Technical University Berlin
Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Algorithmics and Computational Complexity (AKT)

Parameterized Algorithms for Network Flows

Leon Kellerhals

Thesis submitted in fulfillment of the requirements for the degree
"Master of Science" (M. Sc.) in the field of Computer Science

June 2018

Supervisor and first reviewer:	Prof. Dr. Rolf Niedermeier
Second reviewer:	Prof. Dr. Martin Skutella
Co-Supervisors:	Till Fluschnik and Dr. André Nichterlein

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt

Berlin, ____________________ ____________________

Leon Kellerhals

Abstract

Computing maximum flows in directed graphs is one of the most fundamental and applied network flow problems. For graphs with m arcs, n vertices and capacities of size at most C, Lee and Sidford [FOCS '14] provided an $\tilde{\mathcal{O}}(m\sqrt{n}\log^2 C)$-time algorithm, but for several applications this time is still too slow.

In the spirit of the *FPT in P* program introduced by Giannopoulou et al. [Theoretical Computer Science '17] this work analyzes parameterizations for Maximum Flow. Our results are of two kinds. One the one hand, we show that many parameters such as maximum degree or the vertex deletion distance to complete graphs do not help to solve the problem more quickly. On the other hand we provide "good news": First, we provide quasilinear-time applicable data reduction rules that yield a kernel of size linear in the *undirected feedback edge number* (i.e., the edge deletion distance from the undirected input graph to forests). This is, to the best of our knowledge, the first nontrivial kernelization result for Maximum Flow. Second, we introduce a systematic approach for designing fixed-parameter algorithms for Maximum Flow. This approach is based on ideas by Hochstein and Weihe [SODA '07] and generalizes the Push-Relabel method by Goldberg and Tarjan [Journal of the ACM '88]. By applying our approach with the *underlying feedback vertex number k* (i.e., the vertex deletion distance from the undirected input graph to forests) we can solve the problem in $\mathcal{O}(k^4 m)$ time.

Zusammenfassung

Die Berechnung von größtmöglichen Flüssen auf gerichteten Graphen (Maximum Flow) ist eines der fundamentalen Netzwerkflussprobleme. Der Algorithmus von Lee und Sidford [FOCS '17] ist mit einer Laufzeit von $\mathcal{O}(m\sqrt{n}\log^2 C)$ auf Graphen mit m gerichteten Kanten, n Knoten und Maximalkapazität C der zur Zeit schnellste, jedoch genügt diese Laufzeit nicht den Ansprüchen aller Anwendungen.

In Anlehnung an das *„FPT in P"*-Projekt von Giannopoulou et al. [Theoretical Computer Science 17] analysieren wir in dieser Arbeit Parametrisierungen für Maximum Flow. Wir präsentieren zwei Arten von Ergebnissen. Einerseits zeigen wir, dass viele Parameter – zum Beispiel der Maximalgrad oder die Knotenlöschungsdistanz zu vollständigen Graphen – nicht zu einem schnelleren Algorithmus führen können. Andererseits – „positiv" – zeigen wir zweierlei: Erstens zeigen wir in quasilinearer Zeit anwendbare Datenreduktionsregeln, die zu einem Kern führen, dessen Größe linear in der *ungerichteten kreiskritischen Kantenzahl* ist, d.h. die kleinste Anzahl an Kanten, die aus der ungerichteten Kopie des Eingabegraphen zu entfernen sind, damit ein Wald entsteht. Nach unserem Wissensstand ist dies das erste Resultat, das einen (nicht-trivialen) Problemkern für Maximum Flow zeigt. Zweitens führen wir ein systematisches Verfahren zur Konzeption von parametrisierten Algorithmen für Maximum Flow ein, das auf Ideen von Hochstein und Weihe [SODA 07] basiert und die „Push-Relabel"-Methode von Goldberg und Tarjan [Journal of the ACM 88] verallgemeinert. Mit der Anwendung dieses Verfahrens können wir einen Algorithmus für das Problem mit einer Laufzeit von $\mathcal{O}(k^4 m)$ ableiten, wobei k die *ungerichtete kreiskritische Knotenzahl* ist, d.h. die kleinste Anzahl an Knoten, die – analog der ungerichteten kreiskritischen Kantenzahl – zu entfernen sind, damit ein Wald entsteht.

Contents

Chapter 1

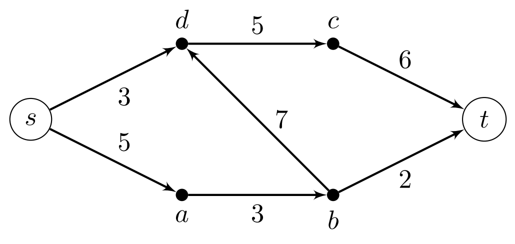
Introduction

“Networks are pervasive” [AMO93], and the applications for network flow problems are endless. Transport networks and scheduling problems, but also fundamental problems such as Bipartite Matching and Minimum Path Cover are some exemplary problems that can be solved with network flows. We refer to Ahuja, Magnanti, and Orlin [AMO93] for an overview of applications. Probably the most fundamental variant of the network flow problems is the polynomial-time solvable Maximum Flow problem: Given a directed graph where each arc has a flow capacity, and two terminals s and t, the task is to send as much flow as possible from s to t while honoring the flow capacities and ensuring that at no vertex (other than s and t) any excess flow may gather. We depict in Figure 1.1 an exemplary instance of Maximum Flow.

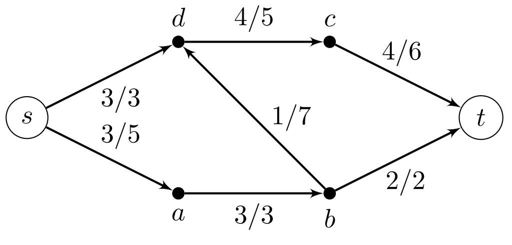
Starting with the seminal algorithm by Ford and Fulkerson [FF56], finding efficient algorithms for Maximum Flow has been an active research topic now for six decades, and the problem does not cease to attract plenty of researchers. The upper bound on the running time is being improved continuously. But due to the amount of information stored in the world growing exponentially over time [HL11], the demand on solving network problems such as Maximum Flow more quickly has grown over time as well. From a practitioner’s point of view, a linear-time algorithm would be ideal, but neither are the best known algorithms close to running in linear time, nor do we know whether such an algorithm is possible for Maximum Flow. In the last decade a new research direction for Maximum Flow has emerged in which one trades optimal solutions for better running times [Chr+11; LRS13; She13]. For instance, Kelner et al. [Kel+14] presented an algorithm that finds a $(1-\varepsilon)$-approximation for Maximum Flow on undirected m-edge graphs in $\mathcal{O}(m^{1+o(1)} \cdot \varepsilon^{-2})$ time.

Our goal is to develop more efficient algorithms without giving up the quest for optimal solutions. In the young and upcoming field of of *FPT in P* one pursuits “fixed-parameter algorithms for problems already known to be solvable in polynomial time” [GMN17], that is, for problems with instance size x solvable in $\mathcal{O}(x^\alpha)$ time, one tries to find algorithms running in $f(k) \cdot x^\beta$ time, where k is a parameter, f is a computable function and $\beta < \alpha$. We want to investigate the Maximum Flow problem in the spirit of this field and try to find algorithms with running times of the kind $f(k) \cdot \tilde{\mathcal{O}}(x)$.[1] Further, we want to provide efficient (i.e., (quasi-)linear-time solvable) *data reduction*

[1]The notation $\tilde{\mathcal{O}}(f) \coloneqq \mathcal{O}(f \log^c f)$ for constant c hides polylogarithmic factors in the running time.



(a) The input.



(b) A maximum flow.

Figure 1.1: An exemplary instance of Maximum Flow. The arcs are labeled with their capacity in (a) and with their flow/capacity in (b).

rules—algorithms that shrink the size of instances to some upper bound, typically a function depending only on the size of a parameter. In the research field on algorithms for NP-hard (decision) problems, this is known as *kernelization*. We also want to show that certain parameters "do not help" to solve Maximum Flow more efficiently, even if they are small. While these might first sound just like negative "intractability" results, they also give insight into the Maximum Flow problem that might help designing efficient algorithms. For example, by reducing any instance of Maximum Flow to an instance with maximum degree three in linear time, one can see that a fixed-parameter algorithm with respect to the maximum degree does not help, and when designing new algorithms for Maximum Flow one can assume that the maximum degree of the input graph is three.

Note that in this work we will restrict ourselves to the Maximum Flow problem with integer capacities.

Related work. We start off by giving a condensed overview over the history of the best algorithms for Maximum Flow. Ahuja, Magnanti, and Orlin [AMO93] discuss many algorithms and applications for Maximum Flow in detail. Goldberg and Tarjan [GT14] provide a compact and recent survey on the development of Maximum Flow and cover several of the current research directions for the problem. Goldberg and Rao [GR98] provide a detailed list of Maximum Flow algorithms that shows how the best running time upper bounds have improved over time. Table 1.1 shows a selection of these results as well as two remarkable, more recent findings. Orlin [Orl13] provides an $\mathcal{O}(nm + m^{31/16} \log^2 n)$-time algorithm, that, among other things, calls the algorithm by Goldberg and Rao [GR98] as a subroutine. Calling the latter algorithm if the graph is sparse, and the algorithm by King, Rao, and Tarjan [KRT94] otherwise, allows to bound the running time of Maximum Flow by $\mathcal{O}(nm)$. Lee and Sidford [LS14] presented a new algorithm for efficiently solving linear programs and have shown that applying it to the linear programming formulation of Maximum Flow yields the currently best known running time upper bound for Maximum Flow.

While the running time upper bound for Maximum Flow improved over time, there are no nontrivial results on running time lower bounds for Maximum Flow. Contrarily, Krauthgamer and Trabelsi [KT18] showed that if one finds a lower bound for

Table 1.1: Running time upper bounds for the MAXIMUM FLOW problem on directed graphs with n vertices, m arcs and a maximum arc capacity of C.

YEAR	WORST CASE RUNNING TIME	REFERENCES
1956	$\mathcal{O}(nmC)$	Ford and Fulkerson [FF56]
1970	$\mathcal{O}(nm^2)$	Dinic [Din70], Edmonds and Karp [EK72]
1970	$\mathcal{O}(n^2m)$	Dinic [Din70]
1973	$\mathcal{O}(nm \log C)$	Dinic [Din73], Gabow [Gab85]
1974	$\mathcal{O}(n^3)$	Karzanov [Kar74]
1988	$\mathcal{O}(nm \log(n^2/m))$	Goldberg and Tarjan [GT88]
1994	$\mathcal{O}(nm \log_{m/(n \log n)} n)$	King, Rao, and Tarjan [KRT94]
1998	$\mathcal{O}(\min\{n^{2/3}, \sqrt{m}\} m \log(n^2/m) \log C)$	Goldberg and Rao [GR98]
2013	$\mathcal{O}(nm)$	Orlin [Orl13]
2014	$\tilde{\mathcal{O}}(m\sqrt{n} \log^2 C)$	Lee and Sidford [LS14]

MAXIMUM FLOW with one of the prominent techniques, fine-grained reductions [VW15] in combination with the Strong Exponential Time Hypothesis (SETH) [IPZ01], then the Nondeterministic Strong Exponential Time Hypothesis would be refuted. In contrast, Krauthgamer and Trabelsi [KT18] showed that ALL PAIRS MAXIMUM FLOW—given a capacitated graph D, compute the maximum v–w-flow for every pair of vertices $v, w \in V(D)$—cannot be solved in $\mathcal{O}(n^{3-\varepsilon})$ time under the assumption that the SETH holds.

