaxSimFlow*: Given the m graphs G_i from 2.2. h_i denoting the s_i-t_i flow values in the graphs G_i. What is the maximum of $h := \sum_{i=1}^{m} h_i$ so that the* simultaneous property *holds.*

3. From Flows to Simultaneous Flows

First of, we want to have a look at some general properties we get, when interacting with shared and non-shared edges. To do that, we'll look at properties of maximum flows and try to find equivalent ones for simultaneous flows. In the process, we'll get an insight into the difficulties of solving SimFlow. We start by looking the min-cut max-flow duality, continue with augmenting paths, then look at the integrality property.

Throughout this chapter, we'll be referring to the m graphs $G_i = (V, E_{shared} \cup E_i)$, if needed with flow values h_i from sources s_i to sinks t_i, with $1 \leq i \leq m$, from Definitions 2.2, 2.4 or 2.6, respectively. If we just have vertices s and t in the context of simultaneous flows, this means that all s_i equal s and all t_i equal t. *Min-cut* denotes a min-cut separating source and sink of the graphs we are looking at. The *value* of a min-cut is its capacity.

3.1. Min-Cuts and Simultaneous Flows

A lot of the structural properties of flows don't translate directly to simultaneous flows. In this section we'll see, that there is no strong connection between min-cuts and maximum simultaneous flow. For the sake of simplicity and without loss of generality, all sources s_i and all sinks t_i equal a common source s or a common sink t, respectively.

Even though a min-cut in all G_i trivially yields an upper bound to the maximum simultaneous flow, because a simultaneous flow consists of feasible standard flows, this upper bound isn't always reached.

Lemma 3.1. *A min-cut in all G_i does not yield a feasible simultaneous flow in general.*

Proof. Looking at Figure 3.1: All edges have capacity 1. The min-cut value for each individual graph is 1. If there is any flow in the graphs, in G_1 (left) flow has to use the red edge, in G_2 (right) the blue one. But since flow in the simultaneous case has to be the same on all shared edges, in particular on the ones leading to the red or blue edge, none of the paths is taken and the simultaneous flow is 0. □

Here we used that the shared network didn't connect source and sink and we had to rely on the non-shared networks to make flow possible. If, on the other hand, the shared network connects source and sink, we get a lower bound for the simultaneous flow. This can be seen in Lemma 3.2.

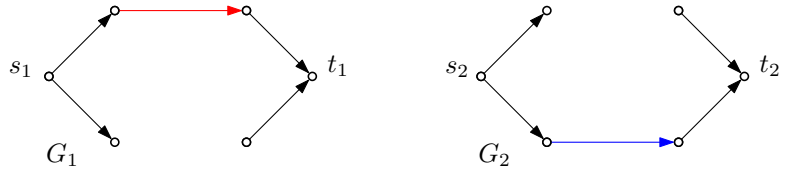

Figure 3.1.: Counter example min-cuts:All edges have capacity 1. The maximum simultaneous flow is 0 in both networks, all min-cuts have value 1. The shared edges are black, the non-shared edges red in G_1, blue in G_2.

Lemma 3.2. *If the shared network contains source s and sink t, the maximum simultaneous flow in each network is at least the value of a min-cut of the shared network. If it even connects the two vertices, the maximum simultaneous flow is not 0.*

Proof. A min-cut of the shared network yields a maximum flow in the shared network, due to duality of max-flow and min-cut. Since source and sink are part of the shared network, we can take this flow as feasible simultaneous flow in all graphs. If the shared network does not connect s and t, the min-cut value and lower bound are 0, which always holds. If they are connected, the min-cut value is non-zero, thus the lower bound for the maximum simultaneous flow in each network is greater than 0 as well. □

In Figure 3.1, we only had non-shared edges of one network on the upper or lower path. We'll see in Lemma 3.3 and then in Definition 3.5 how the non-shared networks have to look like, for flow to be transferred from one node in the shared network to another. Given a simultaneous flow, we look at the shared and the non-shared networks separately. In the shared network, there can be more flow than the network itself can handle, if the non-shared network of all G_i can cope with this additional flow.

Lemma 3.3. *There can only be a violation of the flow conservation in the shared network, that means different influx than outflux of a node, if **all** non-shared networks can provide this influx or take up the outflux, respectively.*

Proof. Given a simultaneous flow. The flow on the shared part of all G_i is the same. But a simultaneous flow is a feasible flow in all G_i, that means in each graph, the non-shared network has to provide the necessary influx or take up the outflux to fix flow conservation in every node. □

This especially means, a non-shared edge (u, v), $u, v \in V$ can't be used if there is one non-shared network, that doesn't have an edge leaving u or arriving in v. This is a very strong property, as we'll see throughout this thesis.

Coming back to min-cuts. We saw, that a min-cut in every G_i doesn't yield a simultaneous flow. What about the other way around, does a non-zero maximum simultaneous flow correlate with min-cuts? Maybe at least in some of the graphs? Let c_i be the capacity of a min-cut of G_i.

Lemma 3.4. *It is possible that $h_i < c_i$ for all G_i.*

Proof. We look at Figure 3.2 a): The value c_i of any min-cut in each network is 5, but the maximum simultaneous flow value h_i in each network is 4. This example proofs, that there doesn't have to be any min-cut, if we have a maximum simultaneous flow. □

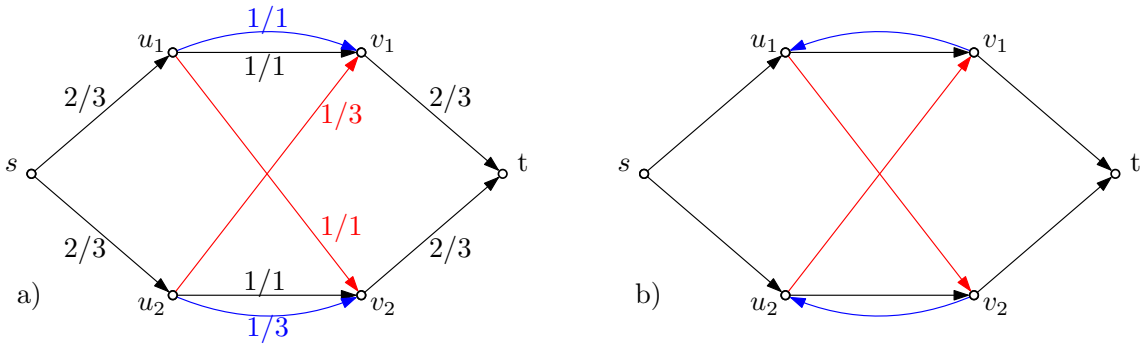

Figure 3.2.: In a) Example for maximum simultaneous flow not leading to a min-cut in any of the networks. The red edges are the ones of network G_1, the blue ones are of G_2. In b) the edges of the non-shared network of G_2 are reversed, to show the cycle-like structure, if non-shared edges are used to transfer flow from one shared node to another.

This means, there is no duality of simultaneous flows and min-cuts in the graphs G_i.

Figure 3.2 a) brings up what we saw in Lemma 3.3 already. For the vertex v_1 the capacity of non-shared edges arriving at it is 1 in G_2, for the vertex v_2, the capacity it is 1 in G_1. Thus, all non-shared flow is limited by 1 on each edge arriving at one of the two vertices. Since all non-shared edges arrive in one of the two vertices, the flow on all of them is limited by 1, even if they have higher capacities. Looking at this more generally, we'll see that there has to be some cycle-like structure for the relation of non-shared edges, if flow can be transferred from a shared node to another using the non-shared network. An influx at a shared node is only possible if an outflux happened somewhere else. This outflux has to be compensated by another influx and so on. Thus, it leads to a cycle-like relation structure, showing which influxes and outfluxes are matched. We'll call this structure *transferring cycle*. For better understanding, we look back at Figure 3.2 b). There the edges of the second non-shared network are reversed. Like this, we get a cycle, which indicates that we can transfer flow on it. The minimum capacity of an edge in the cycle is 1, which is the limit of flow, that can be transferred on these edges, as we saw above.

Formally, we can define transferring cycles like we'll do now. Given the set of vertices V and the non-shared edge sets E_i of all G_i, we reverse all edges of E_1 to get $E_1' := \{(v,u)|(u,v) \in E_1\}$. We further define $G_i' := (V, E_i \cup E_1')$, $(2 \leq i \leq m)$.

Definition 3.5. transferring cycle*: If there is a non-empty union of cycles $\mathcal{C}_i$ in each G_i' $(2 \leq i \leq m)$, with at least one edge from E_1' and at least one edge from E_i per cycle $C_i \in \mathcal{C}_i$. If there is a set of cycles $\{C_2, ..., C_m\}, C_i \in \mathcal{C}_i$ consisting of the same set $A \subseteq V$ of vertices in all cycles C_i, a* transferring cycle *consists of the edges $e_i \in E_i$ and the edges $e \in E_1$ for which there exists $C_i \in \mathcal{C}_i$ for any $2 \leq i \leq m$ with $e_i \in C_i$ or $reverse(e) \in C_i$. A transferring cycle E is called* prime*, if there is no transferring cycle $E' \subset E$.*

Roughly spoken this means: For all flow, leaving or entering the shared network through the non-shared network, there have to be edges in all networks that can restore flow conservation. Thus, a structure that fulfills Lemma 3.3.

But why this cyclic relation? If we look at Figure 3.2 b) once more, we can see, that if one of the non-shared edges of the transferring cycle is used, all others have to be used with the same flow as well, to guarantee flow conservation. Like this, we have a powerful new tool. We can enforce multiple edges in one graph to have the same flow. Lemma 3.6 explains this in more detail.

Lemma 3.6. *Given a flow network, we can enforce two edges to have the same flow, using simultaneous flows.*

Proof. Let (u_1, v_1) and (u_2, v_2) be the two edges of the flow network that are supposed to have the same flow. We now take the whole network without the two edges as shared network and put transferring cycles (see Figure 3.3) in the place where they were. If $u_1 \neq u_2$ and $v_1 \neq v_2$ we can use a) from the figure, if $u_1 = u_2$ or $v_1 = v_2$ we use b) or c) respectively. The different cases are necessary because if the start or the end are the same, this construction would not lead to anything but the dotted edge itself. The whole transferring cycle will be used with the same flow, thus the red edges now ensure that there is the same flow from u_1 to v_1 and from u_2 to v_2. □

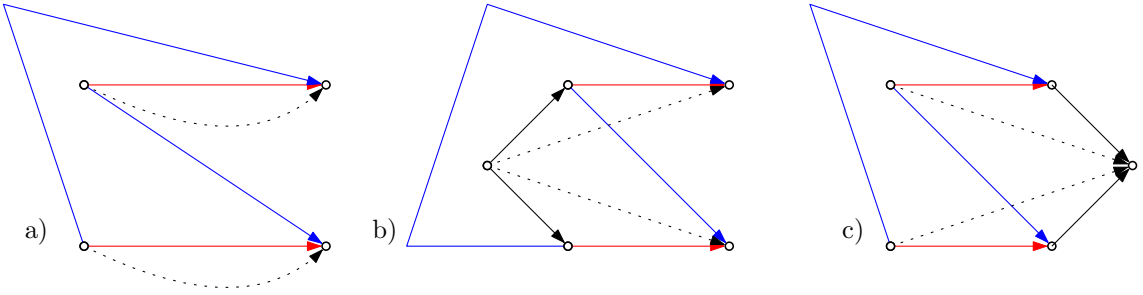

Figure 3.3.: Transferring cycles that lead to two edges having the same flow: The edges that are supposed to have the same flow are denoted by the dotted edges. a) shows the transferring cycle, if the dotted edges have different starting points. b) and c) show the case if either start or end of the dotted edges is the same. The shared edges are black, the non-shared edges red in G_1, blue in G_2.

This construction can not yet be used arbitrarily often, to enforce multiple pairs of edges in the same graph to have the same flow, like we'll see in the proof of Lemma 3.7. In the next chapter we'll look at this in more detail, Lemma 4.7 will show us that, with minor adjustments we can even enforce multiple arbitrary large sets of edges to have the same flow using transferring cycles.

Lemma 3.7. *If two transferring cycles intersect in one vertex i and a non-shared edge is added to connect two other vertices, a new transferring cycle is build. One intersection doesn't lead to a new transferring cycle.*

Proof. We have a look at Figure 3.4: The figure illustrates two transferring cycles with one intersection and what happens if another intersection is added. In a) the two transferring cycles only intersect in node i. If e_1 is used, the whole lower transferring cycle has to be used to guarantee flow conservation in u (like in Lemma 3.3). Thus, e_3 has to have the same flow as e_1. The same holds for e_2 and e_4. If one or both of the cycles were reversed, the same argumentation would hold. That means, if two transferring cycles only have an intersection in one node, they don't build a new transferring cycle that can only be used all at once. Now we move to part b) of the figure. There we can see two possible cases of where a non-shared edge can be inserted. Edges e_5 and e_6. In c) we can see the blue edges of G_2 reversed in the new transferring cycles of e_5 and e_6. But why does one more connection of the cycles always lead to a new cycle? We'll now look more closely at what happens at node u_0. We only need to look at 4 cases which are shown in Figure 3.5. There, for each case the new dashed edge either enters (in a) and b)) or leaves (in c) and d)) the transferring cycle u_0, u_1 and u_2 lay on. In case a) the new edge enters at u_0. u_0 also has a red edge of G_1 entering, which can be used now to guarantee flow conservation at u_0.