This work is in spirit of the field of fixed-parameter algorithms for polynomial-time solvable problems (FPT in P) which was initiated by Giannopoulou, Mertzios, and Niedermeier [GMN17]. The main goal of this field is to find fixed-parameter algorithms that outperform the best unparameterized algorithms if the parameter is sufficiently small. Exemplary problems that have been studied in the spirit of FPT in P include DIAMETER [AWW16; BN18], GRAPH HYPERBOLICITY [Flu+17] and TRIANGLE ENUMERATION [Ben+17]. One of the most studied problems in the field of FPT in P, and thus possibly the coming up “drosophila”[2] of the field, is the MAXIMUM MATCHING problem. There are fully polynomial fixed-parameter algorithms (i.e., the running time is at most polynomial in the parameter) with respect to the parameters difference between maximum and minimum degree [Yus13], treewidth [Fom+17], modular width [KN18] and treedepth [IOO17]. The latter of the four results also holds for the weighted variant of the problem. Mertzios, Nichterlein, and Niedermeier [MNN17] showed that MAXIMUM MATCHING admits a linear-size kernel with respect to the feedback edge number and an exponential-size kernel with respect to the feedback vertex number. They also showed that the BIPARTITE MAXIMUM MATCHING problem—a special case of MAXIMUM FLOW—admits a cubic-size kernel with respect to the vertex deletion distance to chain graphs.

For MAXIMUM FLOW there also exist fixed-parameter algorithms (see Table 1.2). Chambers, Erickson, and Nayyeri [CEN12] provided a fixed-parameter algorithm for MAXIMUM FLOW with respect to the genus of the input graph. For a generalization of

[2]The fruit fly “Drosophila melanogaster” is one of the most-studied organisms in the field of genetics [Hal+15]. Niedermeier [Nie06] called the VERTEX COVER the “drosophila” of FPT studies [Nie06].

Table 1.2: Fixed-parameter algorithms for the Maximum Flow problem as well as so-called "hardness" results that show that Maximum Flow parameterized by the parameter does not become easier than its unparameterized version.

Parameter k	Worst case running time	Reference
Crossing number	$\mathcal{O}(k^3 n \log n)$	[HW07]
Genus	$\mathcal{O}(k^8 n \log^2 n \log^2 C)$	[CEN12]
Treewidth	$\mathcal{O}(2^{\mathcal{O}(k^2)} n)$	[Hag+98]
Vertex cover number	$\mathcal{O}(k^3 m)$	Thm 5.20
(Undirected) feedback vertex number	$\mathcal{O}(k^4 m)$	Cor 5.25
(Undirected) feedback edge number	$\mathcal{O}(k^{3/2} \log^2 C + (n+m) \log n)$	Cor 3.15
Domination number		Prop 4.3
Bisection number		Prop 4.4
Degeneracy	not easier to solve than the	Prop 4.5
Distance to bipartite graphs	unparameterized version	Cor 4.6
Maximum degree		Thm 4.7
Distance to complete graphs		Thm 4.13

Maximum Flow called Multi-Terminal Flow, Hagerup et al. [Hag+98] presented a fixed-parameter algorithm with respect to the treewidth of the input graph. Lastly, based on the Push-Relabel method by Goldberg and Tarjan [GT88], Hochstein and Weihe [HW07] provided an algorithm for Maximum Flow on "nearly planar graphs" which is parameterized by the crossing number; thus one can see it as an algorithm for Maximum Flow on planar graphs plus k non-planar arcs. This result is one of the main motivators of our work.

Data reduction rules and kernelization results play a central role in the young field of FPT in P. Liers and Pardella [LP11] provided several data reduction rules for Maximum Flow. While exhaustively applying the data reduction rules takes $\mathcal{O}(n^5)$ time in the worst case, the rules can be applied quickly in practice and allow an empirically observed improvement in the running times of the best practical Maximum Flow algorithms. Further, Weihe [Wei97] stated observations which can be translated to simple data reduction rules. While the final running time of the algorithm stated in his work is quasilinear, it is not clear whether the data reduction rules can be applied exhaustively in quasilinear time.

Our contributions. The results of this work can be divided into three parts. First, we provide simple (i.e., quasilinear-time applicable) data reduction rules for Maximum Flow. The data reduction rules are partially based on observations by Weihe [Wei97], and are less complex than those presented by Liers and Pardella [LP11]. We further show that Maximum Flow admits a linear-size kernel with respect to the (undirected) feedback edge number, i.e., the edge deletion distance from the undirected version of the input graph to forests. That is, one can reduce the size of any instance of Maximum Flow in quasilinear time such that the resulting instance has $6k+6$ vertices and $14k+12$ arcs, where k is the feedback edge number (Theorem 3.14). From this we obtain a fixed-parameter algorithm for Maximum Flow with respect to the feedback edge number (Corollary 3.15).

Second, using a concept called *General Problem hardness* by Bentert et al. [Ben+17], we show that there are several parameters such as the vertex deletion distance to bipartite graphs (Corollary 4.6) or the maximum degree (Theorem 4.7), for which presumably there exist no fixed-parameter algorithms. The concept of General Problem hardness usually bases on running time lower bounds for the respective problem. But since no lower bounds are known for Maximum Flow, we show instead that a fixed-parameter algorithm for a certain parameter—as for example the maximum degree—would imply an algorithm that outperforms the best known unparameterized algorithms for Maximum Flow.

Third, inspired by and based on the work of Hochstein and Weihe [HW07], we provide a systematic approach that allows for easy design of fixed-parameter algorithms with respect to vertex deletion distances of certain graph classes. We present two applications of this approach.

One of the two applications of the systematic approach for fixed-parameter algorithms is as follows: We provide an algorithm for Maximum Flow with respect to the (undirected) feedback vertex number k, i.e., the vertex deletion distance from the undirected version of the input graph to forests, running in $\mathcal{O}(k^4 m)$ time (Corollary 5.25). The algorithm makes heavy use of the data reduction rules introduced in Chapter 3. Comparing the result to the fixed-parameter algorithm by Hochstein and Weihe [HW07] running in $\mathcal{O}(k^3 n \log n)$ time, note that the parameters feedback vertex number and crossing number are incomparable. Further, albeit the feedback vertex number being NP-hard to compute, one can approximate the feedback vertex number within a factor of four in linear time [BY+98]. This is in contrast to the crossing number, for which there is no constant-factor approximation, unless $\mathsf{P} = \mathsf{NP}$ [Cab13]. The result is further strengthened by the fact that to this date, for the smaller parameter treewidth, the best known fixed-parameter algorithm for Maximum Flow requires time exponential in the parameter size [Hag+98].

Table 1.2 gives an overview of fixed-parameter algorithms for Maximum Flow as well as a list of parameters that do not help to solve Maximum Flow more efficiently. We present our results in a condensed form in Figure 1.2, which is an excerpt of the graph parameter hierarchy [SW16].

Structure of the work. In Chapter 2 we provide formal definitions on graph theory, parameterized problems and data reduction rules, and flows and capacities. In Chapter 3 we present four efficient data reduction rules and a kernel for Maximum Flow with respect to the undirected feedback edge number. Chapter 4 is devoted to the question which parameters (and other restrictions) probably do not help to solve Maximum Flow more efficiently and shows that Maximum Flow cannot be solved faster with certain parameters than without. In Chapter 5, after presenting the Push-Relabel method of Goldberg and Tarjan [GT88] in detail, we introduce our systematic approach for finding fixed-parameter algorithms for Maximum Flow and present two applications of our approach. Finally, in Chapter 6 we recapitulate the contents of this thesis and discuss further research opportunities.

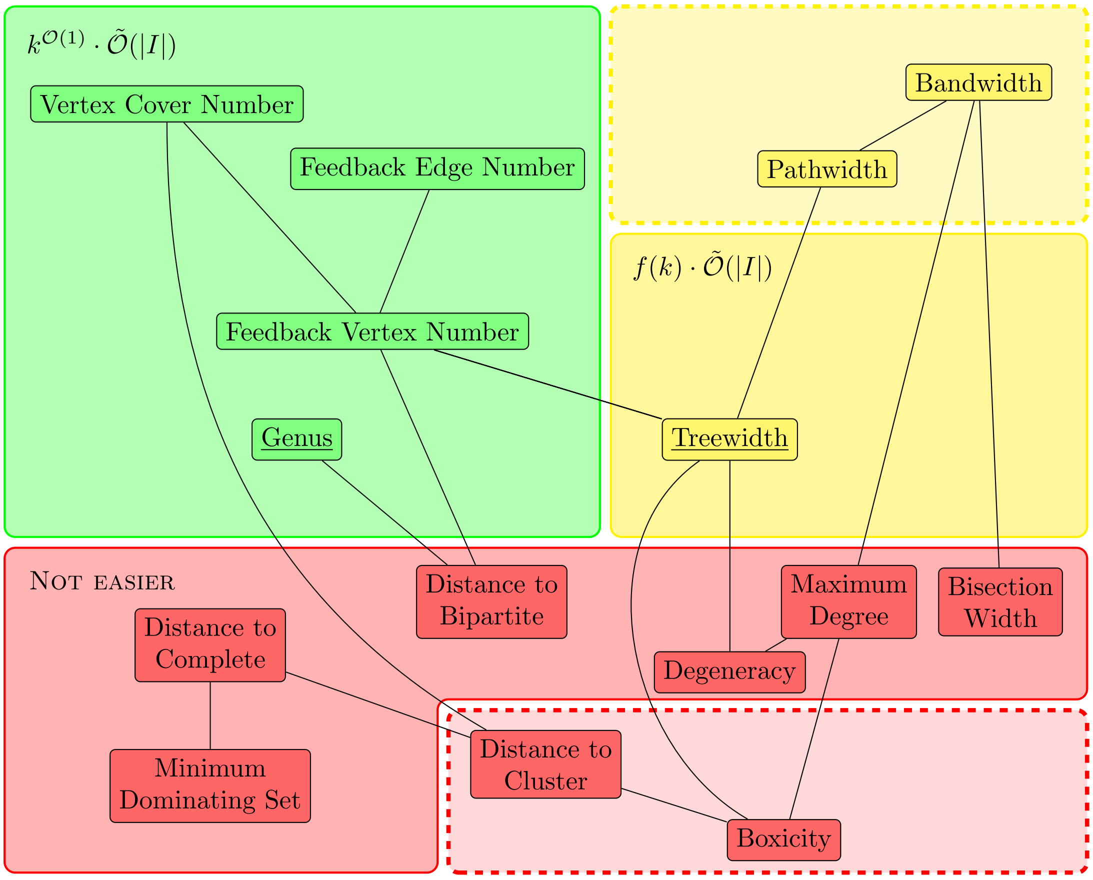


Figure 1.2: An overview over the parameterized complexity of MAXIMUM FLOW. There is an edge between two parameters if the lower parameter can be upper-bounded by a function in the upper parameter. If a parameter k is highlighted green, then there exists an $k^{\mathcal{O}(1)} \cdot \tilde{\mathcal{O}}(|I|)$-time algorithm. If k is highlighted yellow, then there exists an $f(k) \cdot \tilde{\mathcal{O}}(|I|)$-time algorithm, where f is a computable function. If k is highlighted red, then MAXIMUM FLOW does not become easier with respect to k. If a parameter is highlighted in any of the light colors (inside the dashed boxes above the yellow box or the below red box), then a result for the parameter can be deduced from a parameter in the corresponding darker color. The corresponding references are listed in Table 1.2. Apart from the results for the parameters treewidth [Hag+98] and genus [CEN12], all results are either shown in this work or can be deduced from results for the other parameters.

Chapter 2

Preliminaries

In this chapter we provide some basic notation on graphs, parameterized problems and flows. Note that we predominantly use standard notation throughout this work. The lists are intended to provide easy look-up of specific notation. As a convention, by $\mathbb{N}$ we denote the set of natural numbers without zero, and by $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$ the set of natural numbers with zero. Further, we assume $\log n := \log_2 n$, that is, the base of the logarithm is two, if unspecified. Let C be a set. We denote by $C = A \uplus B$ the partition of C into two sets A, B such that $A \cup B = C$ and $A \cap B = \emptyset$.

2.1 Graph Theory

This section provides the notation that we use for graphs throughout this work. Note that in general we work with simple (undirected and directed) graphs, that is, the graphs do not contain any self loops or parallel arcs or parallel edges. Most of the notation is based on the book of Diestel [Die17]. For a set S we denote by $\binom{S}{2}$ the set of all size-two subsets of S, and by $S^2 = S \times S$ the set of all pairs (a, b) with $a, b \in S$.

Undirected graphs. An undirected graph is a pair $G = (V(G), E(G))$. In this context, we denote by

$V(G)$	the *vertex set* of G;
$E(G)$	the *edge set* of G with $E(G) \subseteq \binom{V(G)}{2}$; for an edge $e = \{u, v\} \in E(G)$ the two vertices u and v are called *endpoints* of e;
n_G	the number $\lvert V(G)\rvert$ of *vertices*;
m_G	the number $\lvert E(G)\rvert$ of *edges*;
$N_G(v)$	the (open) *neighborhood* of v, formally, $N_G(v) := \{u \in V \mid \{u, v\} \in E(G)\}$;
$N_G(V')$	the *(open) neighborhood* of V', formally, $N_G(V') := \big(\bigcup_{u \in V'} N_G(u)\big) \setminus V'$ for $V' \subseteq V(G)$;
$\deg_G(v)$	the *degree* of v, formally, $\deg_G(v) := \lvert N_G(v)\rvert$;

$\Delta(G)$	the *maximum degree* of G, formally, $\Delta(G) := \max_{v \in V}\{\deg_G(v)\}$;
$G[V']$	the *induced subgraph* of G on $V' \subseteq V$, formally, $G[V'] := (V', E(G) \cap \binom{V'}{2})$);
$G - E'$	the graph obtained from G by deleting the edges $E' \subseteq E(G)$, formally, $G - E' := (V(G), E(G) \setminus E')$;
$G - V'$	the graph obtained from G by deleting the vertices $V' \subseteq V(G)$, formally, $G - V' := G[V(G) \setminus V']$.

If the graph G is clear from the context, then we will omit the subscript G. As a convention throughout the work, if we call a graph by the letter G (possibly with sub- or superscript), then the graph is undirected. We say that two vertices $v, w \in V(G)$ are *adjacent* if $\{v, w\} \in E(G)$. We say that a vertex v is *incident* to an edge $e \in E(G)$, and that e is incident to v, if $v \in e$.

Directed graphs. A directed graph is a pair $D = (V(D), A(D))$. We denote by

$V(D)$	the *vertex set* of D;
$A(D)$	the *arc set* of D with $A(D) \subseteq V(D)^2$; for an arc $a = (u, v) \in A(D)$ the vertex u is called *source* and v is called *destination* of a;
n_D	the number $\lvert V(D) \rvert$ of vertices;
m_D	the number $\lvert V(D) \rvert$ of arcs;
$G(D)$	the *underlying undirected graph*, formally, $G(D) := (V(D), \{\{u, v\} \mid (u, v) \in A(D) \vee (v, u) \in A(D)\})$;
$\overleftarrow{D}$	the *reverse graph* of D, formally, $\overleftarrow{D} := (V(D), \{(v, u) \mid (u, v) \in A(D)\}$;
$N^-_D(v)$	the *ingoing neighborhood* of v, formally, $N^-_D(v) := \{u \in V(D) \mid (u, v) \in A(D)\}$;
$N^+_D(v)$	the *outgoing neighborhood* of v, formally, $N^+_D(v) := \{w \in V(D) \mid (v, w) \in A(D)\}$;
$N_D(v)$	the *(combined) neighborhood* of v, formally, $N_D(v) := N^+_D(v) \cup N^-_D(v)$;
$\deg^-_D(v)$	the *indegree* of v, formally, $\deg^-_D(v) := \lvert N^-_D(v) \rvert$;
$\deg^+_D(v)$	the *outdegree* of v, formally, $\deg^+_D(v) := \lvert N^+_D(v) \rvert$;
$\deg_D(v)$	the *(combined) degree* of v, formally, $\deg_D(v) := \lvert N_D(v) \rvert$;
$\Delta^-(D)$	the *maximum indegree* of v, formally, $\Delta^-(D) := \max\{\deg^-_D(v) \mid v \in V(D)\}$;
$\Delta^+(D)$	the *maximum outdegree* of v, formally, $\Delta^+(D) := \max\{\deg^+_D(v) \mid v \in V(D)\}$;
$\Delta(D)$	the *maximum (combined) degree* of v, formally, $\Delta(D) := \max\{\deg_D(v) \mid v \in V(D)\}$;

$D[V']$	the *induced subgraph* of D on $V' \subseteq V(D)$, formally $D[V'] \coloneqq (V', A(D) \cap V(D)^2)$;
$D - A'$	the graph obtained from D by deleting the arcs $A' \subseteq A(D)$, formally, $D - A' \coloneqq (V(D), A(D) \setminus A')$;
$D - V'$	the graph obtained from D by deleting the vertices $V' \subseteq V(D)$, formally, $D - V' \coloneqq D[V(D) \setminus V']$.

As with undirected graphs, the subscript D is dropped if the graph meant is clear from the context. As a convention throughout the work, if we call a graph by the letter D (possibly with sub- or superscript), then the graph is directed. Further we say that two vertices $v, w \in V(D)$ are *adjacent* if $(v, w) \in A(D)$ or $(w, v) \in A(D)$. We say that a vertex $v \in V(D)$ is incident to an arc $a \in A(D)$, and that a is incident to v, if v is either the source or the destination of a. For every arc $a = (u, v) \in A(D)$ we denote by $\overleftarrow{a}$ the arc (v, u).

Graph properties. Let G be an undirected graph. A graph G' is a *subgraph* of G, written $G' \subseteq G$, if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G) \cap \binom{V(G')}{2}$, and an *induced subgraph* of G, if $V(G') \subseteq V(G)$ and $E(G') = E(G) \cap \binom{V(G')}{2}$. Let D be a directed graph. A (directed) graph D' is a *subgraph* of D, written $D' \subseteq D$, if $V(D') \subseteq V(G)$ and $A(D') \subseteq A(D) \cap (V(D')^2)$, and an *induced subgraph* of D, if $V(D') \subseteq V(D)$ and $A(D') = A(D) \cap V(D')^2$.

A *path* is an undirected graph P with vertex set $V(P) = \{v_1, \dots v_n\}$, $n \geq 2$, and edge set $E(P) = \{\{v_i, v_{i+1}\} \mid 1 \leq i < n\}$. We denote the path by $v_1 \dots v_n$, and the vertices v_1 and v_n are called *endpoints*, and all other vertices are called *inner vertices.* We say that the path is a *v_1–v_n-path*, and the *length* of the path is $m = n - 1$, and we say that the path visits the vertices $v_1, \dots, v_n$ and the edges $\{v_1, v_2\}, \dots, \{v_{n-1}, v_n\}$.

Let G be an undirected graph. We say that $v \in V(G)$ *reaches* $w \in V(G)$, or v and w are connected to each other, if G contains a v–w-path. The distance $d_G(v, w)$ is the length of the shortest v–w-path. An undirected graph is *connected* if every pair of vertices $v, w \in V(G)$ is connected. A *connected component* of G is a subgraph induced by a maximal set of pairwise connected vertices. A vertex $v \in V(G)$ is a *cut vertex* if $G - \{v\}$ contains more connected components than G.

A *(directed) path* is a directed graph P with vertex set $V(P) = \{v_1, \dots v_n\}$ and arc set $A(P) = \{(v_i, v_{i+1}) \mid 1 \leq i < n\}$. We denote the path by $v_1 \dots v_n$ and the vertices v_1 and v_n are called *endpoints*, and all other vertices are called *inner vertices.* We say that the path is a *v_1–v_n-path*, and the *length* of the path is $m = n - 1$, and we say that the path visits the vertices $v_1, \dots, v_n$ and the arcs $(v_1, v_2), \dots (v_{n-1}, v_n)$.

Let D be a directed graph. A vertex $v \in V(D)$ *reaches* a vertex $w \in V(D)$, and w is reached by v, if there is a directed v–w-path in D. The distance $d_D(v, w)$ is the length of the shortest v–w-path. Two vertices $v, w \in V(D)$ are connected if they are connected in the underlying undirected graph $G(D)$. This is also called *weakly connected.* A directed graph D is *(weakly) connected* if every pair of vertices in $V(D)$ is connected. A *(weakly) connected component* of a directed graph is a subgraph induced by a maximal set of pairwise connected vertices. A vertex $v \in V(D)$ is called a *source vertex*, or source, if $\deg^-(v) = 0$, and it is called a *sink vertex*, or sink, if $\deg^+(v) = 0$. A vertex $v \in V(D)$ is a *cut vertex* if it is a cut vertex in the underlying undirected graph $G(D)$.

Special graphs and graph parameters. An undirected graph G with $V(G) = \{v_1, \ldots, v_n\}$ is a

cycle	if G is connected and each vertex has degree two. We denote the cycle by the sequence of its vertices $v_1 v_2 \ldots v_n v_1$.
independent set	if G contains no edge;
complete graph	if $E(G) = \binom{V(G)}{2}$. We denote a complete graph on n vertices by K_n.
bipartite graph	if $V(G) = V_1 \uplus V_2$ and $G[V_1]$ and $G[V_2]$ are independent sets;
complete bipartite graph	if $V(G) = V_1 \uplus V_2$ and $E(G) = \{\{v, w\} \mid (v, w) \in V_1 \times V_2\}$. We denote a complete bipartite graph by $K_{p,q}$, where $p := \lvert V_1 \rvert$ and $q := \lvert V_2 \rvert$.
planar graph	if G can be embedded into the plane (drawn with points for vertices and curves for edges) without crossing edges;
forest	if G contains no cycle;
tree	if G is a forest and is connected;
cluster graph	if every connected component is a complete graph;
interval graph	if there are real intervals $\{I_v \mid v \in V(G)\}$ such that $\{v_i, v_j\} \in E(G)$ if and only if $I_{v_i} \cap I_{v_j} \neq \emptyset$.

A *$p \times q$-grid* is an undirected graph G with vertex set $V(G) := \{v_{i,j} \mid 1 \leq i \leq p,\ 1 \leq j \leq q\}$ and edge set $E(G) := \{\{v_{i,j}, v_{i',j'}\} \mid |i - i'| + |j - j'| = 1\}$. A *grid graph* is a subgraph of a grid.

Let T be a tree. Note that in a tree, every pair of vertices is connected by a unique path. We call $v \in V(T)$ a *leaf* if $\deg(v) = 1$. A *rooted tree* is a tree T with a designated *root* $r \in V(T)$. We say that T is *rooted in* r. Let $v, w \in V(T)$ be two vertices such that the unique r–w-path visits v. Then we call w *an ancestor of* v and v *a descendant of* w. If additionally v and w are adjacent, then we call w *a child of* v and v *the parent of* w.

A directed graph D with vertex set $V(D) = \{v_1, \ldots, v_n\}$ is a *(directed) cycle*, if $A(D) = \{(v_i, v_{i+1}) \mid 1 \leq i < n\} \cup \{(v_n, v_1)\}$, and a *(directed) complete graph*, if $A(D) = \{(v, w), (w, v) \mid \{v, w\} \in \binom{V(G)}{2}\}$. A *directed $k \times \ell$-grid* is a directed graph whose underlying undirected graph is a $k \times \ell$-grid. A *directed grid graph* is a directed graph whose underlying undirected graph is a grid graph.

A class $\mathcal{C}$ of (directed or undirected) graphs is a set of graphs. As an example, let $\mathcal{C}$ be the class of complete graphs. Then for every $n \in \mathbb{N}$, $K_n \in \mathcal{C}$. We call a class $\mathcal{C}$ of graphs *hereditary* if it holds that for every graph $G \in \mathcal{C}$, every induced subgraph G' of G is also contained in $\mathcal{C}$. The class of complete graphs, the class of independent sets and the class of forests are examples for hereditary graph classes. The class of cycles is not hereditary.

Let G be a graph. We say that a vertex subset $V' \subseteq V(G)$ is a

vertex cover	if $G - V'$ is an independent set;
dominating set	if $N(V') \cup V' = V(G)$;
feedback vertex set	if $G - V'$ is a forest;
vertex deletion set to $\mathcal{C}$	if $\mathcal{C}$ is a class of graphs and $G - V' \in \mathcal{C}$.

An edge subset $E' \subseteq E(G)$ is a

feedback edge set	if $G - E'$ is a forest;
bisection	if $G - E'$ has exactly two connected components C_1 and C_2 such that the number of vertices in C_1 and C_2 differ by at most one.