Because the red edge is used now, we need to guarantee flow conservation at u_1 as well. This is done by using the edges of the transferring cycle all the way to the intersection with the second cycle. In Figure 3.4 this was i. In general, we have the same 4 cases at the intersection again: we arrive with an edge either incoming or outgoing of the second cycle and the second cycle has two incoming or two outgoing edges of this intersection. In the Figure 3.4, i has incoming edges in any case. Thus, we can move onto the second cycle with our flow conservation argument which can then be passed on, along the cycle, until we reach v. Like this, we have a new transferring cycle. Case d) works the same way. In cases b) and c) flow is transferred in the network of G_2 from v to u_2 or the other way around. Then, we start using the transferring cycle at u_2. For a better understanding, it is recommended to look at Figure 3.4 b) and c) again. There this construction is visible. □

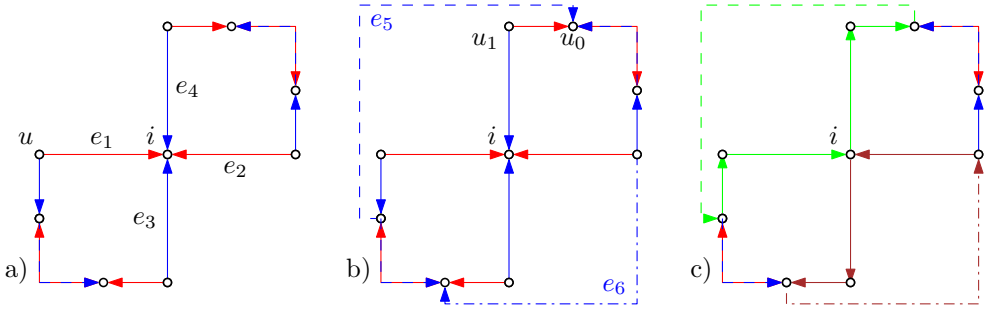

Figure 3.4.: Transferring cycles building new transferring cycles, depending on the amount of intersections: a) shows two transferring cycles with one intersection in the middle. b) shows the same cycles but with two new edges, that connect vertices of the two cycles. c) shows, that the inserted edges in b) lead to new transferring cycles. The new cycles are shown by reversing the non-shared edges of G_2 that are part of the cycles and coloring one of them brown, the other green. The shared edges are black, the non-shared edges red in G_1, blue in G_2.

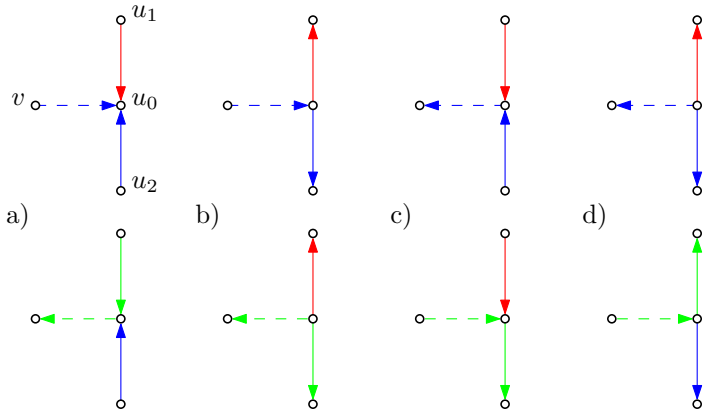

Figure 3.5.: Possibilities of how an edge can connect to a cycle. The dashed edge is the new edge, the other edges are part of a transferring cycle. In the lower part of each case, the edges of the non-shared network of G_2 that are used in the new tranfering cycle are reversed and the edges that are part of the new cycle are highlighted in green.

This lemma means, that if we want to ensure large sets of edges to have the same flow, similar to what we did in Lemma 3.6, we have to be careful that the transferring cycles

only have one intersection at maximum. If they had more than one intersection, it would lead to new transferring cycles and it would not be ensured anymore that all edges along our original transferring cycles have the same flow.

To conclude this section on min-cuts, we want to look at classes of simultaneous min-cuts like in [Kri09]. Our previous examples work for these already. One class are *small simultaneous cuts*, that means we cut between the same vertices and want the sum of all these cuts to be as small as possible. Figure 3.1 already showed that this doesn't work in general. If we look at *minimum simultaneous cuts*, that means cuts that are as similar as possible and have a minimum value the same example can be used. We can conclude that if simultaneous min-cuts look at edges in the shared and non-shared network, it doesn't lead to feasible flows in general.

The shared network in all graphs is the same, so what if we only look for cuts there, which then will be a cut in all shared networks? Like we already saw in Lemma 3.2, min-cuts in the shared network yields a lower bound for the simultaneous flow. Can we take a min-cut in the shared network with its dual flow and progressively increase simultaneous flow from there on?

Lemma 3.8. *In a maximum simultaneous flow, min-cuts in the shared network don't have to be used to their full capacity.*

Proof. We take a look at Figure 3.6: The maximum simultaneous flow of 3 in both graphs is displayed. The min-cut value in the shared network is 2. If we wanted to use this min-cut to its full capacity, which separates the shared network between u and v, the two shared edges before it would need to transport at least a flow of 2 to u. This happens, but leaves u through the non-shared network. But we can't just use the shared instead of the non-shared network from u on because we wouldn't be able to use any of the non-shared edges anymore, since they all build a transferring cycle together which can only be used if all edges along it are used with the same flow. Then, we'd be left with the shared network only and a submaximal simultaneous flow of 2. □

This means we can't directly build up simultaneous flow from maximum flows in the shared network.

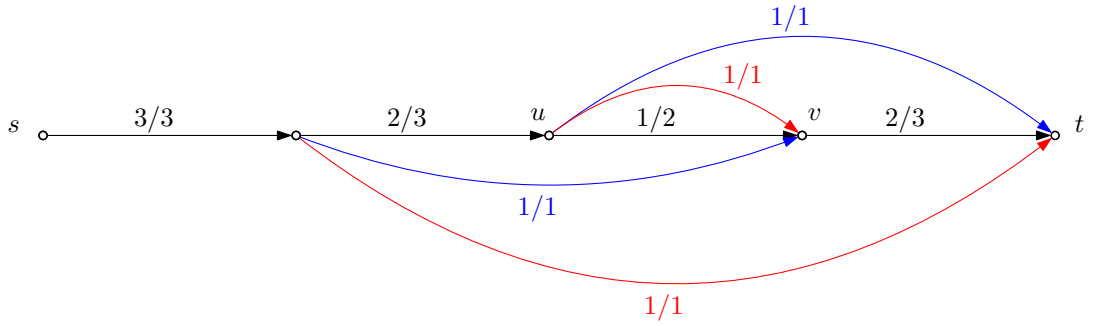

Figure 3.6.: Min-cuts in the shared network don't have to be used to their full capacity: Here the shared edge between u and v with capacity 2. The shared edges are black, the non-shared edges red in G_1, blue in G_2.

Also, there is no obvious connection to simultaneous min-cuts.

3.2. Augmenting Flow in the Simultaneous Case

Since we are dealing with general properties of simultaneous flows, a lot of what we'll need in this section was already motivated or shown in the section on min-cuts. This section goes into more detail what can happen when we want to augment simultaneous flow step by step. Looking at augmenting paths from standard flows, the question arises if transferring cycles (see Definition 3.5) have similar properties or if there are easier structures that can be used.

Even though we already defined transferring cycles, we want to see if a simultaneous variant of augmenting paths may work as well. A simultaneous augmenting path can be defined as a path with non-zero residual capacity in each G_i, so that the set of shared residual edges is the same. Thus, if we augment flow along it, it will be the same on all shared edges.

Lemma 3.9. *If there is no more simultaneous augmenting paths, the simultaneous flow value is not maximal in general.*

Proof. Looking at Figure 3.7: All edges have capacity 1. Assuming there is already a flow on the red non-shared edges in G_1. Then there are no more augmenting paths left in this graph. And the sum of flow in both graphs is 1.

In G_2 a flow of 1 is possible if the flow in G_1 had taken the shared edge instead of the non-shared ones, leading to the sum of flow being 2. Since there are no augmenting paths in one graph, there neither is a simultaneous augmenting path anymore, but the simultaneous flow is not maximal yet. □

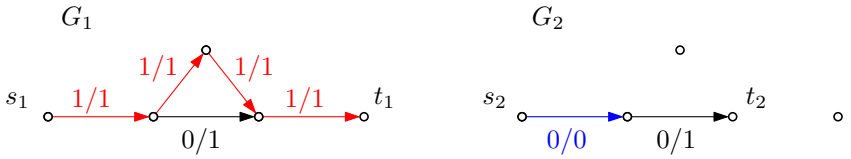

Figure 3.7.: Counter example simultaneous augmenting paths: There is no augmenting path in G_1, thus no simultaneous augmenting path anymore, but the sum of flow is not maximal yet. Flow in G_1 needs to take the shared edge, so flow in G_2 is possible. The shared edges are black, the non-shared edges red in G_1, blue in G_2.

In the following we are going to see another reason why there are no augmenting paths. Looking at another transferring cycles example (see Figure 3.8), we can't only increase the flow by 1 but have to make sure that the whole cycle is augmented, otherwise we violate flow conservation. In the example we have to increase the flow by 2 in the left instance and by 3 in the right one. After increasing, we have got maximum simultaneous flows for both graphs in both instances. There is not one path along which we can augment flow by 2 or 3, but there is two or three different paths in each graph. By making the cycle bigger, like in the example, we can enforce arbitrary large all-or-nothing flows. Here, the flow on all non-shared edges of both graphs must have the same value (see Lemma 3.3 on flow conservation). This still holds if the capacity of the edges is not 1. In Lemma 4.7 we'll see how to use this to enforce the same flow between certain vertices.

The problem with transferring cycles is that they don't necessarily augment flow through the whole graph but transport it from some shared nodes to some others. This makes it hard to choose which cycles to use. As we just saw, some of the cycles might need a lot of flow at once to be established. If some shared or non-shared edges don't have enough capacity left, because they were used together with smaller transferring cycles, it

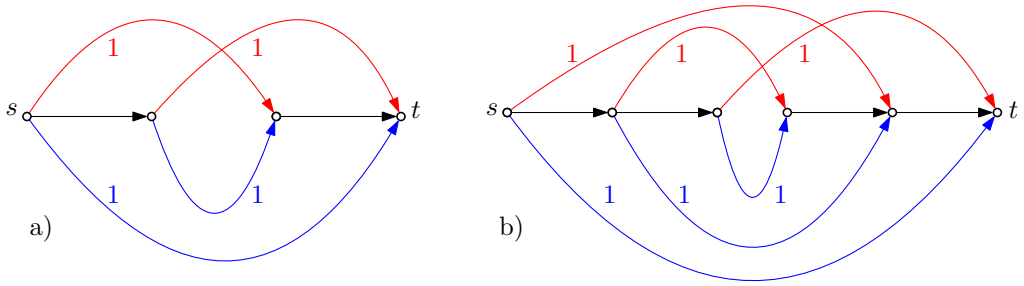

Figure 3.8.: All or nothing flow: The maximum simultaneous flow in a) is 2, in b) 3 in both graphs. These flows can only be established at once, otherwise flow conservation is violated. The shared edges are black, the non-shared edges red in G_1, blue in G_2.

can block these big cycles from being used and potentially a maximum simultaneous flow from being reached. Theorem 3.10 shows how using the *wrong* non-shared edge can lead to submaximal simultaneous flow with no more augmenting paths or cycles.

Theorem 3.10. *Simultaneous flow doesn't have to be maximal if no simultaneous augmenting paths and no transferring cycles are left.*

Proof. See Figure 3.9: In a) a flow of 1 is transferred over the non-shared network of G_2 leaving us with no transferring cycles and no simultaneous augmenting paths. In b) it is shown how a different choice of edges to put flow on would have led to a higher simultaneous flow value. □

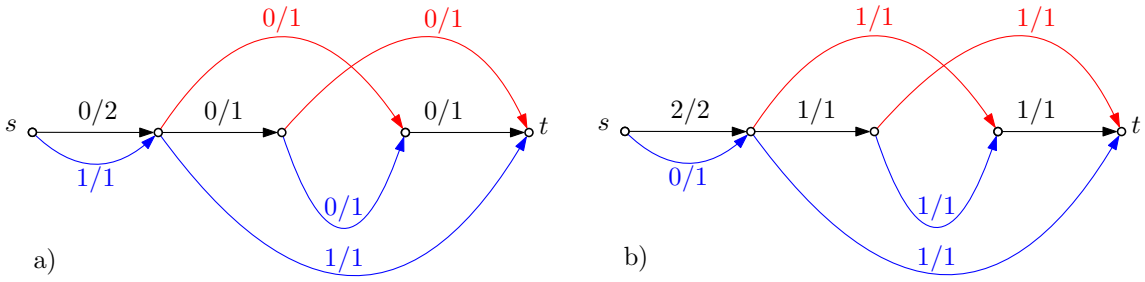

Figure 3.9.: In a): Example for submaximal simultaneous flow although there are no transferring cycles that have any capacity left . In b) the maximum simultaneous flow is displayed. The shared edges are black, the non-shared edges red in G_1, blue in G_2.