We define the following graph parameters for an undirected graph G:

The *degeneracy* of G	is the smallest number d such that for every subgraph $G' \subseteq G$, it holds true that $\Delta(G') \leq d$;
the *vertex cover number*	is the size of a minimum vertex cover of G;
the *domination number*	is the size of a minimum dominating set of G;
the *feedback vertex number*	is the size of a minimum feedback vertex set of G;
the *vertex deletion distance to $\mathcal{C}$*	is the size of the minimum vertex deletion set to $\mathcal{C}$, where $\mathcal{C}$ is a class of graphs;
the *feedback edge number*	is the size of a minimum feedback edge set of G;
the *bisection number*	is the size of a minimum bisection of G.

Data structures for graphs. If given an undirected (directed) graph as input, we assume to be given the number of vertices and edges (arcs) and, for each vertex, its adjacency list (list of ingoing and outgoing neighbors). This allows for constant-time access to any vertex and to the list of all edges (arcs) incident to a specific vertex, but does not allow for quick access to a selected edge (arc). Thus, when needed, we manage the edges (arcs) of the input graph in an AVL-Tree. The operations of looking up, inserting and deleting an edge (arc) all take $\mathcal{O}(\log m) \subseteq \mathcal{O}(\log n^2) \subseteq \mathcal{O}(\log n)$ time. Creating the AVL-Tree can be done by m consecutive inserts in $\mathcal{O}(m \log n)$ time.

2.2 Parameterized Problems and Data Reduction Rules

Let Σ be a finite alphabet, and let Σ^* be the set of *all words* over Σ. A *decision problem* P is a subset of Σ^*. An instance $I \in \Sigma^*$ is a yes-instance for problem P if $I \in P$, and a no-instance otherwise.

The notation $\tilde{\mathcal{O}}(f) := \mathcal{O}(f \log^c f)$ for a function f and a constant c is used to hide polylogarithmic factors. We call a function $g : \mathbb{N} \to \mathbb{N}$ *quasilinear*, if $g(n) \subseteq \mathcal{O}(n \log^{\mathcal{O}(1)} n)$ for $n \in \mathbb{N}$. Let f and p be computable functions. We call an algorithm running on an instance of size $|I|$ with a running time of the kind $\mathcal{O}(f(k) \cdot p(|I|))$ a *fixed-parameter algorithm* if p is polynomial, a *linear-time fixed-parameter algorithm* if p is linear, and a *quasilinear-time fixed-parameter algorithm* if p is quasilinear. We call a fixed-parameter algorithm *fully polynomial* if f is polynomial.

A *parameterized problem* $P \subseteq \Sigma^* \times \mathbb{N}$ is a set of instances (I, k) where $I \in \Sigma^*$, and $k \in \mathbb{N}$ is the parameter. We call the set of corresponding instances I without the parameter the *unparameterized decision problem* associated to P. We say that two instances (I, k) and (I', k') of a parameterized problem P are *equivalent* if (I, k) is a yes-instance for P if and only if (I', k') is a yes-instance for P.

A *kernelization* is an algorithm that, given an instance (I, k) of P, computes in polynomial time an equivalent instance (I', k') of P (the *kernel*) such that $|I'| + k' \leq f(k)$ for some computable function $f \colon \mathbb{N} \to \mathbb{N}$. We say that f measures the *size* of the kernel.

A *data reduction rule* is an algorithm that takes as an input an instance I of a problem P, and outputs another instance I' of P. We say that a data reduction rule R is *correct* if the new instance I' that results from applying R to I is equivalent to I'. We say that R is *exhaustively applied* to an instance I if further application of this rule has no effect on the instance.

2.3 Maximum Flow

In this section, we define the problems Maximum Flow and k-Flow, and any notation related to the problems that are used throughout this work. Lastly, we will present the Ford-Fulkerson method [FF56] for solving Maximum Flow.

Let D be a directed graph. A capacity function is a total function $c \colon A(D) \to \mathbb{N}_0$. We denote by $C := \max\{c(a) \mid a \in A(D)\}$ the *maximum capacity* of c. For the sake of convenience, if $(u, v) \notin A(D)$, then we assume $c((u, v)) = 0$.

We start by defining what a flow is.

Definition 2.1 (Flow)**.** Let D be a directed graph, let $s \neq t \in V(D)$ be two terminals, and let $c \colon A(D) \to \mathbb{N}_0$ be a capacity function. We call a function $f \colon A(D) \to \mathbb{N}_0$ an *s–t-flow* on D, if

$$\forall a \in A(D) \colon 0 \leq f(a) \leq c(a) \qquad \text{(capacity constraint), and}$$

$$\forall v \in V(D) \setminus \{s, t\} \colon \sum_{u \in N^-(v)} f((u, v)) - \sum_{w \in N^+(v)} f((v, w)) = 0 \qquad \text{(conservation constraint).}$$

The *value* of flow f is defined as $\operatorname{val}(f) := \sum_{(s,v) \in A(D)} f((s, v))$.

A *maximum s–t-flow* is an s–t-flow of maximum value. For better readability, we denote by $c(u, v)$ the capacity, and by $f(u, v)$ the flow on the arc $(u, v) \in A(D)$.

Let D be a graph and let D' be a subgraph of D. Let $f\colon A(D) \to \mathbb{N}_0$ be a flow on D, and let $f'\colon A(D') \to \mathbb{N}_0$ be a flow on D'. Then, we define $f + f'$ to be the flow $f''\colon A(D) \to \mathbb{N}_0$, where

$$\forall (u, v) \in A(D')\colon f''(u, v) := f(u, v) + f'(u, v), \text{ and}$$
$$\forall (u, v) \in A(D) \setminus A(D')\colon f''(u, v) := f(u, v).$$

Throughout this work, we are going to work with so-called *residual graphs*. Given a graph D and a flow f, the residual graph shows along which arc flow can be sent, and the residual capacity of an arc shows how much flow can be sent along the arc.

Definition 2.2 (Residual graph and capacities)**.** Let D be a graph, let $c\colon A(D) \to \mathbb{N}_0$ be a capacity function, and let $f\colon A(D) \to \mathbb{N}_0$ be a flow. The directed graph D_f with vertex set $V(D_f) = V(D)$ and arc set

$$A(D) = \{a \mid a \in A(D) \wedge 0 \le f(a) < c(a)\} \cup \{\overleftarrow{a} \mid a \in A(D) \wedge 0 < f(a) \le c(a)\}$$

is called the *residual graph of D with respect to f*. The capacity function $c_f\colon A(D_f) \to \mathbb{N}_0$ where

$$\forall a \in A(D_f)\colon c_f(a) = c(a) - f(a) + c(\overleftarrow{a})$$

is called the *residual capacity function with respect to f*.

We are now ready to state the main problem of this work:

Maximum s–t-Flow (Maximum Flow)
Input: A directed graph D, two vertices $s \neq t \in V(D)$, and a capacity function $c\colon A(D) \to \mathbb{N}_0$.
Task: Compute a maximum s–t-flow.

An instance I of Maximum Flow is a four-tuple (D, s, t, c) consisting of a directed graph D, two terminals $s \neq t$, and a capacity function c.

The above problem is an *optimization problem*, since the task is to find a flow of maximum value. In some cases, we are only interested whether there exists a flow of some value k. For this, we define the corresponding decision problem:

k-Flow
Input: A directed graph D, two vertices $s \neq t \in V(D)$, arc capacities $c\colon A(D) \to \mathbb{N}_0$ and an integer $k \in \mathbb{N}$.
Question: Is there an s–t-flow of value at least k in D?

An instance I of k-Flow is a five-tuple (D, s, t, c, k) where the first four entries are the same as for Maximum Flow, and the fifth entry is the integer k.

Let $I = (D, s, t, c)$ be an instance of Maximum Flow. Then we call the corresponding maximum s–t-flow f the *solution* for I. Let I and I' be two instances of Maximum Flow, and let f and f' be the corresponding solutions for I and I' respectively. We say that I and I' are *equivalent* if $\mathrm{val}(f) = \mathrm{val}(f')$.

Since k-Flow is a decision problem, the definition of equivalence follows from Section 2.2: Two instances I, I' of k-Flow are equivalent if I is a yes-instance if and only if I' is a yes-instance.

Algorithm 2.1: The Ford-Fulkerson method [FF56].

```
1 Initialize flow $f$ such that for all $(u, v) \in A(D)$, $f(u, v) = 0$.
2 while there is an $s$–$t$-path $P$ in $D_f$ do
3     Let $\delta \coloneqq \min\{c_f(a) \mid a \in A(P)\}$ be the smallest capacity in $P$.
4     Send flow of $\delta$ along the arcs of $P$, that is, $\forall a \in A(P)\colon f(a) \coloneqq f(a) + \delta$.
5 end
```

The Ford-Fulkerson method. In some of the proofs in the following chapters we make use of one of the easiest combinatorial algorithms for Maximum Flow—the Ford-Fulkerson method [FF56]. It is described in Algorithm 2.1.

The algorithm works as follows: It sends flow along s–t-paths in D_f, until there are no more s–t-paths in D_f. To prove that f is a maximum s–t-flow on termination, Ford and Fulkerson [FF56] stated the following seminal theorem:

Theorem 2.3. *Let D be a graph, let $s \neq t \in V(D)$ be two terminals and let c be a capacity function on D. Then, an s–t-flow f is maximum if and only if there is no s–t-path in D_f.*

Chapter 3

Preprocessing

Preprocessing simple parts of a problem instance can often lead to highly improved running times. A cardinal example for this are ILP-solvers [AS05; GG11; Sav94], but preprocessing also has its justification in the world of combinatorial algorithms [Bag+12; BBN14; Bru+12]. Another hot topic related to preprocessing is the field of finding problem kernels. Recently, problem kernels have become a field of interest for polynomial-time algorithms. Examples for this are the works of Mertzios, Nichterlein, and Niedermeier [MNN17] for Maximum Matching, [Flu+17] for Graph Hyperbolicity and of Bentert et al. [Ben+17] for Triangle Enumeration.

In 2011, Liers and Pardella [LP11] presented data reduction rules for Maximum Flow. Their rules can be split into two categories: reducing high capacities and contracting arcs of high capacities. While applying the rules exhaustively takes $\mathcal{O}(n^5)$ time in the worst case, they have shown that in practice, applying their rules before executing some of the established algorithms for Maximum Flow yields running time improvements over executing the algorithms without application of their rules.

Inspired by the work on kernels for polynomial-time solvable problems and by the effectiveness of the preprocessing by Liers and Pardella [LP11], we consider four data reduction rules for Maximum Flow in this chapter. Three of the four rules are based on so-called assumptions made by Weihe [Wei97]. The rules are simpler in nature than the preprocessing rules by Liers and Pardella; each of the rules can be applied in quasilinear time respectively. We show that the exhaustive application of three of the four rules yield a linear kernel in the feedback edge number of the underlying undirected graph, computable in quasilinear time. Ideally, we would be able to apply all four reductions rules together in quasilinear time, but this proves to be very challenging. Nevertheless, we were able to provide an $\mathcal{O}(nm)$-time algorithm for applying the four rules exhaustively together.

Chapter outline. In Section 3.1 we present four data reduction rules and prove their correctness. Next, in Section 3.2 we show that an exhaustive application of the first three reduction rules can be done in quasilinear time and yields a problem kernel linear in the size of the feedback edge number of the underlying undirected graph. Finally, in Section 3.3 we depict an $\mathcal{O}(nm)$-time algorithm for applying all four reduction rules exhaustively and discuss the possibility and challenges of doing so in quasilinear time.

3.1 Reduction Rules for Maximum Flow

In this section we will state the four data reduction rules and prove that they are correct. The first two rules are based on an assumption stated by Weihe [Wei97]. The assumption and the two rules follow from the obvious fact that a maximum s–t-flow only sends flow along arcs that are visited by some s–t-path.

Observation 3.1. *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow*. Then there exists a maximum s–t-flow f such that for every arc $a \in A(D)$ that is not visited by any s–t-path, $f(a) = 0$, and for every vertex $v \in V(D)$ that is not visited by any s–t-path, the sum of the incoming flows and the sum of the outgoing flows is zero.*

Proof. Let f be the flow obtained by executing the Ford-Fulkerson method [FF56] on I (see Section 2.3 on page 22). Since the Ford-Fulkerson method only sends flow along s–t-paths, we have $f(a) = 0$ for any arc $a \in A(D)$ that is not visited by any s–t-path. If a vertex $v \in V(D)$ is not visited by any s–t-path, then its incident arcs are not visited by any s–t-path either. Hence, no flow is sent through v, that is,

$$\sum_{u \in N^-(v)} f(u, v) = \sum_{w \in N^+(v)} f(v, w) = 0. \qquad \square$$

So ideally, to reduce the instance size, one removes every vertex and arc that is not visited by any s–t-path. Unfortunately, identifying all arcs or vertices that are visited by some s–t-path in D is NP-hard. It is even NP-complete to decide whether there exists an s–t-path that visits some given arc a or vertex v as shown in the following observation.

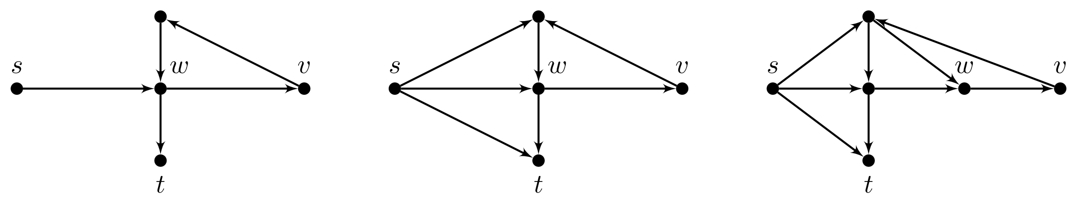
Observation 3.2. *Let D be a directed graph and let $s, t \in V(D)$. Then, for a given arc $a \in A(D)$ (or a vertex $v \in V(D)$) it is NP-complete to decide whether there exists an s–t-path that visits a (or v).*

Proof sketch. Clearly the stated problem is in NP. Fortune, Hopcroft, and Wyllie [FHW80] have shown that, given four vertices $s_1, s_2, t_1, t_2 \in V(D)$, to decide whether there exists an s_1–t_1-path P and an s_2–t_2-path Q such that P and Q are vertex disjoint is NP-complete. Adding arc $a = (t_1, s_2)$ to D yields an equivalent instance to the problem asking whether arc a is visited by an s–t-path. Adding vertex v with arcs (t_1, v) and (v, s_2) yields an equivalent instance to the problem asking whether vertex v is visited by an s–t-path. □

So, the goal of Weihe's assumption, and thus of our first two data reduction rules, is to remove as many vertices and arcs as possible, along which no flow can be sent. We first make a simple observation that allows us to assume that s is a source and t is a sink, and remove any arc (v, s) and (t, v) otherwise.

Observation 3.3. *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow*. Then there exists a maximum s–t-flow f such that for every $v \in N^-(s)$ and for every $w \in N^+(t)$, $f(v, s) = f(t, w) = 0$.*

We are now ready to state a data reduction rule based on one of the two assumptions made by Weihe [Wei97].



(a) There is an s–v-path and a v–t-path, but there is no s–t-path visiting v.

(b) This graph does not contain any cut vertices and no s–t-path visiting v.

(c) There is an s–v-path and a w–t-path, both via (w, v), but no s–t-path via (w, v).

Figure 3.1: Three examples to show that the BFS-Rule (a), the Cut-Rule (b) and Weihe's Rule (c) do not delete all vertices that are not visited by an s–t-path.

Reduction Rule 3.1 (Weihe's Rule)**.** *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow *and let $(v, w) \in A(D)$ be an arc such that there exists no s–w-path visiting (v, w) or no v–t-path visiting (v, w). Then delete (v, w).*

Unfortunately, Weihe [Wei97] has not shown how to apply this rule exhaustively in quasilinear time, and in the scope of this work we were not able to find a way to do so either. The main challenge we see in applying Weihe's Rule in (quasi-)linear time is that after applying the rule on one arc, the rule may become applicable on other arcs as well. We do not know at this point how to denominate those arcs on which the rule then becomes applicable. Instead, we introduce two data reduction rules that are based on, but are not as powerful as Weihe's Rule. In Section 3.2 we will show that we can apply the next two reduction rules exhaustively in linear time. The first of the two rules can be considered as the vertex version of Weihe's Rule.

Reduction Rule 3.2 (BFS-Rule)**.** *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow *and let $v \in V(D)$ be a vertex such that there exists no s–v-path or no v–t-path. Then delete v.*

Recall that in a path, vertex may be visited twice. Thus the BFS-Rule does not necessarily remove all vertices that are not visited by an s–t-path, since for some vertex v, the s–v-paths and v–t-paths may share vertices. Figure 3.1a shows a simple example for a vertex v that is not removed by the rule, but is not visited by any s–t-path. Note though that in Figure 3.1a, all s–v-paths and all v–t-paths visit the cut vertex w. We can generalize on this observation as follows: Given a cut vertex $w \in V(D)$, consider the (weakly) connected components of $D - \{w\}$. Then for every component that does not contain s or t it holds that none of its vertices can be visited by any s–t-path. From this we derive our second reduction rule.

Reduction Rule 3.3 (Cut-Rule)**.** *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow *where the BFS-Rule is not applicable, and let $w \in V(D)$ be a cut vertex. Let $v \in V(D) \setminus \{s, t\}$ be a vertex such that all s–v-paths and all v–t-paths visit w. Then delete v.*

Note that after applying the BFS-Rule and the Cut-Rule exhaustively, our graph may still contain arcs on which Weihe's Rule still is applicable. An example can be seen in Figure 3.1b. But Weihe's Rule still cannot delete all arcs that are not visited by an s–t-path, as shown in Figure 3.1c.

The correctness of the two reduction rules presented so far is easy to see; it follows from Observation 3.1.

Observation 3.4. *The BFS-Rule and the Cut-Rule are correct.*

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow. The correctness of the BFS-Rule follows directly from Observation 3.1.

For showing that the Cut-Rule is correct, let $v \in V(D) \setminus \{s, t\}$ be a vertex such that each s–v-path and each v–t-path visits the same cut vertex w. Let $P_s = sx_1x_2 \dots x_kv$ be any such s–v-path and let $P_t = vy_1y_2 \dots y_\ell t$ be any such v–t-path. Then the concatenation of the two paths $sx_1 \dots x_kvy_1 \dots y_\ell t$ visits w twice, and thus is no path. Hence, there is no s–t-path visiting v and by Observation 3.1 v can be removed. It follows that the Cut-Rule is correct. □

At this point we can also observe that s cannot be a cut vertex if the BFS-Rule was applied exhaustively.

Observation 3.5. *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow *on which the BFS-Rule is not applicable. Then s is not a cut vertex.*

Proof. Since s is a source vertex, any neighbor v of s that does not reach t is removed by the BFS-Rule. Thus, after removing s from D, every vertex still reaches t. □

Next, we present two reduction rules that do not only exploit the structure of the input graph, but also take the capacities of arcs into consideration. The first of the two rules is based on an assumption by Weihe [Wei97] and deletes vertices of degree two:

Reduction Rule 3.4 (Deg-2-Rule). *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow *and let $v \in V(D) \setminus \{s, t\}$ be a vertex of combined degree two, and let u and w be the neighbors of v. If there exists a u–w-path, then ensure that $(u, w) \in A(D)$ (if necessary, add (u, w) with capacity zero), and increase $c(u, w)$ by $\min\{c(u, v), c(v, w)\}$. Proceed analogously if there exists a w–u-path. Then remove v from D.*

The correctness of the Deg-2-Rule follows from the flow conservation constraint.

Observation 3.6. *The Deg-2-Rule is correct.*

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow. Let $v \in V(D)$ be a vertex with two neighbors u and w. Note that, due to the conservation constraint, the highest flow that can be sent along (u, v) and (v, w) is $\min\{c(u, v), c(v, w)\}$ and the highest flow that can be sent along (w, v) and (v, u) is $\min\{c(w, v), c(v, u)\}$. Since any flow passing through v must also pass through u and w, the Deg-2-Rule can be applied without changing the value of the maximum s–t-flow of the network and thus is correct. □

The last rule that we present exploits that at least one arc of an s–t-path of length two can be saturated. Hence, we can send flow along this path and remove at least one of its arcs. Note that the implementation of the work of Boykov and Kolmogorov [BK04] hints that they apply a similar rule.

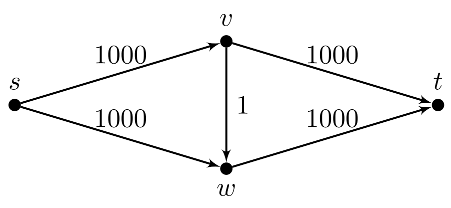


Figure 3.2: An example why the SVT-Rule does not generalize to longer paths.

We formulate the rule for the decision variant k-Flow. For applying the rule on the optimization variant, Maximum Flow, we need to remember how much flow has been sent along the path, and then take care of the altered arcs in a postprocessing step (for further details we refer to Theorem 5.24 on page 66).

Reduction Rule 3.5 (SVT-Rule)**.** *Let $I = (D, s, t, c, k)$ be an instance of k-Flow and let $v \in V(D) \setminus \{s, t\}$ such that there is a path $P = svt$ in D. Then decrease k by $\min\{c(s, v), c(v, t)\}$, and decrease $c(s, v)$ and $c(v, t)$ by the same value. If arc $a \in A(P)$ has capacity zero, delete it.*

For showing that the rule above is correct, we prove that there exists an s–t-flow f such that in any s–t-path of length two the arc with the lower capacity is saturated.

Lemma 3.7. *The SVT-Rule is correct.*

Proof. Let $I = (D, s, t, c, k)$ be an instance of k-Flow and let $v \in V(D)$ be such that there is a path $P = svt$ in D. We show that from applying the SVT-Rule on P we obtain an equivalent instance $I' = (D', s, t, c', k')$, where $k' = k - \min\{c(s, v), c(v, t)\}$. We execute the Ford-Fulkerson method [FF56] (see Section 2.3 on page 22) on I and assume without loss of generality that P is the first s–t-path on which we augment. Then we send flow of value $\min\{c(s, v), c(v, t)\}$ along P. Note that afterwards, the method will not send flow along the reverse arcs of P since they cannot be visited by any s–t-path. Hence, when the algorithm terminates, the s–t-flow f computed on I sends flow of value at least $\min\{c(s, v), c(v, t)\}$ along the arcs of P.

Assume that $\mathrm{val}(f) \geq k$, thus I is a yes-instance. Let $A(P) = \{a, b\}$, and without loss of generality let $c(a) \leq c(b)$. When applied on P, the SVT-Rule decreases $c(a)$ and $c(b)$ by $\min\{c(a), c(b)\} = c(a)$. We obtain a valid s–t-flow f' on I' from f by setting $f'(a) = f(a) - c(a)$ and $f'(b) = f(b) - c(a)$. Then,

$$\mathrm{val}(f') = \sum_{v \in N_D^+(s)} f'(s, v) = \Big(\sum_{v \in N_D^+(s)} f(s, v) \Big) - c(a) = \mathrm{val}(f) - \min\{c(s, v), c(v, t)\} \geq k',$$

implying that I' is a yes-instance. Given that I' is a yes-instance, we can send additional flow of value $\min\{c(s, v), c(v, t)\}$ in I, implying that I is a yes-instance as well. □

Note that the SVT-Rule does not generalize to longer paths, since it is not necessarily optimal to saturate the arc with the lowest capacity. An example for this is shown in Figure 3.2. After choosing the path $svwt$ of length three and removing arc (v, w), one cannot reach the maximum flow value of 2000. Note that one can add structure to the graph such that the path $svwt$ is the shortest path.

3.2 A Kernel for Maximum Flow

In this section we will show that the BFS-Rule, the Cut-Rule and the Deg-2-Rule yield a kernel with respect to the feedback edge number of the underlying undirected graph. Recall that a feedback edge set of an undirected graph G is a subset $E' \subseteq E(G)$ of edges such that $G - E'$ is a forest. The feedback edge number of G is the smallest number k such that there is a feedback edge set of size k in G. A feedback edge set F of an undirected graph G can be computed in $\mathcal{O}(n + m)$ time by computing a spanning tree $T \subseteq G$ and setting $F = E(G) \setminus E(T)$.