3.3. Integrality Property and Integrality Gap

The maximum flow problem has the property that for a maximum solution, there always exists an integer solution with the same flow value as well, if all capacities were integers. This is called *integrality property*. In this short section we'll see that this isn't the case for simultaneous flows and look at how much integer and real solutions can differ.

Lemma 3.11. *The maximum real simultaneous flow can be higher than the maximum integer simultaneous flow.*

Proof. See Figure 3.10: All unlabeled edges have capacity 1. In a) we can see the integer case, there in G_1 only one shared edge gets flow. Thus, the flow on both diagonal edges in

G_2 is 0, leaving us with a flow of (1,1) in the two graphs. In b) we can see the real case. There both shared edges get flow in G_1. The diagonal edges in G_2 are not bottlenecked anymore and have flow of 0.5 as well. This leaves us with a flow of (1,2) which is higher than in the integer case. □

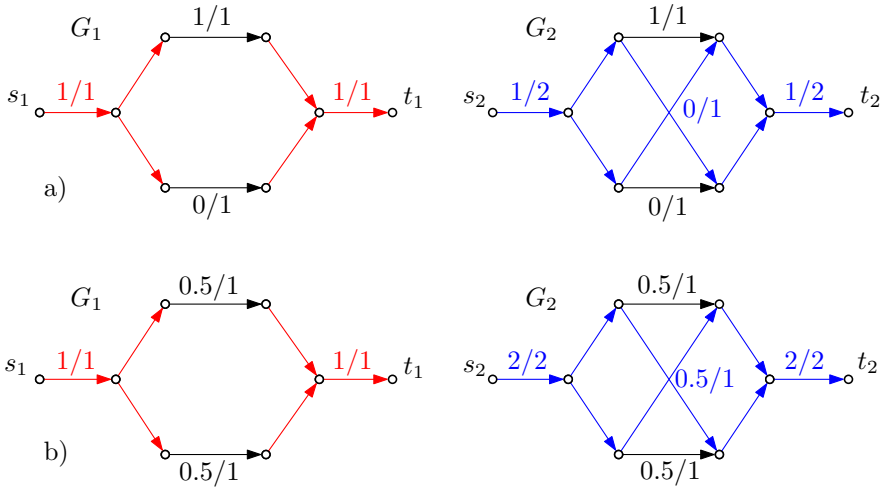

Figure 3.10.: Integrality gap: All unlabeled edges have capacity 1. a) shows the maximum integer simultaneous flow of 1 in each graph, b) the maximum real simultaneous flow of 1 in G_1 and 2 in G_2.

By putting multiple of the gadgets from Figure 3.10 in parallel we can get to quite high integrality gaps.

This means there is no integrality property for simultaneous flows.

4. SimFlow is Strongly NP-Complete

In this chapter we show that SimFlow is strongly NP-complete. We'll see, that it stays NP-complete even if we strongly restrict the shared or non-shared networks. In Section 4.1 we show strong NP-completeness for planar two-network instances, by reducing 3Sat to SimFlow. We'll then, in Section 4.2, look at HomArcFlow, show that it is strongly NP-complete and reduce it to SimFlow afterwards to get the same for instances where the non-shared network only connects nodes in the shared network. This result will be improved in the next section, Section 4.3 where we show strong NP-completeness in the two-network case with the union of all shared and non-shared networks being planar and all non-shared networks only connecting vertices in the shared network. This will be done by reducing Planar monotone 3Sat to SimFlow. The final Section 4.4 will not use any SAT variant, but reduce SubSetSum to SimFlow to show NP-completeness if the shared network only consists of a path and the non-shared network connects vertices in the shared network.

Throughout this chapter, we'll be referring to the m graphs $G_i = (V, E_{shared} \cup E_i)$, if needed with flow values h_i from sources s_i to sinks t_i, with $1 \leq i \leq m$, from Definitions 2.2, 2.4 or 2.6, respectively.

4.1. Basic 3Sat Reduction

Decision Problem 4.1. 3Sat*: Given (V, C) where V is a set of variables and C a set of sets of literals. A literal is a positive or negative occurrence of a variable. Each set $C \in \mathcal{C}$ is called clause with $1 \leq |c| \leq 3$. We now want to decide if there is an assignment function $f : V \to \{-1, 1\}$ so that each clause contains at least one positive literal that is assigned 1 or one negative literal that is assigned -1. An assignment like this is called* satisfying assignment*. The clauses can be seen as a formula in conjunctive normal form.*

Lemma 4.2. 3Sat *is NP-complete.*

Proof. See [Kar72]. □

We'll start of by reducing 3Sat to SimFlow to provide a basis for what's to come. Multiple of the upcoming reductions will use gadgets similar to the ones used in this 3Sat reduction.

At first, we'll see that we can assume that each clause contains exactly 3 literals.

Lemma 4.3. 3Sat *stays NP-complete if all clauses have exactly 3 literals.*

Proof. For all clauses of a 3Sat instance that contain 3 literals, there is nothing to do. For clauses with 2 literals we proceed as follows:

$$(x_i \vee x_j) \text{ will be turned to } (x_i \vee x_j \vee x_{d1}) \wedge (x_i \vee x_j \vee \bar{x}_{d1})$$

Clauses with 1 literal (x_i) will be replaced by:

$$(x_i \vee x_{d1} \vee x_{d2}) \wedge (x_i \vee x_{d1} \vee \bar{x}_{d2}) \wedge (x_i \vee \bar{x}_{d1} \vee x_{d2}) \wedge (x_i \vee \bar{x}_{d1} \vee \bar{x}_{d2})$$

Like this, we only add two dummy variables x_{d1}, x_{d2}, a constant amount of clauses per clauses with less than 3 literals and we don't change the satisfiability of our 3Sat instance, since the dummy variables are irrelevant. □

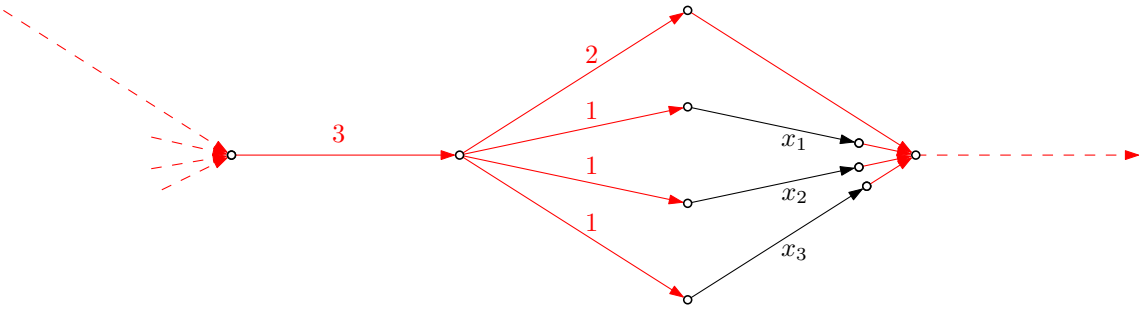

Figure 4.1.: Clause gadget for 3Sat. All these gadgets are connected at the dotted edges to ensure that a flow of exactly 3 goes through all clauses to the sink if and only if (iff) the 3Sat instance is solvable. Because of Lemma 4.3, each clause has exactly 3 literals.

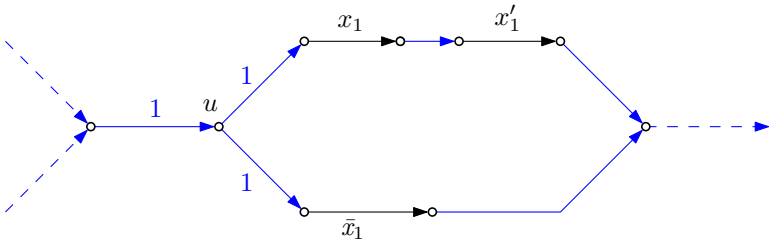

Figure 4.2.: Variable gadget for 3Sat. All these gadgets are connected at the dotted edges to ensure that a flow of exactly 1 goes through all clauses to the sink iff the 3Sat instance is solvable.

Theorem 4.4. SimFlow*, even when restricted to two-network instances with planar networks, is strongly NP-complete*

Proof. We want to show that if we solve SimFlow then we can also solve 3Sat. Given an instance of 3Sat, we build two graphs. For each 3Sat clause we use a clause gadget (Figure 4.1). It assures, that in each clause, there is at least one literal edge, that has flow 1. This literal edge is part of the shared network and will appear in the variable gadget (see bellow) as well. Due to flow conservation, concatenating these clause gadgets works like a conjunction. Therefore graph G_1 consists of these concatenated clause gadgets. If the flow through the whole graph is 3, there has to be a variable edge with flow 1 in each clause, since the non-shared network only has capacity 2. For each variable we use a variable

gadget (Figure 4.2). In case there are multiple occurrences of a variable x_i as a literal in a clause, we just add them as shared edges behind the literal edge of x_i as indicated by x_i' in the figure. If a flow of 1 enters the gadget, at the vertex u it can either take the the upper way, leading to the variable being positive in all occurrences or the lower way, leading to it being negative. Graph G_2 is the concatenation of all variable gadgets for all variables. Again using flow conservation, we are assured, that all variables will either be positive or negative.

If we define the literal edges to be shared, we force a feasible flow to satisfy all clauses while still respecting all variables and their negation to have opposite values. Therefore, if we find a simultaneous flow through G_1 with value 3 and through G_2 with value 1 we, due to flow conservation, have a satisfying values for all variables.

Since the capacity of all edges is less than or equal to 3 SimFlow is strongly NP-complete. Because the gadgets are planar and concatenating them doesn't affect planarity, this holds for planar networks.

For better understanding of the construction, Figure 4.3 shows an example clause graph for $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3)$, Figure 4.4 shows the corresponding variable graph. □

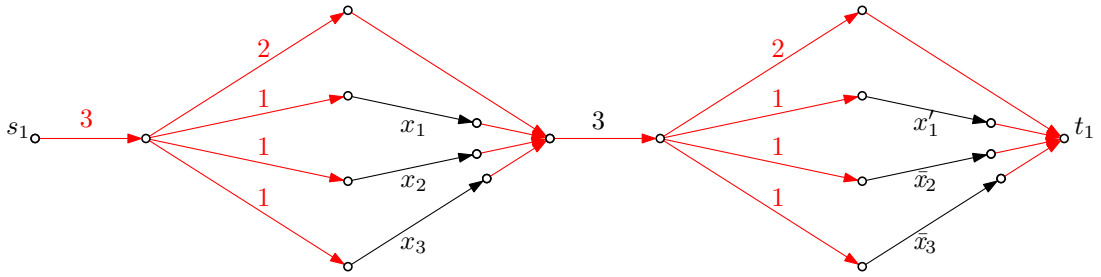

Figure 4.3.: Clause gadget graph for the formula $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3)$.

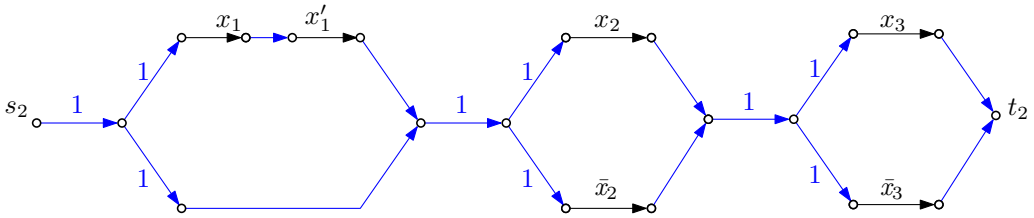

Figure 4.4.: Variable gadget graph, corresponding to clause graph in Figure 4.3.

4.2. Same Flow on Two Edges – HomArcFlow Reduction

In this section we'll look at HomArcFlow and its connection to SimFlow. In Subsection 4.2.1 we'll define HomArcFlow and prove that it is strongly NP-complete, then in Subsection 4.2.2 we'll reduce HomArcFlow to SimFlow to show NP-completeness for instances where the non-shared network only connects vertices in the shared network.

4.2.1. HomArcFlow is Strongly NP-complete

Like in [Sah74] we define the decision problem HomArcFlow (in the paper called Integer flows with homologous arcs).

Decision Problem 4.5. HomArcFlow*: Given a directed graphs $G := (V, E)$ with specific vertices s and t, (source and sink) and a capacity function $c : E \to \mathbb{N}$. Additionally*

we have a set H of tuples of edges, so called homologous arcs. We want to decide if there is a flow $f : E \to \mathbb{N}_0$ with $0 \leq f(e) \leq c(e)$ $(e \in E)$ where $f(e_1) = f(e_2)$ for all $(e_1, e_2) \in H$.

HomArcFlow is a quite similar problem to SimFlow. A solution for HomArcFlow decides whether there is an integer flow with the constraint, that on pairs of edges called *homologous arcs*, the flow is the same.

Lemma 4.6. HomArcFlow *is NP-complete.*

Proof. See [Sah74]. □

In the paper it is only shown that HomArcFlow is NP-complete. Using our gadgets from Theorem 4.4 we can easily improve on the findings of the paper to show that HomArcFlow is strongly NP-complete in the planar case already.

Theorem 4.7. HomArcFlow *is strongly NP-complete, even for planar instances.*

Proof. We can reduce 3Sat to HomArcFlow with almost the same construction we used in Theorem 4.4. We use the gadget graphs G_1 and G_2 but we take a new starting and ending node s and t and connect them to the sources s_1 and s_2 or to the sinks t_1 and t_2, respectively. Now instead of having shared edges, we just define the previously shared edges to be homologous arcs. Since planarity is not lost in this construction, it yields a solution for 3Sat if we can solve HomArcFlow for a planar graph with unit capacity. For better understanding, Figure 4.5 shows an example of this construction. □

Theorem 4.7 will help us in the next section to get results for SimFlow as well.