We first show that the BFS-Rule, the Cut-Rule and the Deg-2-Rule can be exhaustively applied in quasilinear time. Afterwards we will show how to upper-bound the size of a correspondingly reduced undirected graph in the feedback edge number. Finally we will show that the size of an instance of Maximum Flow on which the three aforementioned reduction rules were exhaustively applied is upper-bounded by a linear function in the feedback edge number of the input graph. While the parameter feedback edge number tends to be rather large, this is to the best of our knowledge the first problem kernel result for Maximum Flow.

As shown in the following, the BFS-Rule can be applied in linear time.

Lemma 3.8. *The BFS-Rule can be applied exhaustively in $\mathcal{O}(n + m)$ time.*

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow. We run a breadth-first search in D starting in s, and a breadth-first search in the reverse graph $\overleftarrow{D}$, starting in t. Let $B_s, B_t \subseteq V(D)$ be the vertices found by the two search instances, and let $s \in B_s$ and $t \in B_t$. Set $D := D[B_s \cap B_t]$. For the correctness of the approach observe that B_s contains s and all vertices for which there exists an s–v-path in D, and B_t contains t and all vertices for which there exists a t–v-path in $\overleftarrow{D}$, that is, all vertices for which there exists an v–t-path in D.

Towards determining the running time of applying the BFS-Rule note that we need to remove at most n vertices, which takes $\mathcal{O}(\sum_{v \in V(D)} \deg(v)) \subseteq \mathcal{O}(m)$ time. Computing a breadth-first search takes $\mathcal{O}(n + m)$ time, so overall we need $\mathcal{O}(n + m)$ time to apply the BFS-Rule exhaustively. □

We apply the Cut-Rule by running a modified depth-first search on the underlying undirected graph $G(D)$ starting in s. In the following we describe the algorithm in detail (see also Algorithm 3.1). The approach is based on an algorithm to find cut vertices as shown by Even [Eve11, Chapter 3].

Towards showing that Algorithm 3.1 exhaustively applies the Cut-Rule we first fix some notation. Let T be the search tree rooted in s resulting from the depth-first search computed in lines 3 to 11 of Algorithm 3.1. The edges $E(G(D))$ of the underlying undirected graph can be partitioned into the sets of *tree edges* $E(T)$ and *back edges* $E(G(D)) \setminus E(T)$. During the search, every vertex $s \neq v \neq t$ is given an increasing number $r(v) \geq 2$. We call this number $r(v)$ the *traversal number* of v. Vertices s and t are assigned the traversal numbers zero and one, respectively. The *lowpoint* $\ell(v)$ of v is the lowest traversal number $r(w)$ of a vertex w that can be reached from v by a path (possibly of length zero) downwards along T, followed by *at most one* back edge. It is easy to see that this is the same as the smallest of the following:

Algorithm 3.1: Implementation of the Cut-Rule.

```
1  Let $S$ be a stack, set $k = 2$, and set $r(v) = \infty$ for all $v \in V(D)$.
2  Push $s$ onto $S$.
3  while stack $S$ is not empty do                 (Depth-first search starting in $s$)
4      Pop uppermost vertex $v$ from stack $S$.
5      if $r(v) = \infty$ then
6          if $v = s$ then Set $r(v) = 0$.          (Special numbering for $s$ and $t$)
7          else if $v = t$ then Set $r(v) = 1$.
8          else Set $r(v) = k$ and increase $k$ by one.
9          for $w \in N(v)$ do Push $w$ onto $S$.
10     end
11 end
12 Let $T$ be the search tree rooted in $s$ resulting from the depth-first search above.
13 for $w \in V(T)$, bottom-up towards $s$ do
14     Set $\ell(w) = \min \left\{ \begin{array}{l} r(w), \\ \min\{\ell(v) \mid v \text{ is a child of } w\}, \\ \min\{r(v) \mid \{v,w\} \in E(G(D)) \setminus E(T)\}. \end{array} \right\}$
       (Remove all children $v$ of $w$ that have $\ell(v) \geq r(w)$ and do not lead to $t$)
15     for each child $v$ of $w$ with respect to $T$ do
16         if $w \neq s$ then
17             if $\ell(v) \geq r(w)$ then Remove $v$ and its ancestors.
18         end
19     end
20 end
```

(1) the traversal number of v,

(2) the smallest of the lowpoints of the children of v, and

(3) the smallest of the traversal numbers of those neighbors of v that are reachable by back edges.

The formal computation of $\ell(v)$ can be found in line 14 of Algorithm 3.1. Afterwards, for all vertices $w \neq s$ we remove a child v and its ancestors if $\ell(v) \geq r(w)$. Towards showing that this is exactly the set of vertices as described in the Cut-Rule we show the following lemma.

Lemma 3.9. *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow *on which the BFS-Rule is not applicable. Let $w \in V(D) \setminus \{s\}$ and let v be a child of w with respect to the search tree T. Let T' be the subtree of T rooted at v. Then, for all $v' \in V(T')$, all s–v'-paths and v'–t-paths visit w if and only if $\ell(v) \geq r(w)$.*

Proof. Suppose first that for all $v' \in V(T')$, all s–v'-paths and v'–t-paths visit w. Then there cannot be any back edges connecting vertices of $V(T')$ with vertices of $V(T) \setminus (V(T'))$. Hence, $\ell(v) \geq r(w)$.

Suppose now that $\ell(v) \geq r(w)$. We assume towards a contradiction that t can be reached from $v' \in V(T')$ without visiting w. Clearly, $r(w) \geq r(t)$. If $t \in V(T')$, then $w \neq t$ and $t = v$ or t is an ancestor of v. Thus $\ell(v) \leq r(t)$. By definition, $r(t) < r(w)$, contradicting $\ell(v) \geq r(w)$. Thus $t \notin V(T')$ and we need to bypass w by using a back edge from v' to a descendant of t. Assume that there is such a back edge $\{v', t'\}$ where $t' \in V(T) \setminus (V(T') \cup \{w\})$ and $\ell(t') \leq r(t)$. By Even [Eve11, Lemma 3.4] a back edge may only connect a vertex to one of its descendants. So t' must be a descendant of v', and also of w. Hence, $r(t') < r(w)$. Due to edge $\{v', t'\}$ we have $\ell(v) \leq r(t') < r(w)$, contradicting $\ell(v) \geq r(w)$. Thus back edge $\{v', t'\}$ cannot exist and v cannot reach t without visiting w. Since $r(s) < r(t)$, v cannot reach s without visiting w either. □

With Lemma 3.9 and Observation 3.5 at hand we can prove that the Cut-Rule is applicable in linear time.

Lemma 3.10. *Given an instance of* Maximum Flow *where the BFS-Rule is not applicable, Algorithm 3.1 exhaustively applies the Cut-Rule in $\mathcal{O}(m)$ time.*

Proof. The correctness of Algorithm 3.1 follows from Lemma 3.9 and Observation 3.5.

A depth-first search can be executed in $\mathcal{O}(n + m)$ time. Our modification to the search is to check whether a vertex is s or t, and to assign a different traversal number. This can be accomplished in constant time and thus does not affect the running time of our depth-first search.

The search tree T can be computed in linear time as well. In line 13, the bottom-up ordering of the vertices w can be found by a traversal of T, also running in linear time. The computation of $\ell(w)$ accesses the traversal numbers of other vertices and the lowpoint values of the children of w. The traversal numbers are already known, and since we traverse the vertices of T in a bottom-up manner, the lowpoint values of the children of w are known at this point as well. So the computation of all lowpoint values can be done in $\mathcal{O}(m)$ time.

For the for-loop starting in line 15 observe that iterating over the children of a tree in a bottom-up manner takes $\mathcal{O}(\sum_{v \in V(T)} \deg_T(v)) \subseteq \mathcal{O}(m_T) \subseteq \mathcal{O}(n)$ time. Since a deletion of a vertex v runs in $\mathcal{O}(\deg_D(v))$ time, and every vertex can be visited only once, the deletion step in line 17 can be done in overall $\mathcal{O}(\sum_{v \in V(D)} \deg_D(v)) \subseteq \mathcal{O}(m)$ time. All in all, the time required to run Algorithm 3.1 is in $\mathcal{O}(m)$, given that the BFS-Rule was exhaustively applied. □

Lastly, before showing that we can apply the BFS-Rule, the Cut-Rule and the Deg-2-Rule together exhaustively in quasilinear time, we need to show how to apply the Deg-2-Rule exhaustively.

Lemma 3.11. *The Deg-2-Rule can be applied to a single vertex v in $\mathcal{O}(\log n)$ time and exhaustively in $\mathcal{O}((m + n) \log n)$ time.*

Proof. Let $v \in V(D) \setminus \{s, t\}$ be a vertex with two neighbors u and w. Towards applying the Deg-2-Rule on a vertex $v \in V(D)$, assume that the arc set of D is managed in an AVL-tree. We can then check the degrees of v in constant time. If the only neighbors of v are u and w, then we can check in $\mathcal{O}(\log n)$ time whether arcs (u, w) and (w, u) exist. If any of them do not exist, then we add them to the arc set. Removing v and its incident arcs and increasing the capacities of the arcs between u and w can then be done in $\mathcal{O}(\log n)$ time. If an arc that was added in the process has capacity zero, then we dispose of it.

Since the reduction rule applies to at most $n-2$ vertices, it can be applied exhaustively in $\mathcal{O}(n \log n)$ time if the arcs of D are managed in an AVL-tree, and in $\mathcal{O}((m+n)\log n)$ time otherwise. □

Given Lemmata 3.8, 3.10 and 3.11 it is easy to see that we can exhaustively apply the BFS-Rule, the Cut-Rule and the Deg-2-Rule in quasilinear time:

Observation 3.12. *The BFS-Rule, the Cut-Rule and the Deg-2-Rule can be applied exhaustively in $\mathcal{O}((m+n)\log n)$ time.*

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow.

We apply the BFS-Rule, the Cut-Rule and the Deg-2-Rule in that order. By Lemmata 3.8, 3.10 and 3.11 this can be done in $\mathcal{O}(m+n+m+n\log n) \subseteq \mathcal{O}(m \log n)$ time if $A(D)$ is sorted, and in $\mathcal{O}(m+n+m+(m+n)\log n) \subseteq \mathcal{O}((m+n)\log n)$ time otherwise. Note that deleting the set of vertices $V^w \subseteq V(D)$ that reach s and t only via a cut vertex $w \in V(D)$ does not affect the existence of s–u-paths or u–t-paths for vertices $u \in V(D) \setminus V^w$. This means that if the BFS-Rule was applied exhaustively then it does not become applicable from applying the Cut-Rule. Note further that the operations of the Deg-2-Rule do not change the reachability between the neighbors u and w and the endpoints s and t. Hence, if the BFS-Rule and the Cut-Rule were applied exhaustively beforehand, they do not become applicable from applying the Deg-2-Rule. □

Having shown that we can apply the three reduction rules (BFS-Rule, Cut-Rule and Deg-2-Rule) in quasilinear time, we can advance towards showing that Maximum Flow admits a kernel of size linear in the feedback edge number. As a last prerequisite we show the following lemma upper-bounding the size of an undirected reduced graph linearly in the feedback edge number.

Lemma 3.13. *Let G be an undirected graph that contains n_1 vertices of degree one and n_2 vertices of degree two, and let $F \subseteq E(G)$ be a minimum feedback edge set of size k. Then $n_G \leq 6k + 2n_1 + n_2$ and $m_G \leq 7k + 2n_1 + n_2$.*

Proof. Without loss of generality assume that G is connected (otherwise, consider each connected component separately). Let $F \subseteq E(G)$ be a minimum feedback edge set of G, and let $k := |F|$. Let $V_1(G-F)$, $V_2(G-F)$ and $V_{\geq 3}(G-F)$ be the sets of vertices of $G-F$ that are of degree one, two and at least three, respectively. A vertex $v \in V(G-F)$ is in $V_1(G-F)$ either because it is a degree-one vertex in G, or because it is incident to at least one edge in F. Since removing an edge from G lowers the degree of the two incident vertices, we have $|V_1(G-F)| \leq n_1 + 2k$. With the same argumentation one can also upper-bound the number of degree-two vertices $|V_2(G-F)|$ by $n_2 + 2k$.