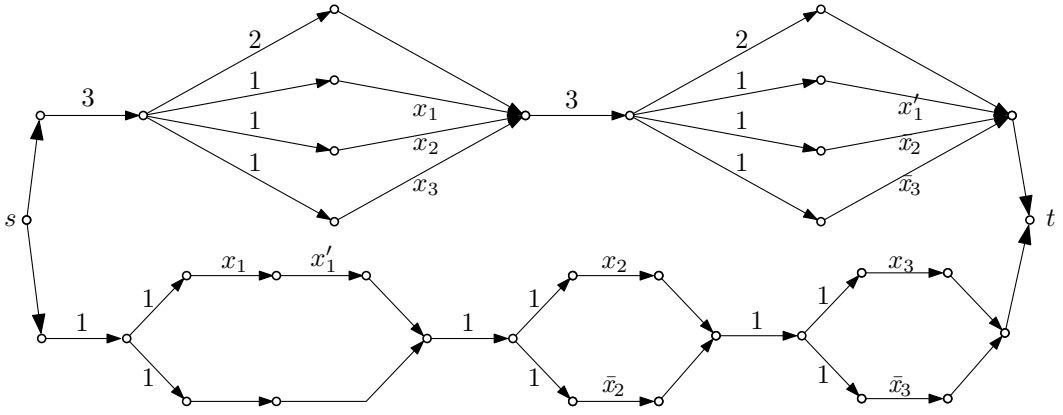

Figure 4.5.: Example for the construction in the 3Sat reduction Theorem 4.7. Similar to Figure 4.3 and Figure 4.4 for the formula $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3)$.

4.2.2. NP-Completeness for Restricted Non-Shared Network

Theorem 4.8. HomArcFlow *can be reduced to* SimFlow *in polynomial time.*

Proof. Given an instance of HomArcFlow $G := (V, E)$ and sets of homologous arcs H, we can transform the graph to an instance of SimFlow as follows. First, we remove all the homologous edges from the graph. Now we use two copies of this graph as shared network for SimFlow. We put a homologous arcs gadget (see Figure 4.6) in the places where homologous arcs were. In the figure, the previous homologous edges are the dotted edges.

The gadget mainly consists of non-shared edges in each graph G_i. Like that, a transferring cycle is created, that enforces all edges along it to have the same flow. The additional shared edges are just there so the different transferring cycles for different homologous arcs don't have any intersections, possibly leading to a new, unintended transferring cycle, similar to Lemma 3.7. In the Figure, there are three cases displayed because this way of creating a cycle only works if starting node and ending node differ for both edges. If the start or the end of one of the supposed-to-be homologous arcs is the same, a new node is added and connected to the previous start/end in the shared network. In later gadgets we are going to see that this way of using transferring cycles to enforce flow to be the same doesn't only work for two edges but for an arbitrary large set of edges. Red edges in the figure stand for edges in G_1, blue edges for ones in G_2. Black edges are additional shared edges in both. If the homologous edges have different starting end ending nodes, we use construction a) on the left, if starting node or ending node are the same we use construction b) or c), restrictively. Due to flow conservation in each network (see Lemma 3.3), it is enforced, that the red and blue edges have the same flow. Taking this flow as the solution for HomArcFlow solves this problem. □

Corollary 4.9. SimFlow *stays NP-complete on two-network instances, even if the non-shared network only connects vertices in the shared network.*

Proof. In Theorem 4.7 we have shown that HomArcFlow is strongly NP-complete on planar instances. The construction used in Theorem 4.8 proofs the claim. □

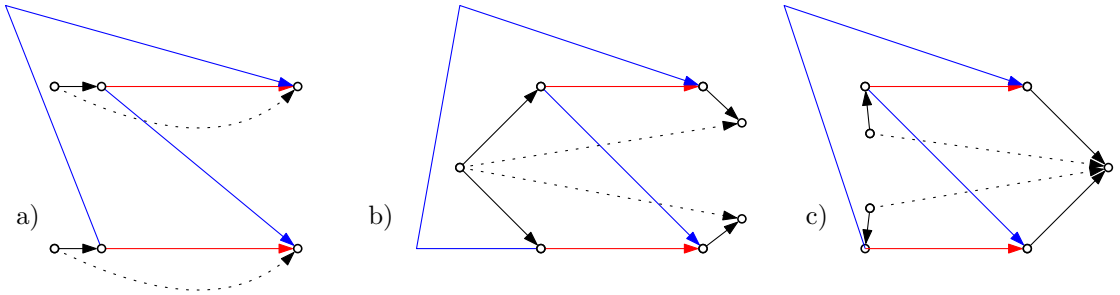

Figure 4.6.: Homologous arcs gadget: This gadget illustrates how to enforce, that two edges have the same flow. The dotted edges are the edges that are supposed to have the same flow, but are removed before the gadget is inserted.

The homologous arcs gadget in Figure 4.6 is very important for the next couple of sections. We'll use it to enforce multiple edges to have the same flow and like that get further NP-completeness results.

Lemma 4.10. *We can enforce arbitrary large sets of edges to have the same flow.*

Proof. We can use transferring cycles like displayed in Figure 4.8 for 3 edges. There, the dotted edges are the edges which are supposed to have the same flow. Red edges in the figure stand for edges in G_1, blue edges for ones in G_2. Black edges are additional shared edges in both. If the edges with the same flow have different starting and ending nodes, we can use the construction on the left. If starting node or ending node are the same for some of the edges, we use the middle or the right construction or mixtures of them, respectively. In each case, a transferring cycle is build which can only be used with the same flow on all edges. If there is no intersections, this construction works just fine. The problem here is the same we already saw in Lemma 3.7 for Figure 3.3. If the cycles for multiple equal flow edge sets intersect in more than one point, there is a new transferring cycle and we can't

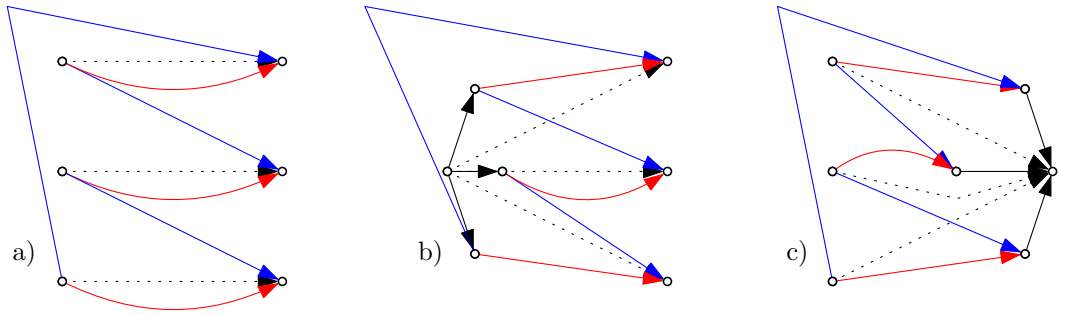

Figure 4.7.: Transferring cycles for equal flow on multiple edges: This figure shows how the transferring cycles have to look like to enforce equal flow on where the dotted edges were. This construction only works if the transferring cycles have at maximum one intersection and no additional non-shared connection. a) shows dotted edges starting at different nodes. b) and c) show the case that the dotted edges start or end at the node.

ensure there to be the same flow an all equal flow edge sets. If we don't want to restrict the equal flow edge sets to only use the same vertices once, we can look at Figure 4.7. There, additional shared edges ensure that all transferring cycles have no intersection at all. Both Figures only show the case of 3 equal flow edges. For k edges the pattern stays the same. Since the additional shared edges ensure that all starting and ending node of equal flow edges are different, we can assign each starting node and each ending node a number from 1 to k. The red edges go from starting node i to ending node i, the blue edges go from starting node i to ending node $(i+1)$ *mod* $(k+1)$. Like this, the cyclic dependency of a transferring cycle is created. □

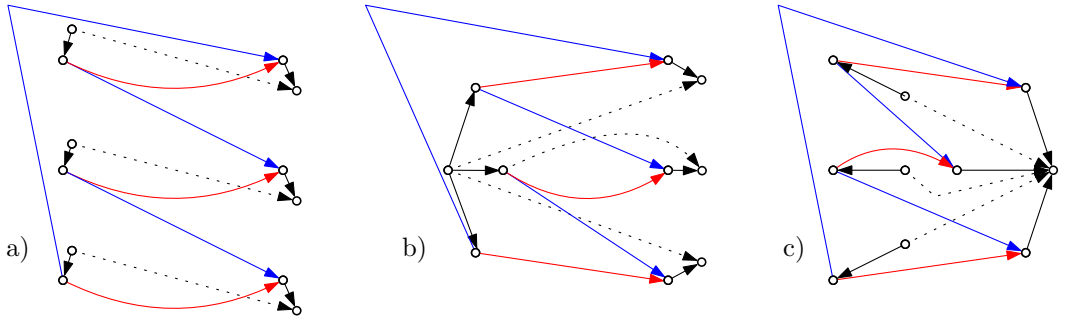

Figure 4.8.: Equal flow on multiple edges: This figure illustrates how to enforce, that three edges have the same flow. The dotted edges are the edges that are supposed to have the same flow, but are removed before the non-shared red and blue edges are inserted. a) shows dotted edges starting at different nodes. b) and c) show the case that the dotted edges start or end at the node.

4.3. Planar monotone 3Sat Reduction

This section now contains the final result of how much we can restrict an instance of SimFlow for it to stay strongly NP-complete. We'll show that if the shared network is connected, the non-shared network only connects vertices in the shared network and we have planarity of the union of the G_i, SimFlow stays strongly NP-complete for two-network instances. Union of the G_i means, that the edge set is the union of the non-shared networks and the shared network, since all G_i have the same vertices and same shared network

already. In this section, we start by introducing Planar monotone 3Sat a version of 3Sat, then we reduce it to SimFlow.

Definition 4.11. monotonous*: A boolean formula in CNF is called* monotonous*, if all clauses consist of only positive or only negative variables.*

Decision Problem 4.12. Planar monotone 3Sat*: Given a monotonous instance of* 3Sat $I := (V, C)$ *with variables* V*, clauses* C *and an additional restriction. Namely, the variable-clause graph of* I*, that consists of the nodes* $V \cup C$ *connected by an edge if one of the nodes is a variable and the other node is a clause that uses this variable. The variable-clause graph can be drawn such that all variables lie on a horizontal straight line, positive and negative clauses are drawn as horizontal line segments with integer y-coordinates below and above that line, respectively, and arcs connecting clauses and variables are drawn as non-intersecting vertical line segments. A solution now decides if this special* 3Sat *instance is satisfiable. An example can be seen in Figure 4.9.*

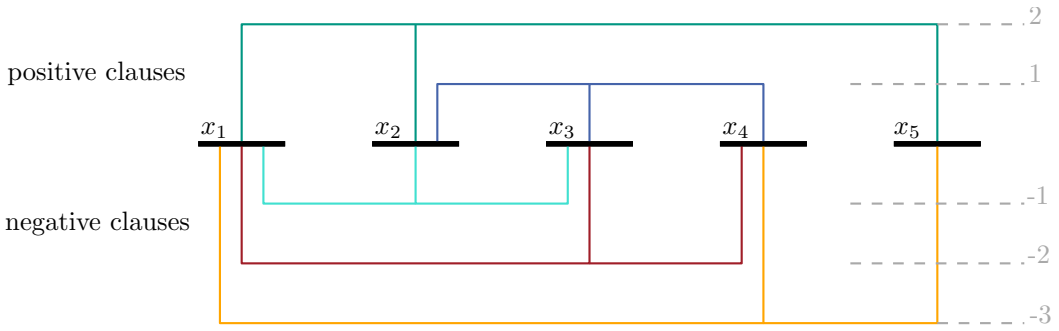

Figure 4.9.: An variable clause graph of an instance of Planar monotone 3Sat with the formula $(\bar{x}_1 \vee \bar{x}_4 \vee \bar{x}_5) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee x_2 \vee x_5) \wedge (x_2 \vee x_3 \vee x_4)$. The clauses are all on different integer y-coordinates, the variables all at $y = 0$. All clauses with y-values greater 0 only contain positive literals, all clauses with y smaller than 0 only contain negative literals. If there is an edge between a variable and a clause, the variable appears as a literal.

Lemma 4.13. Planar monotone 3Sat *is NP-complete.*

Proof. See [dBK10]. □

Roughly spoken, the next lemma allows us, that shared and non-shared edges have a polynomial amount of crossings for our next reduction to still yield a planar instance of SimFlow.