Since $G-F$ is a tree, and in a tree we have $m = n-1$, and in any graph H we have $\sum_{v \in V(H)} \deg(v) = 2m_H$, we can upper-bound the number of vertices of degree at least three by solving

$$\begin{aligned} &3|V_{\geq 3}(G-F)| + 2|V_2(G-F)| + |V_1(G-F)| \\ \leq &\sum_{v \in V(G-F)} \deg(v) = 2(|V(G-F)|-1) \\ = &2\big(|V_{\geq 3}(G-F)| + |V_2(G-F)| + |V_1(G-F)| - 1\big) \\ \iff &|V_{\geq 3}(G-F)| \leq |V_1(G-F)| - 2 \leq n_1 + 2k - 2. \end{aligned}$$

Altogether, we have

$$n_G = |V_1(G-F)| + |V_2(G-F)| + |V_{\geq 3}(G-F)| \leq 6k + 2n_1 + n_2.$$

As $G-F$ is a tree, we have $m_G = m_{G-F} + k \leq n_G - 1 + k$, thus $m_G \leq 7k + 2n_1 + n_2$. □

Since any instance of Maximum Flow in which the BFS-Rule, the Cut-Rule and the Deg-2-Rule are not applicable does not contain any vertices of degree less than three, Lemma 3.13 yields a quasilinear-time kernel for Maximum Flow linear in the size of the feedback edge number of the underlying undirected graph.

Theorem 3.14. Maximum Flow *admits a kernel computable in quasilinear time and consisting of at most* $6k+6$ *vertices and at most* $14k+12$ *arcs, where* k *is the feedback edge number of the underlying undirected graph* $G(D)$*.*

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow. We apply the BFS-Rule, the Cut-Rule and the Deg-2-Rule in that order. By Observation 3.4 this results in an equivalent instance $I' = (D', s, t, c')$ and by Observation 3.12 the application is exhaustive and can be done in $\mathcal{O}((m+n)\log n)$ time.

Observe that D' is connected due to the BFS-Rule and that D' contains at most two degree-one vertices, namely s and t. If any other vertex v has (combined) degree one, then all s–v-paths and all v–t-paths visit the unique neighbor w of v. Since w then is a cut vertex, this contradicts the assumption that the Cut-Rule was applied exhaustively. Further, due to the Deg-2-Rule, D' contains at most two vertices of (combined) degree two. By Lemma 3.13, we have $|V(D')| \leq 6k+6$. Since the directed graph may have two arcs (in both directions) between a pair of vertices, we have $|A(D')| \leq 2|E(G(D))| \leq 14k + 12$. □

The best known algorithm for Maximum Flow runs in $\tilde{\mathcal{O}}(n\sqrt{m}\log^2 C)$ time [LS14]. Applying this algorithm on the reduced instance gives us the following.

Corollary 3.15. Maximum Flow *can be solved in* $\mathcal{O}((m+n)\log n + k^{3/2} \cdot \log^2 C)$ *time, where* k *is the feedback edge number of the underlying undirected graph* $G(D)$*.*

3.3 Towards Exhaustively Applying all Reduction Rules

While not all known reduction rules are always needed for finding a problem kernel, it is typically valuable in practice to be able to apply all known reduction rules for a given problem exhaustively in as little time as possible. As shown in Observation 3.12, we can apply three of the four reduction rules (BFS-Rule, Cut-Rule and Deg-2-Rule) in quasilinear time. Our goal is to apply all four reduction rules (including the SVT-Rule) in the same time bound. Unfortunately we were not able to fulfill this goal. This section is dedicated to show the challenges that come with this goal by showing how the reduction rules *depend* on each other, and which dependencies pose algorithmic challenges for the quasilinear exhaustive application. We will then show that the four reduction rules can be applied in $\mathcal{O}(nm)$ time.

We first define what it means if a reduction rule depends on another reduction rule.

Definition 3.16. (Dependency) Let A and B be reduction rules and let I be an instance of Maximum Flow on which reduction rule A is not applicable. We say that reduction rule A *depends* on B if A may become applicable after applying B. We call the dependency *local* if after applying B one can determine in $\mathcal{O}(\log n)$ time on which vertices A becomes applicable.

Figure 3.3 depicts an example instance of Maximum Flow. Following the order of application of the reduction rules as shown below the figures, one can find all dependencies between the four reduction rules and compare with Figure 3.4—a display of all dependencies between the reduction rules. Overall there can be at most $2 \cdot \binom{4}{2} = 12$ dependencies between four reduction rules. By showing which rules do not depend on each other we prove that Figure 3.4 is complete.

Observation 3.17. *The BFS-Rule does not depend on the Cut-Rule, the BFS-Rule and the Cut-Rule do not depend on the Deg-2-Rule, and the SVT-Rule does not depend on the BFS-Rule and the Cut-Rule.*

Proof. It is shown in the proof of Observation 3.12 that the BFS-Rule does not depend on the Cut-Rule and that the BFS-Rule and the Cut-Rule do not depend on the Deg-2-Rule. The SVT-Rule does not depend on the BFS-Rule or on the Cut-Rule since the latter two rules only remove vertices; the arc set of the remaining subgraph is not altered. □

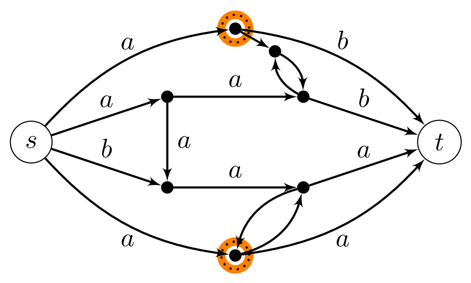
Next, we want to show that some of the dependencies between the reduction rules are local. Towards this we first need to show how to apply the SVT-Rule.

Lemma 3.18. *The SVT-Rule can be applied to a single vertex v in $\mathcal{O}(\log n)$ time and exhaustively in $\mathcal{O}((m+n)\log n)$ time.*

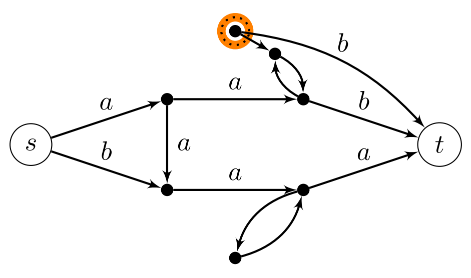
Proof. Assume that the arcs of D are managed in an AVL-tree. Checking the existence and capacity of arcs (s,v) and (v,t) for a given vertex v can be done in $\mathcal{O}(\log n)$ time. Removing at least one of the two arcs and updating their capacities according to the SVT-Rule can then be done in $\mathcal{O}(\log n)$ time. Since the SVT-Rule can be applied to at most $n-2$ vertices, an exhaustive application takes $\mathcal{O}(n \log n)$ time if the arc set of D is managed in an AVL-tree, and $\mathcal{O}(m \log n)$ time otherwise. □

We are now ready to show that the dependencies that are displayed blue (dash-dotted) in Figure 3.4 are local.

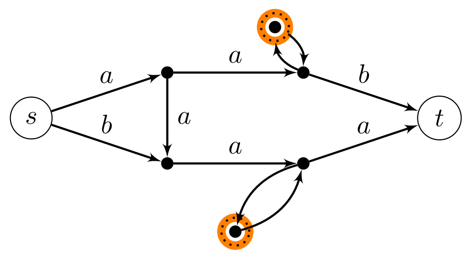
Lemma 3.19. *The dependencies of the Deg-2-Rule on the BFS-Rule, the Cut-Rule and the SVT-Rule as well as the dependency of the SVT-Rule on the Deg-2-Rule are local.*



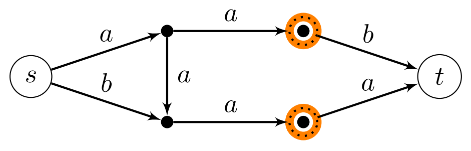
(a) SVT-Rule ($a < b$)



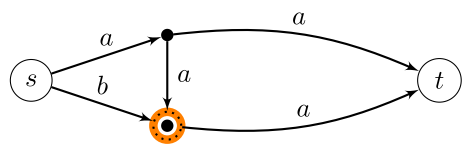
(b) BFS-Rule



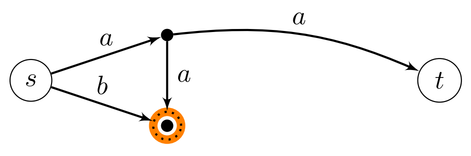
(c) Cut-Rule



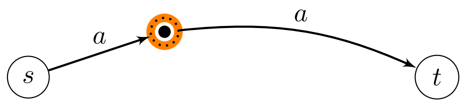
(d) Deg-2-Rule



(e) SVT-Rule ($a < b$)



(f) BFS-Rule



(g) Deg-2-Rule

Figure 3.3: An example instance of Maximum Flow which can be reduced to a path st by applying the four reduction rules in the correct order. Applying the named reduction rule on the marked vertices (orange, dotted) yields the instance shown in the next figure. Assume that we have capacities $a < b$. Note that if in step (f) the arc with capacity a at the marked vertex were directed the other way, then the Deg-2-Rule would become applicable, showing that the Deg-2-Rule depends on the SVT-Rule.

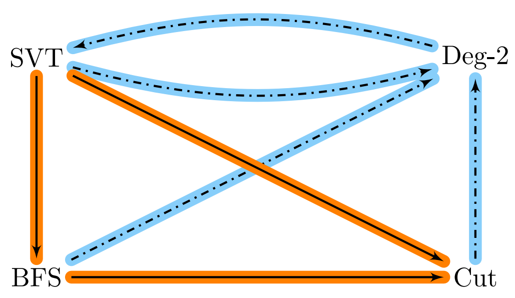
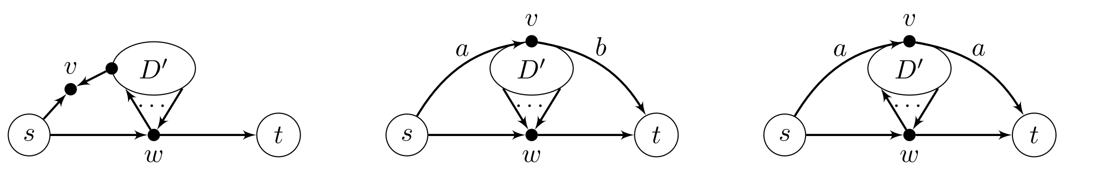


Figure 3.4: Displaying the dependencies of the four reduction rules. An arc (A, B) means that reduction rule B depends on A. The dependency is local if the arc is blue (dash-dotted, proven in Lemma 3.19). If the arc is orange (solid), then it is not known whether the dependency is local.



(a) Rule A: BFS-Rule, rule B: Cut-Rule **(b)** Rule A: SVT-Rule, rule B: BFS-Rule (assume $a < b$) **(c)** Rule A: SVT-Rule, rule B: Cut-Rule

Figure 3.5: The challenge in the three "non-local" dependencies: By applying rule A on v (or svt), rule B become applicable on subgraph D' in the examples.

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow and assume that the Deg-2-Rule is not applicable. Observe that if the BFS-Rule or 3.3 applies to some vertex $v \in V(D) \setminus \{s, t\}$, then it is removed and the degree of the neighbors of v is updated. If the degree of a neighbor $u \in N(v) \setminus \{s, t\}$ becomes two, then the Deg-2-Rule becomes applicable on u. By keeping a counter for the degree of every vertex, we can determine in constant time whether u has degree two; thus the dependency of the Deg-2-Rule on the BFS-Rule and the Cut-Rule is local.

For showing that the dependency of the Deg-2-Rule on the SVT-Rule is local, let $P = svt \subseteq D$. When applying the SVT-Rule on P, then only the degrees of s, v and t are changed. When removing at least one of the arcs, the degree counter of s, v and t is updated; thus we can check in constant time whether the Deg-2-Rule becomes applicable on v.