Lemma 4.14. *If an embedding of the union of all* G_i *exists, in which the shared network is planar and the non-shared edges of the different* G_i *don't cross, but they may cross shared edges. Then there exist graphs* G_i' *that have the same decision of* SimFlow *as the* G_i *for any values* h_i *but there exists an embedding in which the union of the* G_i' *is planar.*

Proof. Taking the embedding with crossings of shared and non-shared edges: If there is an intersection between a shared and a non-shared edge of a G_i, we create G_i' by adding a node in the intersection. Since this node is only part of one non-shared network, no flow goes from the shared to the non-shared network or the other way around, through this node. Doing this for every crossing yields a planar embedding. □

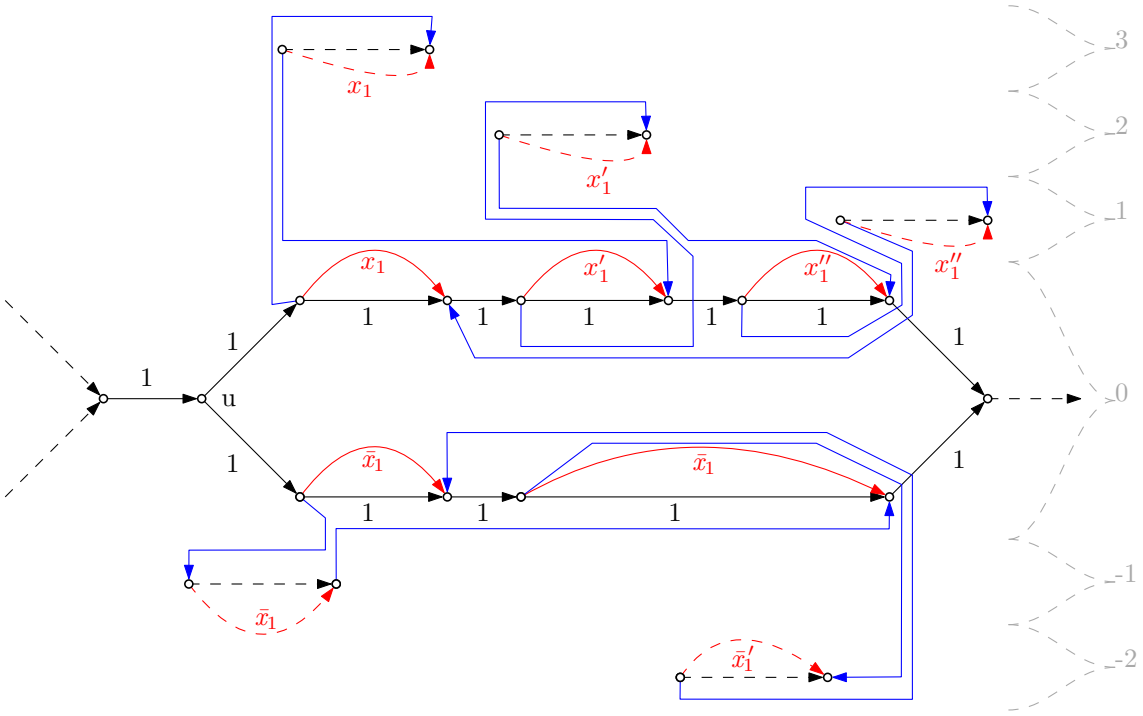

Figure 4.10.: Construction of planar variables: In this case for x_1 and $\bar{x}_1$ with 3 and 2 occurrences as literals in clauses, respectively. There are two transferring cycles, one for the positive occurrences one for the negative. They ensure that if x_1 is positive in one clause, it is positive in all clauses. In node u flow can either go up towards the positive literals or down, towards the negative literals. On the right the y-coordinates of the embedding of the variable clause graph are indicated.

Theorem 4.15. *If the shared network is connected and the same in all $G_i = (V, E_{shared} \cup E_i)$, the non-shared edges connect vertices in the shared network, all G_i are planar, and especially the graph $G = (V, E_{shared} \cup \bigcup_{1 \leq i \leq m} E_i)$ with the edge set being the union of the edge sets of all G_i is planar,* SimFlow *is strongly NP-complete.*

Proof. We reduce Planar monotone 3Sat to our restricted problem in a similar fashion we did in the previous proofs.

Our idea is to build an instance of Planar monotone 3Sat similar to the one for 3Sat using the clause and variable gadgets in Figure 4.1 and 4.2, but this time ensuring flow on true 3Sat variables by homologous arcs constructed with two non-shared edges similar to Figure 3.3. Since Planar monotone 3Sat ensures, that the connection of variables to clauses is planar, we just have to proof that our two-non-shared-edge homologous arc construction doesn't affect planarity.

Let $I := (V, C)$ be an instance of Planar monotone 3Sat with a given variable-clause graph like in definition 4.12. From this instance we want to build graphs G_1 and G_2 as an instance of SimFlow. They will both consist of the same shared network, a concatenation of gadgets as described bellow. Basically we'll take the variable-clause graph and put the variable and clause gadgets in there for the variables and clauses. In the figures of these gadgets, only the red and blue non-shared edges will make the difference of G_1 and G_2.

For variables $x \in V$ we look at Figure 4.10. In case there are multiple occurrences of x as a literal in a clause $c \in C$, we just add them as edges behind the literal edge of x as

indicated by x_1 and x_1' in the figure. In the node u, flow of 1 can either go the upper way, leading to the variable being positive in all occurrences or the lower way, leading to it being negative. The middle part of G_1 and G_2 is the concatenation of all variable gadgets for all variables, like in the variable clause graph. Flow conservation guarantees, that if the concatenation of variable gadgets has a flow of 1 going through it, it has passed all variable gadgets. Now looking at planarity: The non-shared blue edges can go along the planar embedding of edges of the variable-clause graph of I (see Figure 4.9), therefore it is planar as well. According to Lemma 4.14 the blue, non-shared edges crossing the black, shared ones does not contradict planarity. The number of crossings is polynomial in the input because we only have to cross one shared edges per y-coordinate towards the y-level we want to get to and then a constant amount of crossings in the variable and clause gadget itself. Similar to Lemma 4.10, this non-shared construction enforces all occurrences of a variable to have the same flow by creating one transferring cycle that does not intersect any other transferring cycles. To enforce that either x_1 xor $\bar{x}_1$ have flow, we limit the flow going through the gadget by 1. Like this only one of the transferring cycles representing a variable or its negotiation can be used. Theoretically, none of them could be used as well if the flow is transferred via the shared edges only. If each clause is satisfied anyways, this would just mean, that this variable is irrelevant for solving this instance of Planar monotone 3Sat. A SAT example where this could happen is $(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2)$, here, the outcome of the formula doesn't depend on x_1. If not all clauses are satisfied and this variable would make a difference solving it, the transferring cycle would be used, since only then, we solved SimFlow correctly.

For the clauses we look at Figure 4.11. Each clause $c \in C$ gets one of these gadgets. If a clause contains less than three literals it is changed to look like in Figure A.1 or A.2 for one or two literals, respectively. The positive clauses are then concatenated and so are the negative ones, like that they build up the upper and lower part of both G_1 and G_2, similar to the embedding of the variable-clause graph. Similar to the 3Sat clause gadgets in Figure 4.1, the concatenation together with flow conservation guarantees, that if these concatenated gadget subgraphs of G_1 and G_2 have flow 6, there is flow 6 through each of the gadgets. To guarantee that at least one of the variable edges has to be taken in each gadget, the flow on the shared network is limited by 5 in each gadget, right where the non-shared edges connect. The dashed edges are the ones from Figure 4.10. They ensure, that if for example x_3 is true and relevant for the solution, the whole transferring cycle and the x_3 edge of our gadget have flow 1. Looking at planarity, already for the variable gadgets we showed that all gadgets can be reached in a planar fashion. Again with Lemma 4.14 planarity is kept for the intersections of shared and non-shared network. The number of crossings is polynomial in the input with the same argumentation as above.

The start of the variable gadget graph is connected via a shared edge to the source, with capacity 1. The two concatenated clause graphs are connected to the source via a shared edge above and below the concatenated variable graph, both with capacity 6. The ends of all the graphs are connected to the sink in similar fashion.

If we now decide SimFlow on G_1 and G_2 with a flow of $6+6+1=13$ for the positive and negative clauses and the variables, we have a decision for Planar monotone 3Sat as well. All edges have capacity ≤ 6, thus we have strong NP-completeness. The construction holds what is required in the theorem. Especially, the red and blue non-shared edges never cross in a gadget and in between graphs, there are only blue edges. This means, the union of G_1 and G_2 is planar as well. Therefore SimFlow stays strongly NP-complete with these restrictions. □

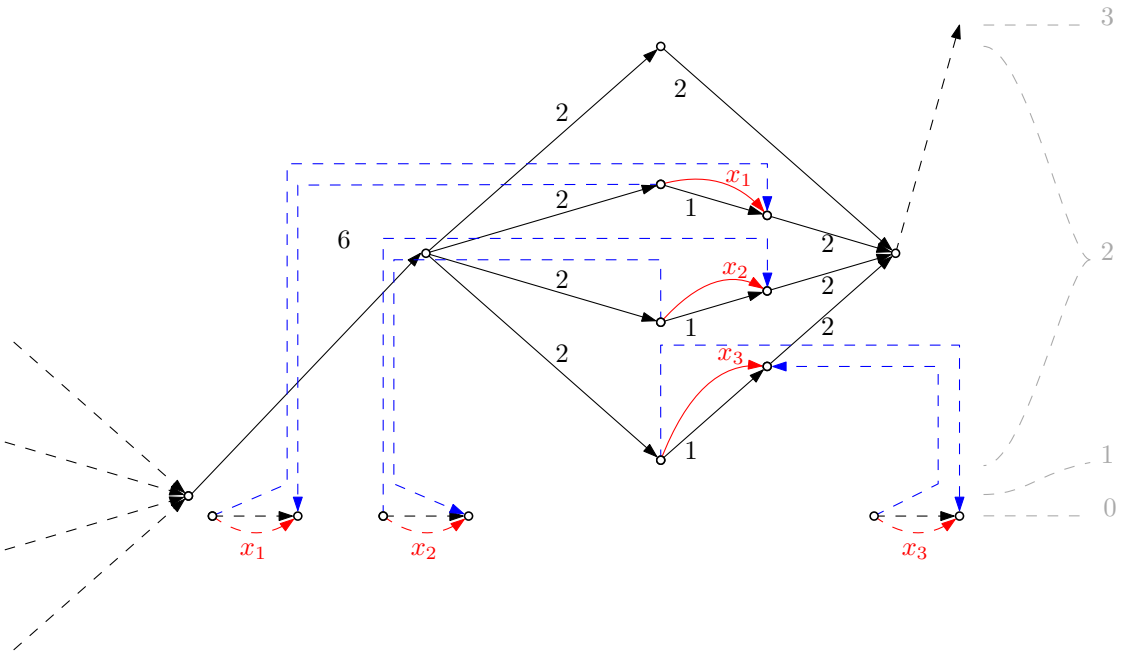

Figure 4.11.: Construction of planar clauses: The dashed edges left and right are where other clause gadgets connect. The dashed edges from bellow are ones of variable gadgets as seen in Figure 4.10, ensuring flow on the literal edges x_1, x_2, x_3 if the variables are positive. The capacity of the shared network is limited by 5, so one of the red literal edges has to be taken for the gadget to transfer 6 flow to the next gadget. On the right the y-coordinates of the embedding of the variable clause graph are indicated.

4.4. SubSetSum Reduction

Now we come to our final proof of a special case of SimFlow being NP-complete. This is the most restrictive one for the shared network, it will only consist of a path, but we don't have planarity of the non-shared networks anymore and the NP-completeness is not strong, like in the other sections in this chapter.

Decision Problem 4.16. SubSetSum*: Given a set A of integer numbers and an integer number s, does there exist a subset $A' \subseteq A$ such that the sum of its elements is equal to $\sum_{a \in A'} a = s$.*

Lemma 4.17. SubSetSum *is NP-complete.*

Proof. See [GJ79]. □

We'll reduce SubSetSum to SimFlow. For our reduction, we'll have a two-network instance of SimFlow again. The shared network will consist of a simple path, the non-shared network will only consist of edges connecting two nodes on the path, thus two nodes of the shared network. To proof that even in this very limited case, no solution can be found efficiently unless $P = NP$, we'll show that SubSetSum can be reduced to this special case, polynomially. For this we'll use all-or-nothing flows (as seen in Figure 3.8) in a combination with enforcing certain edges to have the same flow. Like that, we can build an element $i \in A$ with a graph of size $O(log(i))$, which will either be chosen fully or not at all. The shared path will have capacity 1 in between all gadgets. It is an invariant, that at the end of each gadget, we have flow 1 on the path again. In the end of the proof, there is an explanation, how this is enforced, even if a gadget is not used. In the beginning of the

path there will be some edges with higher capacities making sure all gadgets can get the necessary flow. We first introduce 2 gadgets, the full reduction will then just use these to build an instance of SimFlow from an instance of SubSetSum.

We'll look at a gadget that can double the flow it gets as input. First, we look at a version of it that gets an input of 2, then we'll look at the general version. All gadgets only work if the input is used to its full capacity. In the reduction we'll then start with an input of 1 in each network, thus it can only be used to full capacity or not at all.

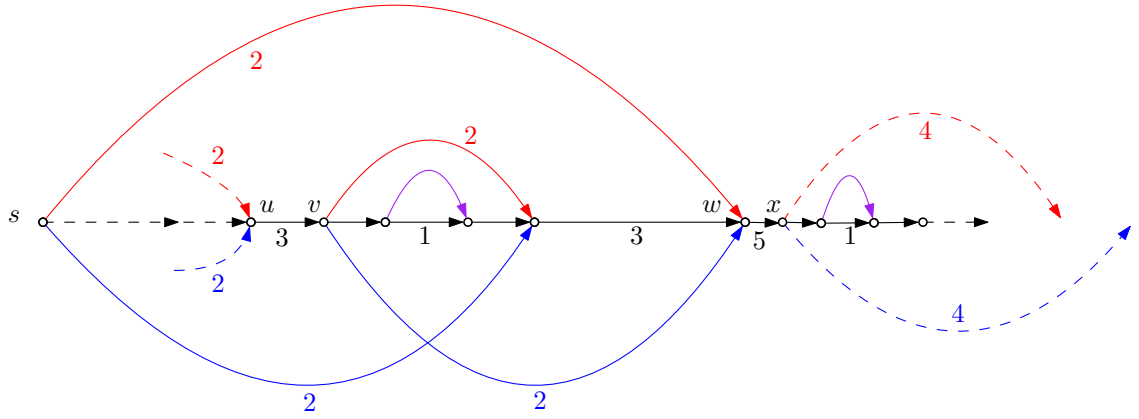

Figure 4.12.: Doubling gadget example: This gadget ensures that either the flow exiting the dashed non-shared edges is 0 or it is 4 depending on the dashed non-shared incoming edges having a flow of 0 or of 2. The purple edge has flow 1 in every case.

We have a look at Figure 4.12. In the middle of the gadget at node u, the 2 or 0 units of flow enter from before. The output is not relevant for an input of 1, since this would mean the all-or-nothing property was violated before the gadget. This flow is then transferred to v. Like this, v either has an excess of 3 or of 1. If the excess is 3, the transferring cycle containing v has to be used. The cycle contains edges, that come all the way from the start. These have to carry a flow of 2, so flow conservation holds in the transferring cycle. Like that we end up with 4 flow in node w. This is then transferred to x, where the non-shared edges with capacity 4 have to be used and 1 unit of flow goes onto the shared path and the invariant holds. If the excess in v is 1 on the other hand, the purple edge comes to play, forcing the flow to take the shared path. Here, and in all other gadgets, the purple edges always have a flow of 1, therefore if there is only one unit of flow in the whole gadget, it needs to take the shared path so it can use the purple edge. In the reduction we'll see how all purple edges are forced to have a flow of 1. Thus, to transfer flow away from v, only the shared path and the purple edge are used until reaching node x. There, again due to the second purple edge, flow has to take the shared path and not the dashed non-shared edges and the invariant holds. To summarize: If 2 units of flow enter through the dashed edges, 4 units of flow leave the gadget on the dashed edges on the right. If no flow enters through the dashed edges, no flow leaves through the dashed edges on the right either.

In the figure we only doubled a flow of 2. In the same way, any flow entering on the dashed edges can be doubled. Since we don't need additional vertices or edges if we double higher amounts of flow, this doubling can be used, to reach a value of 2^i with a graph of size $O(log(i))$. Figure 4.13 shows how the general doubling gadget looks like.

The next gadget we look at is the multi-output gadget that outputs 2^k on *two* outgoing edges in each non-shared network, if it gets an input of 2^k on its input edges or it outputs 0 on the output edges if there is no input. Again we look at an example first, then at the

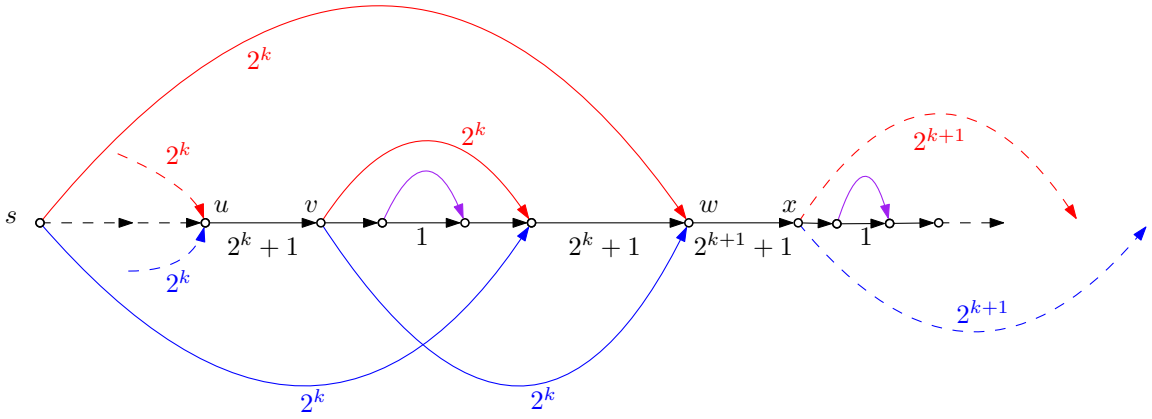

Figure 4.13.: Doubling gadget: This gadget ensures that either the flow exiting the dashed non-shared edges is 0 or it is 2^{k+1} depending on the dashed non-shared incoming edges having a flow of 0 or of 2^k. The purple edge has flow 1 in every case.

general case. In Figure 4.14 a multi output gadget for a flow of 4 is shown. If there is no flow entering at node u, through the dashed non-shared edges, there is only one unit of flow, that entered the gadget on the shared path. In this case, the purple edges ensure that flow stays on the the shared path for the whole gadget. Everywhere, where the one unit of flow could enter a transferring cycle or an output edge, the purple edges are positioned right after, to force the unit of flow to take the shared path. Thus, the invariant holds here. If the dashed non-shared edges have a flow of 4 entering u it will be transferred to v. At v, the excess of 5 has to use all non-shared edges to their full capacity and still use the shared path with one unit of flow. The non-shared edges v build two transferring cycles like we saw them in the doubling gadget in Figure 4.12. Like this, we get an excess of 5 at nodes w_0 and x_0. This is transferred to w_1 and x_1. The shared path leaving these nodes only has capacity 1 and the purple edges enforce this flow, the leftover excess of 4 needs to take the dashed non-shared output edges resulting in the output being the input of 4, twice on different sets of output edges. Since there is still a flow of 1 on the shared path, the invariant holds.

Figure 4.15 shows the general case of the multi-output gadget. Only the capacities have changed in comparison to Figure 4.14 and the argumentation why we get an output of 2^k on *two* outgoing edges in each non-shared network, if the gadget gets an input of 2^k is the same as above. k has to be greater or equal to 1 for this gadget to work, since we would have negative exponents for the capacities otherwise, which would make these edges unusable for integer flows.

Now we come to the reduction of SubSetSum to SimFlow.

Theorem 4.18. *If the shared network only consists of a simple path and the non-shared edges connect vertices in the shared network,* SimFlow *is NP-complete.*

Proof. We build an instance of SubSetSum with the gadgets we just introduced. All gadgets will be connected by the shared path in them To sum it up what the gadgets did: The doubling gadget used in Figure 4.13 produces an output of 2^{k+1} in each non-shared network if the input edges from the previous gadget carry a flow of 2^k. Otherwise it has flow 0 on the output edges. The invariant, of there being 1 unit of flow on the shared path after the gadget holds. The multi-output gadget used in Figure 4.14 outputs 2^k on two outgoing edges in each non-shared network if the input edges from the previous gadget

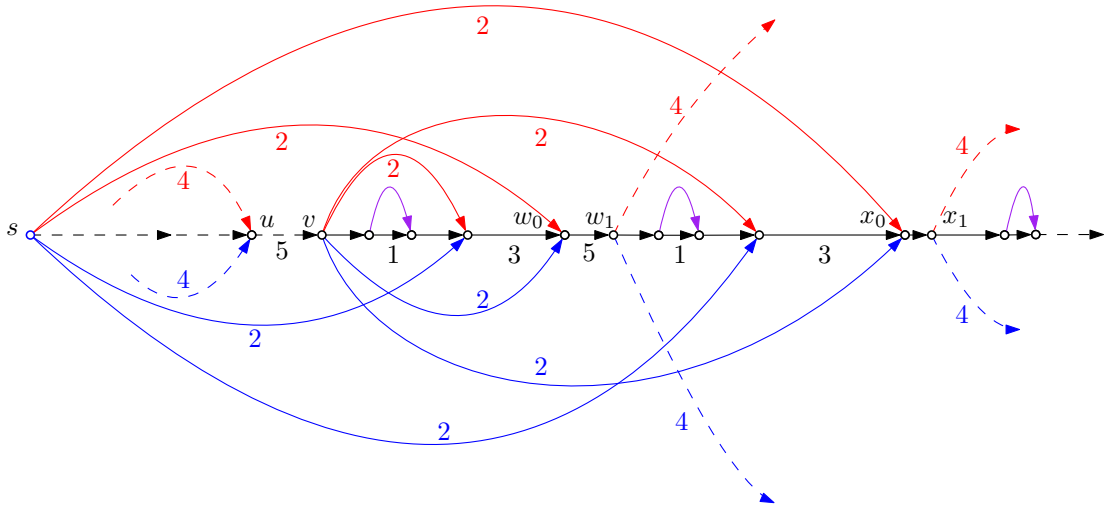

Figure 4.14.: Multi-output gadget example: This gadget ensures that either the flow exiting the two sets of dashed non-shared edges is 0 or it is 4 depending on the dashed non-shared incoming edges having a flow of 0 or of 4. The purple edge has flow 1 in every case.

carry a flow of 2^k. Otherwise it has flow 0 on the output edges. The invariant, of there being 1 unit of flow on the shared path after the gadget holds.

Given an instance of SubSetSum $I = (A, s)$. For each integer $i \in A$ from the set we model the binary representation and connect it to a final edge that has checks if the sum of flow of the gadgets is s.We'll connect outputs of gadgets to inputs of other ones. Thus, if one gadget is used, all others will be used as well. To choose which of these gadget concatenations to use, we insert a non-shared edge with capacity 1 in each non-shared network. Both edges start at a vertex on the shared path, that doesn't have any other shared edges leaving it and that lays before any of the gadgets on the path. The edges end as input to a doubling gadget. For the highest power of 2 in $i \in A$ we double the output of this doubling gadget with doubling gadgets connected output to input, until we reach the desired highest power. For all submaximal powers 2^k, we proceed as follows. We know, that we already have a gadget in our gadget concatenation somewhere, that outputs 2^k. So far, this output goes into a doubling gadget. Instead of this output we'll insert a multiple-output gadget on the path between the doubling gadgets. This multi-output gadget outputs 2^k twice. Like that, we can use one output and connect in to the doubling gadget, the output of 2^k was connected to before and the other to model the binary representation of i by connecting this output to the last node on the path before the sink. The last shared edge has capacity $s + 1$. If there is a simultaneous flow that uses it to its full capacity, there are fully used gadget concatenations that produce a sum of s flow. Since these gadgets represent elements in A we would have a subset, that solves SubSetSum.

We now look at how exactly the purple edges work, that assure the invariant holds even if a gadget is not used. Theoretically it would be possible for the one unit of flow on the shared path, to use the two non-shared output edges of an unused gadget with a flow of 1. Like that, the all-or-nothing property of our gadget combination would be compromised. Like we shaw when introducing the gadgets, we want the purple edges to all have a flow of 1 so this can't happen. To ensure that, we insert a transferring cycle with capacity 1, so they all have the same flow. How this transferring cycle looks like can be seen in Lemma 4.10. To ensure that the transferring cycle is used, we add one more purple edge from before the first gadget to the sink. Now, we only accept SubSetSum if the simultaneous flow in our graph is value of SubSetSum $s + 2$. The $+2$ come from the 1 unit of flow

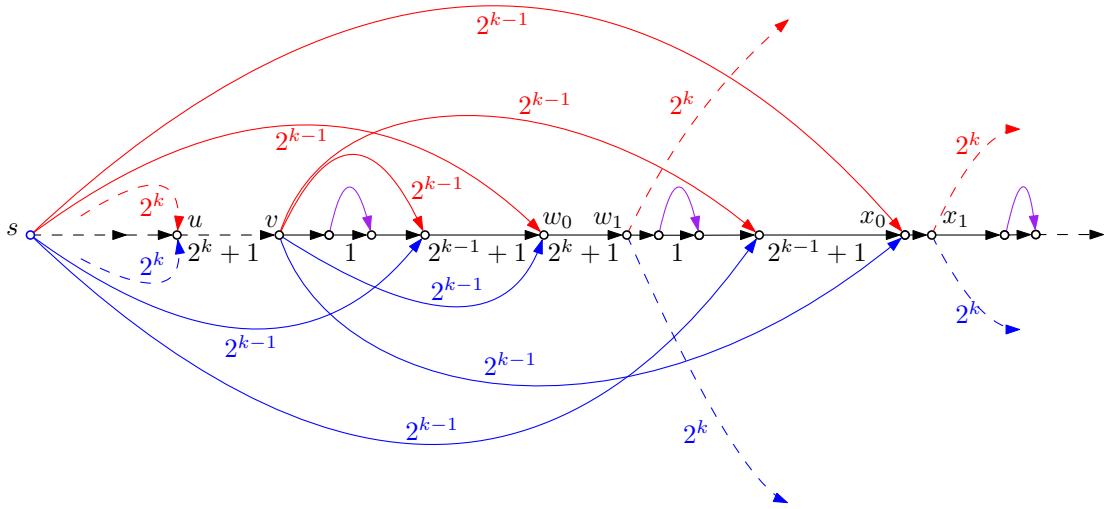

Figure 4.15.: Multi-output gadget: This gadget ensures that either the flow exiting the two sets of dashed non-shared edges is 0 or it is 2^k depending on the dashed non-shared incoming edges having a flow of 0 or of 2^k. The purple edge has flow 1 in every case.

on the path and from the one unit of flow through the purple edges. Like this, if there is a solution of SimFlow that has a flow of $s+2$ in G_1 and in G_2, the last edge on the path is used to its fully capacity. This means, there are fully used gadgets representing $a \in A' \subseteq A$ with $\sum_{a \in A'} a = s$ connected to the start of this last edge. Thus, it is a solution for SubSetSum. If there is no such solution for SimFlow, there is no gadget combination that yields a flow of s, thus there is no subset for SubSetSum either. Figure 4.16 shows an example of this construction for a set that only consists of the integer 4. □

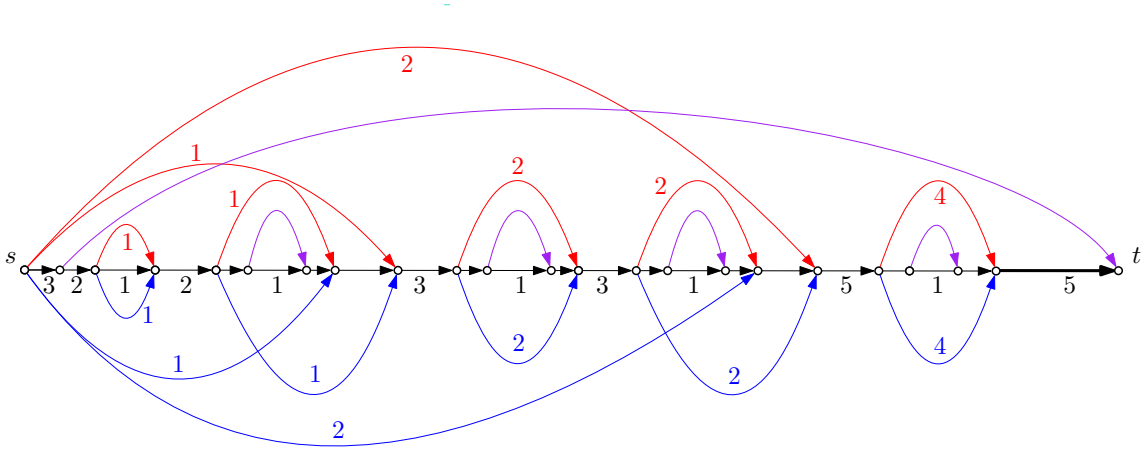

Figure 4.16.: Example for the construction used in the reduction of SubSetSum to SimFlow. In this example, the set only consists of the integer 4 and we want to decide if there is a subset with value 4. Thus, we decide SimFlow with a flow of 7 on both graphs.

5. Inapproximability of MaxSimFlow

In this chapter we are going to look at the optimization variant of SimFlow called MaxSimFlow. In the previous section on NP-completeness we saw that still quite restricted instances are strongly NP-complete. As we'll see, this already shows that there doesn't exist a fully polynomial approximation scheme (FPTAS). Continuing with inapproximability of MaxSimFlow, we'll use that MaxIntegerEqualFlow is polynomially equivalent to MaxSimFlow, that means, MaxSimFlow can be reduced to MaxIntegerEqualFlow and vice versa while preserving approximation properties of a solution. This will lead to $2^{n(1-\epsilon)}$-inapproximability. Due to this polynomial equivalence, we'll use results of [MS09], a paper on MaxIntegerEqualFlow, several times this chapter.

Theorem 5.1. *There doesn't exist a FPTAS for* SimFlow*, even if the shared network is connected and the same in all $G_i = (V, E_{shared} \cup E_i)$, the non-shared edges connect vertices in the shared network, all G_i are planar, and especially the graph $G = (V, E_{shared} \cup \bigcup E_i)$ with the edge set being the union of the edge sets of all G_i is planar.*

Proof. In Theorem 4.15 we have shown that SimFlow with this restrictions is strongly NP-complete. In [Vaz03] it is shown, that there doesn't exist a FPTAS for strongly NP-complete problems. □

Optimization Problem 5.2. MaxIntegerEqualFlow*: Given a directed graph $G := (V, E)$ with specific vertices s and t, (source and sink) and a capacity function $c : E \to \mathbb{N}$. Additionally we have a set $\mathcal{R}$ of disjoint sets of edges, so called equal flow edges. We want to maximize the s-t flow value h of $f : E \to \mathbb{N}_0$ with $0 \leq f(e) \leq c(e)$ $(e \in E)$ where for each $R_k \in \mathcal{R}$, $\forall e, e' \in R_k \ f(e) = f(e')$.*

MaxIntegerEqualFlow is an integer flow problem as well. In contrast to MaxSimFlow which looks at m graphs with each having shared edges that has the same flow in all graphs, MaxIntegerEqualFlow looks at *one* graph that has multiple edge-sets $R_k \in \mathcal{R}$ that all must have the same flow. If the capacity of the equal flow edge sets is two, $|R_k| = 2$ for all $R_k \in \mathcal{R}$ we call the optimization problem MaxHomArcFlow as well.

MaxHomArcFlow is the optimization variant of HomArcFlow. We have already reduced HomArcFlow to SimFlow (see Theorem 4.8). Similar to the reduction of HomArcFlow to SimFlow in Theorem 4.8 we'll now proceed with MaxIntegerEqualFlow. We'll use the generalization of our previous gadget to arbitrary large sets of edges with the same flow that is described in Lemma 4.10 and Figure 4.7 for 3 edges.

Lemma 5.3. MaxIntegerEqualFlow *can be reduced to* MaxSimFlow *in polynomial time while preserving approximation properties.*

Proof. Given an instance $I = (V, E, \mathcal{R})$ of MaxIntegerEqualFlow, we want to construct an instance I' of MaxSimFlow. We proceed similar to the construction used for HomArcFlow. The shared network of I' is the network $(V, E \setminus \bigcup_{\mathcal{R}} R_k)$ that means the network of I without the equal flow edges. Applying Lemma 4.7 to all $R \in \mathcal{R}$ we now add non-shared edges as replacement for previous edges from each R. This ensures, that the flow is the same on all non-shared edges replacing previous edges from an R. I' now is this combination of shared and non-shared edges. Since an optimal flow for MaxSimFlow on I' would guarantee all equal edges to have the same flow and be on the same network, it would be optimal for MaxIntegerEqualFlow on I as well. Figure 5.1 shows an example construction of I'. □

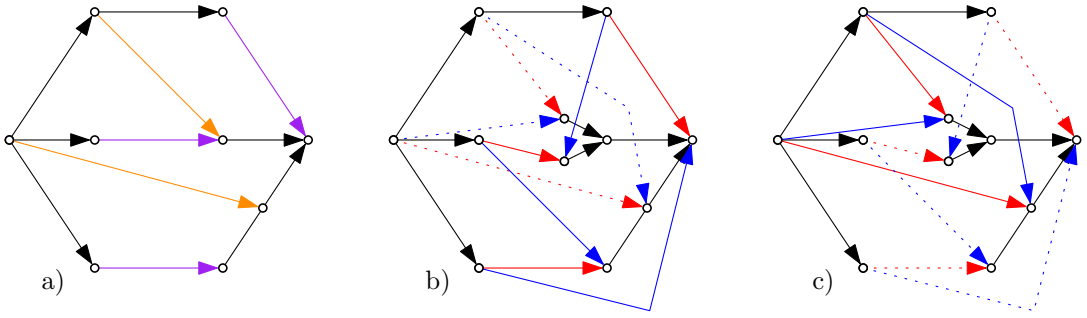

Figure 5.1.: a) is an instance of MaxIntegerEqualFlow with the colored edges being the two sets of equal flow edges. b) and c) are the instance of MaxSimFlow with the red edges being the non-shared edges of G_1, the blue of G_2. b) highlights the non-shared edges added for the purple equal flow edges in a), c) highlights the orange equal flow edges.

Lemma 5.4. MaxSimFlow *can be reduced to* MaxIntegerEqualFlow *in polynomial time while preserving approximation properties.*

Proof. Given an instance $I = \{G_1, G_2, ..., G_m\}$, each $G_i := (V_i, E_i \cup E^i_{shared})$, with sources and sinks $s_1, s_2, ..., s_m$ and $t_1, t_2, ..., t_m$ of MaxSimFlow, we want to construct an instance I' of MaxIntegerEqualFlow. All V_i and E^i_{shared} all have the same vertices or edges respectively for all $i \in M := \{1 \leq i \leq m\}$ but for copying them later in this proof, the index i is added to each vertex and edge. We add a new source s and a new sink t then we take all G_i from MaxSimFlow and put them together in one graph, as seen in Figure 5.2. All previous sources s_i from the G_i are connected to the new source s with unlimited capacity and all previous sinks t_i are connected to the new sink t. All previously shared edges are put in sets $R_e := \bigcup_{i \in M} e_i$, one set for each shared edge $e \in E^1_{shared}$. All together we get an instance $I' = (\bigcup_{i \in M} V_i, \bigcup_{i \in M}(E^i_{shared} \cup E_i), \bigcup_{e \in E^1_{shared}} R_e)$ of MaxIntegerEqualFlow. If we find an optimal solution for this, it is a feasible flow, that has the same flow on all shared edges. The s-t flow value is the sum of s_i-t_i flow values of the G_i, which is exactly definition 2.6, therefore we preserve approximation properties.

If we only have two G_i this obviously leads to an instance of MaxHomArcFlow, thus this reduction can be used as well.

□

Theorem 5.5. *There is no polynomial* $2^{n(1-\epsilon)}$*-approximation algorithm for any fixed* $\epsilon > 0$ *for* MaxSimFlow *even when restricted to two-network instances, unless* $P = NP$.

Proof. Lemma 5.3 together with [MS09] proves the theorem. □

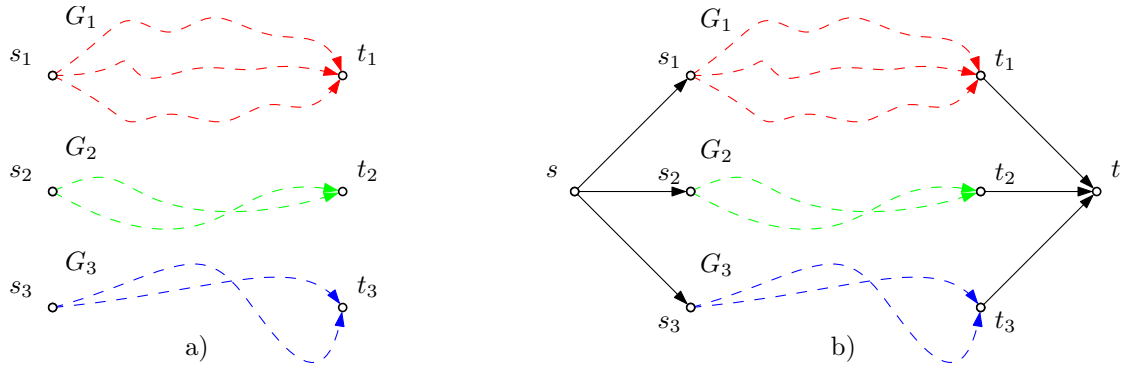

Figure 5.2.: Reduction of MaxSimFlow to MaxIntegerEqualFlow as described in Lemma 5.4 a) is the instance of MaxSimFlow, b) the one of MaxIntegerEqualFlow.

Like already mentioned in the introduction, the heuristics for MaxHomArcFlow from [LL98] and [AKS88] can be used to solve MaxSimFlow on two-network instances as well. To do so, we take the instance I of MaxSimFlow and use the construction from Lemma 5.1. The resulting instance of I' of of MaxHomArcFlow can be solved with one of the heuristics then. [LL98] claims, that his heuristic can be extended to solve the general case of MaxIntegerEqualFlow which would lead to a heuristic for the general case of MaxSimFlow as well.

6. FPT

In this chapter, we show that MinCostSimFlow is fixed-parameter tractable (FPT), in Section 6.1 for the number of shared edges as the parameter, in Section 6.2 for the intersections between the shared and non-shared network or the number of non-shared edges as the parameter.

6.1. FPT in Shared Edges

Decision Problem 6.1. MinCostIntegerEqualFlow*: Given a* MaxIntegerEqualFlow *instance, see Definition 5.2. Given a demand d_i for each node, weakening the flow conservation of node i to $\sum_{j:e_{ij}\in E} f(e_{ij}) - \sum_{j:e_{ji}\in E} f(e_{ji}) = d_i$. We also add a cost $b(e_{ij})$ for each unit of flow on an edge. We want to decide if there exists a flow satisfying the new flow conservation constraint and is equal on all equal flow edges in each node with a total cost $\sum_{e_{ij}\in E} f(e_{ij}) \cdot b(e_{ij})$ equal to a value b.*

Lemma 6.2. *The* MinCostIntegerEqualFlow *is FPT with the number of shared edges being the fixed parameter.*

Proof. In [MS09] it is shown, that MinCostIntegerEqualFlow is solvable in polynomial time for any fixed number of equal flow sets. There it is solved with a mixed ILP, minimizing $\sum_{e_{ij}\in E} f(e_{ij}) \cdot b(e_{ij})$. If we instead minimize $|\sum_{e_{ij}\in E} f(e_{ij}) \cdot b(e_{ij}) - b|$ we decide yes, if the minimum is 0, no otherwise. This change doesn't affect polynomial runtime and decides MinCostIntegerEqualFlow. Therefore, MinCostIntegerEqualFlow is FPT. □

We'll now see that this shows that the generalized version of SimFlow, MinCostSimFlow (see Definition 2.5) is FPT, with the number of shared edges being the fixed parameter.

Theorem 6.3. MinCostSimFlow *is FPT for the number of shared edges being the fixed parameter.*

Proof. Lemma 6.2 shows, that MinCostSimFlow is FPT. We proved in Lemma 5.4, SimFlow can be reduced to MaxIntegerEqualFlow while having the same flow. This exact construction can be reused here to reduce MinCostSimFlow to MinCostIntegerEqualFlow. If we now have an instance of MinCostSimFlow, we just construct an instance of MinCostIntegerEqualFlow and solve it with the corresponding FPT algorithm. Since the reduction does not add any shared edges and just uses the shared edges to construct the sets of equal-flow-edges, the number of shared edges is the fixed parameter for SimFlow in FPT. □

6.2. FPT in Intersections

After seeing, for example in Lemma 3.6 or Figure 3.3, that we can enforce two or multiple edges to have the same flow using non-shared edges in transferring cycles, one might assume that we can use this construction to just use non-shared edges instead of shared edges and get somewhere close to polynomial runtime. The problem about this is, that the gadget only works if all these non-shared edges start and end at nodes in the shared network and the transferring cycles only intersect once, as we saw in 3.7. This is also why in Lemma 4.10 and Figure 4.7 use additional shared edges. To be certain that none of the transferring cycles intersect and thus all edges have the same flow for sure, each transferring cycle needs at least 4 shared edges.

To continue looking at FPT, we can not only solve SimFlow polynomially with a fixed number of shared edges, but also for a fixed number of non-shared edges. Therefore, we are going to build a mixed ILP that has an amount of integer variables limited by the number of non-shared edges. As shown in [Len83] this mixed ILP can be solved with exponential time in the number of integer variables and polynomial in the rest. Like that we have an FPT algorithm if the ILP solves our problem down to polynomial pre- and post calculations. The integer variables are going to be the value of flow going from one network to the other, from the shared to the non-shared, to be precise. The number of nodes that are in both, shared and non-shared network, is limited by two times the number of non-shared edges, since every edge only has a starting and an ending point. Thereby, the number of integer variables is limited by two times the amount of non-shared edges as well.

It of course is limited by two times the number of shared edges as well, so it yields a FPT algorithm in the number of shared edges too, but the one above might have better runtime depending on the structure of the graphs.

Theorem 6.4. SimFlow *is FPT for the number of non-shared edges or the number of intersections between shared and non-shared network being the fixed parameter.*

Proof. In the following, b_{ij} is the cost of flow on an edge, d_i is the demand of a node and y_i represents the amount of flow going from non-shared to shared network in an intersection node. Therefore, y_i is 0 if a node is not in the intersecting set of the shared and non-shared network and an integer otherwise. The y_i respect the demand of the nodes, that means, for a node i they are $y_i = \textit{sum of shared outflux} - \textit{sum of shared influx} - d_i$. But they also are $-y_i = \textit{sum of non-shared outflux} - \textit{sum of non-shared influx}$, assuring that all flow that doesn't leave through the shared network goes to the non-shared. We can use one y_i per node because it is the same in all graphs, since the shared network is the same in all graphs, thus the influx to the shared network ($= y_i$) has to be the same everywhere. We now give the formal mixed ILP. Only the y_i are integer variables, thus we only have exponential running time in the number of intersections of the two networks.

$$
\begin{aligned}
&\min \sum_{(i,j)\in E} b_{ij} f_{ij} \\
\text{s.t.} \quad &\sum_{j:(i,j)\in E_{shared}} f_{ij} - \sum_{j:(j,i)\in E_{shared}} f_{ji} - d_i = y_i \\
&\sum_{j:(i,j)\in E_{non-shared}} f_{ij} - \sum_{j:(j,i)\in E_{non-shared}} f_{ji} = -y_i \text{ for all graphs} \\
&y_i \in \mathbb{Z} \\
&0 \le f_{ij} \le c_{ij}
\end{aligned}
$$

To prove that this yields a correct solution we add the first two equations. This leaves us with $\sum_{j:(i,j)\in E} f_{ij} - \sum_{j:(j,i)\in E} f_{ji} - d_i = 0$ which is just the constraint of flow conservation. This means, a solution will be a feasible real flow, with y_i, the amounts of flow going from non-shared to shared in the node i. Given the existence of a feasible simultaneous min-cost flow, such y_i always exists, because taking this simultaneous flow, we can simply insert the flow and demand values in the equations above and solve for y_i. But so far, the f_{ij} are not necessarily integers. We'll now use the integrality property for standard min-cost-max-flow for the shared and each of the non-shared networks. Since the y_i are integers, we can solve a min-cost-max-flow problem for the shared network, and one for each non-shared network, that contains the demands as follows. If the node i is not in the intersecting set of shared and non-shared network, the demand is simply d_i, which works just fine, because the node is not in the intersection and each network has to take care of this demand itself. If the node i is in the intersecting set, the demand is $y_i + d_i$ in the shared network and $-y_i$ in the non-shared networks. This just models, what we wanted: the y_i were supposed to be the outflux of the non-shared and the influx in the shared network with respect to the original demands. Since all capacities and costs are integers there exists an integer optimal solution with the same cost, which can be found in polynomial time with min-cost-max-flow. Since this is only done once for the shared network, all shared edges have the same flow. Like this, we have found an integer solution that still satisfies the ILP, thus flow conservation and has the same flow on all shared edges. That means the existence of a solution for the mixed ILP is the decision for MinCostSimFlow. The later min-cost-max-flow calculations yield the corresponding simultaneous min-cost-max-flow. The polynomial running time of the min-cost-max-flow calculation together with the potentially exponential runtime in the number of non-zero y_i for the mixed ILP show that MinCostSimFlow is FPT in the number of intersections. □

7. Conclusion

In this thesis we have looked at simultaneous integer flows from several perspectives. We started by comparing them to standard flows, to see that the min-cut max-flow duality, the existence of augmenting paths and the integrality property of maximum flows are all lost when transitioning to the simultaneous case. We then reduced 3Sat, HomArcFlow, Planar monotone 3Sat and SubSetSum to SimFlow to see that deciding if a simultaneous flow with a certain value exists is hard to decide in various restricted settings. In Chapter 5 on inapproximaility we saw that there is no polynomial $2^{n(1-\epsilon)}$-approximation algorithm for any fixed $\epsilon > 0$ for MaxSimFlow. We finished off by showing that MinCostSimFlow is FPT in the number of intersections between the shared and non-shared network.

Bibliography

[ABF+10] *Patrizio Angelini, Giuseppe Di Battista, Fabrizio Frati, Vít Jelínek, Jan Kratochvíl, Maurizio Patrignani, and Ignaz Rutter.* Testing planarity of partially embedded graphs. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 202–221. SIAM, 2010.

[AKS88] *Agha Iqbal Ali, Jeff Kennington, and Bala Shetty.* The equal flow problem. *European Journal of Operational Research*, 36(1):107–115, 1988.

[AMO88] *Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin.* Network flows. 1988.

[BKM+17] *Glencora Borradaile, Philip N Klein, Shay Mozes, Yahav Nussbaum, and Christian Wulff-Nilsen.* Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. *SIAM Journal on Computing*, 46(4):1280–1303, 2017.

[BKR12] *Thomas Bläsius, Stephen G. Kobourov, and Ignaz Rutter.* Simultaneous Embedding of Planar Graphs. *ArXiv*, abs/1204.5853, 2012.

[BR16] *Thomas Bläsius and Ignaz Rutter.* Simultaneous PQ-ordering with applications to constrained embedding problems. *ACM Transactions on Algorithms (TALG)*, 12(2):16, 2016.

[CC01] *Eleonor Ciurea and Laura Ciupal.* Algorithms for minimum flows. *The Computer Science Journal of Moldova*, 9(3):275–290, 2001.

[CLRS01] *Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein.* Introduction to algorithms second edition. *The Knuth-Morris-Pratt Algorithm, year*, 2001.

[dBK10] *Mark de Berg and Amirali Khosravi.* Optimal Binary Space Partitions in the Plane. In My T. Thai and Sartaj Sahni, editors, *Computing and Combinatorics*, pages 216–225, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[EH06] *Alexander Engau and Horst W Hamacher.* Semi-Simultaneous Flows and Binary Constrained (Integer) Linear Programs. 2006.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[HKRS97] *Monika R Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian.* Faster shortest-path algorithms for planar graphs. *journal of computer and system sciences*, 55(1):3–23, 1997.

[Kar72] *Richard M Karp.* Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[KHA79] *L. G. KHACHIYAN.* A polynomial algorithm in linear programming. *Doklady Academii Nauk SSSR*, 244:1093–1096, 1979.

[KKV11] *Pavel Klavík, Jan Kratochvíl, and Tomáš Vyskočil.* Extending partial representations of interval graphs. In *International Conference on Theory and Applications of Models of Computation*, pages 276–285. Springer, 2011.

[Kri09] *Manuel Krings.* Simultane Schnitte in Graphen, 2009.

[Len83] *H. W. Lenstra.* Integer Programming with a Fixed Number of Variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[LL98] *TorjÖrn Larson and Zhuangwei Liu.* An efficient Lagrangean relaxation scheme for linear and integer equal flow problems. *Optimization*, 44(1):49–67, 1998.

[MS09] *Carol A. Meyers and Andreas S. Schulz.* Integer equal flows. *Operations Research Letters*, 37(4):245 – 249, 2009.

[Orl13] *James B Orlin.* Max flows in O (nm) time, or better. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 765–774. ACM, 2013.

[Sah74] *S. Sahni.* Computationally Related Problems. *SIAM Journal on Computing*, 3(4):262–279, 1974.

[Sch02] *Alexander Schrijver.* On the history of the transportation and maximum flow problems. *Mathematical Programming*, 91(3):437–445, 2002.

[Vaz03] Vijay V. Vazirani. *Approximation algorithms.* Springer, Berlin, corr. 2. print. edition, 2003.

Appendix

A. Planar monotone 3Sat Gadget Variants

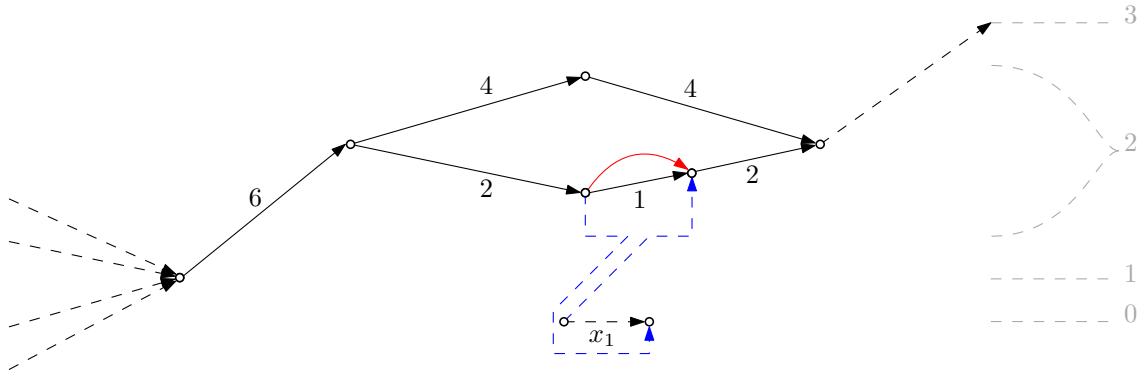

Figure A.1.: Construction of planar clause for clauses with only 1 literal.

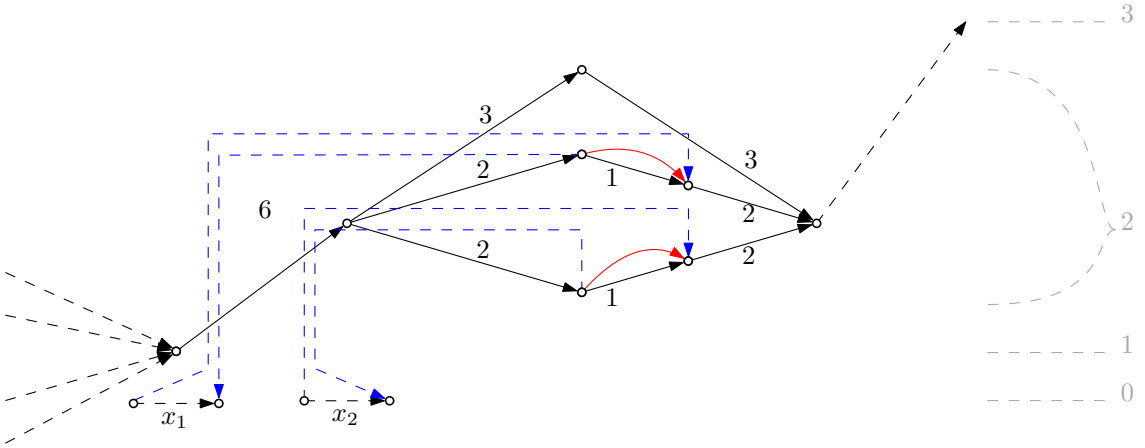

Figure A.2.: Construction of planar clause for clauses with only 2 literals.