Lastly, we show that the SVT-Rule depends locally on the Deg-2-Rule. Assume that the SVT-Rule is not applicable on I and that D contains a path of the type $P = svwt$ or $Q = swvt$, where w is a degree-two vertex. Then, after applying the Deg-2-Rule to w, the SVT-Rule becomes applicable on P or Q. Since the Deg-2-Rule only adds arcs between the neighbors of the removed vertex w if the arcs did not exist yet, the SVT-Rule can only become applicable to v, which can be checked in $\mathcal{O}(\log n)$ time. □

We can exhaustively apply the BFS-Rule, the Cut-Rule and the Deg-2-Rule together in quasilinear time, and we can exhaustively apply the SVT-Rule by itself in quasilinear time. Ideally one could apply all four reduction rules together in quasilinear time as well. Due to the dependencies of the BFS-Rule and the Cut-Rule on the SVT-Rule and the dependency of the Cut-Rule on the BFS-Rule, this turns out to be challenging. We show examples for these dependencies in steps a, b and e of Figure 3.3. Note that after the BFS-Rule or the Cut-Rule become applicable due to one of their dependencies, they do not necessarily become applicable only on a single vertex. It is as well possible that they become applicable on multiple vertices, as shown in Figure 3.5.

We do not know any way to check in less than $\mathcal{O}(m)$ time whether the BFS-Rule or the Cut-Rule is applicable even to a single vertex, and we leave it as an open question whether it is possible to apply the four reduction rules exhaustively in quasilinear time. Instead, in the following, we show a simple way to exhaustively apply the four reduction rules together within a less strict time bound.

Algorithm 3.2: Exhaustive application of the four reduction rules.

```
1 Apply the SVT-Rule exhaustively.
2 while the BFS-Rule, the Cut-Rule or the Deg-2-Rule is applicable do
3     Apply the BFS-Rule and then the Cut-Rule exhaustively.
4     while the Deg-2-Rule is applicable to some vertex $v$ do
5         Apply the Deg-2-Rule on $v$.
6         if there is an $u \in N(v)$ such that there is a path $sut$ then Apply the
           SVT-Rule on $sut$.
7     end
8 end
```

Theorem 3.20. *The Reduction Rules 3.2 to 3.5 (BFS-Rule, Cut-Rule, Deg-2-Rule and SVT-Rule) can be applied exhaustively in $\mathcal{O}(nm)$ time.*

Proof. Let $I = (D, s, t, c)$ be an instance of MAXIMUM FLOW. We apply the four reduction rules by running Algorithm 3.2.

Note that when starting an iteration of the first while-loop (line 2), the SVT-Rule is not applicable. By Observation 3.17 it does not become applicable after applying the BFS-Rule or the Cut-Rule, but by Lemma 3.19 it depends locally on the Deg-2-Rule. It follows that Algorithm 3.2 applies the four reduction rules exhaustively.

Next, observe that during the applications of the BFS-Rule and the Cut-Rule we are able to remember all vertices that have degree two with constant time overhead. Also, after applying the SVT-Rule on a path svt we can check whether v is of degree two. Lastly, also the Deg-2-Rule can generate new degree-two vertices, which can also be remembered in constant time. Thus, when entering the second while-loop (line 4), we can determine a vertex v of degree two in constant time.

Finally, we discuss the running time of Algorithm 3.2. Note first that an iteration of the second while-loop takes $\mathcal{O}(\log n)$ time, since the Deg-2-Rule and the SVT-Rule take $\mathcal{O}(\log n)$ time to apply (see Lemmata 3.11 and 3.18) and are applied at most once per iteration. Further, since in each iteration of the second while-loop the Deg-2-Rule is applied, the number n of vertices in D is decreased by one in every iteration. Note that when entering the first while-loop, one out of the Reduction Rules 3.2 to 3.4 (BFS-Rule, Cut-Rule, Deg-2-Rule) is applicable. Each application of these rules removes a vertex. Hence, for each iteration over the first while-loop, n is decreased by at least one. It follows that for each vertex removed, we have spent either $\mathcal{O}(\log n)$ or $\mathcal{O}(m)$ time. Since in weakly connected directed graphs $m \geq 2(n-1)$, we have $\mathcal{O}(\log n) \subseteq \mathcal{O}(m)$. Thus it takes $\mathcal{O}(nm)$ time to run Algorithm 3.2. □

While the $\mathcal{O}(nm)$ time bound for Algorithm 3.2 matches the running time of the best MAXIMUM FLOW algorithm that does not depend on the maximum capacity [Orl13], it is outperformed by the $\tilde{\mathcal{O}}(m\sqrt{n}\log^2 C)$-time algorithm by Lee and Sidford [LS14] whenever $C \in 2^{o(\sqrt[4]{n})}$. Still, Algorithm 3.2 is farly less complex than the latter two algorithms for MAXIMUM FLOW. Thus, while theoretically the exhaustive application of the four reduction rules yields no benefit, it is possible that in practice it leads to a significant improvement of the running time.

Chapter 4

When Maximum Flow is Not Easy to Solve

In this chapter we want to present results of two kinds: first, that fixed-parameter algorithms with certain parameters are unlikely for Maximum Flow, and second, that there are presumably no faster[1] algorithms for Maximum Flow on certain classes of graphs. Herein, the first kind of results rely on a notion called *General Problem hardness* by Bentert et al. [Ben+17], and the second kind is inspired by it.

Both of these kinds of results usually rely on running time lower bounds for the general problem. Having been a hot research topic for decades, the best running time upper bounds for Maximum Flow have been improved steadily. When, before 2013, it still was an open question whether the "magical" [HW07] upper bound of $\mathcal{O}(nm)$ will be reached, by now it has been met [Orl13] and even improved upon ($\tilde{\mathcal{O}}(m\sqrt{n}\log^2 C)$ [LS14]). But to this date, no running time lower bounds are known for Maximum Flow. Even further, Carmosino et al. [Car+16] have shown that it is unlikely that one can find a lower bound for Maximum Flow with one of the popular tools for showing lower bounds for polynomial-time solvable problems, namely fine-grained reductions [VW15] in combination with the Strong Exponential Time Hypothesis [IPZ01]. Thus, instead of ruling out fast algorithms for Maximum Flow with respect to certain parameters or structural restrictions to the instance, we present results that show that such fast algorithms would lead to algorithms for the general problem (unparameterized or without structural restrictions) that are significantly faster than the best algorithms known to date. Achieving such running times for the general Maximum Flow problem would be considered a breakthrough result. From a positive point of view, some of the results presented in this chapter can also be used as guidelines for designing Maximum Flow algorithms. For example, we show that one can modify any instance of Maximum Flow in linear time such that it becomes bipartite (Corollary 4.6). Hence, when designing an algorithm, one can assume that the input is bipartite. This "technique" was already used by Weihe [Wei97] when presenting an $\mathcal{O}(n \log n)$-time algorithm for Maximum Flow on planar graphs: He formulated as an assumption that every instance of Maximum Flow can be modified in such a way that is has maximum degree three. In Section 4.2

[1]By "faster" we mean algorithms with running times faster than the best current algorithms for general Maximum Flow.

we present in great detail how to construct an instance with maximum degree three from any instance of Maximum Flow in Section 4.2.

Chapter outline. In Section 4.1 we introduce the notion of General Problem hardness as by Bentert et al. [Ben+17] and, as a warm-up, present four simple General Problem hardness results for Maximum Flow. In Section 4.2 we show that a fast fixed-parameter algorithm for Maximum Flow with respect to the maximum degree of the input graph implies an equivalently fast unparameterized algorithm for Maximum Flow, and that Maximum Flow is General-Problem-hard with respect to the maximum degree for instances with large capacities, that is, the largest capacity of the instance is at least as large as the maximum degree. This result is based on an observation made by Weihe [Wei97]. Lastly, we show in Section 4.3 that a linear-time algorithm for Maximum Flow on graph classes that contain complete graphs implies an $\mathcal{O}(n^2)$-time algorithm for Maximum Flow on general graphs.

4.1 A Primer on General Problem Hardness

In this section we describe *General Problem hardness (GP hardness)*—a notion introduced by Bentert et al. [Ben+17]—and present four applications of it on the Maximum Flow problem.

The General Problem hardness relates parameterized problems to their unparameterized (general) counterpart and proves that the parameterized version of a problem is not easier to solve than the unparameterized version of the problem. It is defined as follows:

Definition 4.1 ([Ben+17])**.** Let $P \subset \Sigma^* \times \mathbb{N}$ be a parameterized problem, let $f\colon \mathbb{N} \to \mathbb{N}$ be a function, and let $Q \subseteq \Sigma^*$ be the unparameterized decision problem associated to P. We call P *ℓ-General-Problem-hard(f) (ℓ-GP-hard(f))* if there exists an algorithm $\mathcal{A}$ transforming any input instance I of Q into a new instance (I', k') of P such that

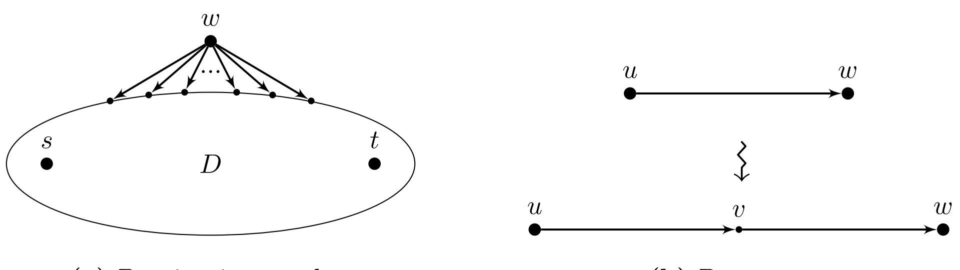
(1) $\mathcal{A}$ runs in $\mathcal{O}(f(|I|))$ time, (3) $k' \leq \ell$, and
(2) $I \in Q \iff (I', k') \in P$, (4) $|I'| \in \mathcal{O}(|I|)$.

We call P *General-Problem-hard(f) (GP-hard(f))* if there exists an integer k such that P is k-GP-hard(f). We omit the running time and call P *k-General-Problem-hard (k-GP-hard)* if f is a linear function.

The central idea behind GP hardness is that if one can preclude the existence of an $\mathcal{O}(f(|I|)$-time algorithm for Q and is able prove that P is GP-hard, then one can also preclude an algorithm that solves P in $\mathcal{O}(g(k) \cdot f(|I|))$ time for a computable function g. This is put to record in the following lemma.

Lemma 4.2 ([Ben+17])**.** *Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function, let $P \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem that is ℓ-GP-hard(f), and let $Q \subseteq \Sigma^*$ be the unparameterized decision problem associated to P. If there is an algorithm solving each instance (I, k) of P in $\mathcal{O}(g(k) \cdot f(|I|))$ time, then there is an algorithm solving each instance I' of Q in $\mathcal{O}(f(|I'|))$ time.*

We are now going to present four simple GP hardness results for Maximum Flow with respect to the parameters domination number, degeneracy, bisection width and



(a) Domination number **(b)** Degeneracy

Figure 4.1: Constructions for showing GP hardness with respect to the domination number and the degeneracy of D.

distance to bipartite graphs. With the four results we can show that linear-time fixed-parameter algorithms for Maximum Flow with respect to these parameters imply linear-time algorithms for Maximum Flow.

Recall that a dominating set of an undirected graph G is a vertex subset $V' \subseteq V(G)$ such that $N[V'] = V(G)$, and that the *domination number* of G is the smallest number γ such that there exists a dominating set of size γ. The *bisection width* of G is the smallest number of edges that need to be removed such that G has two connected components whose number of vertices differ by at most one. The *degeneracy* of G is the smallest number d such that every subgraph of G contains at least one vertex of degree at most d. Lastly, the *distance to bipartite graphs* of G is the number of vertices that need to be removed such that the vertices of the resulting graph can be partitioned into two independent sets.

In order to show that Maximum Flow is GP-hard with respect to the domination number of the underlying undirected graph, we only need to add one vertex to the instance.

Proposition 4.3. Maximum Flow *is* 1*-GP-hard with respect to the domination number γ of the underlying undirected graph $G(D)$ of D.*

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow. We create an instance $I' = (D', s, t, c')$ such that $G(D')$ has domination number one by adding a source vertex that is adjacent to all vertices in $V(D)$. Formally, we set $V(D') := V(D) \cup \{w\}$ and $A(D') := A(D) \cup \{(w, u) \mid u \in V(D)\}$ and set $c'(w, u) := 1$ for all $u \in V(D)$ and $c'(u, v) := c(u, v)$ for all $(u, v) \in A(D)$. See Figure 4.1a for an illustration.

Since $\mathbb{N}(w) \cup \{w\} = V(D')$, the singleton $\{w\}$ is a dominating set in D', and the domination number γ of D is one. Further, w is a source, and thus by Observation 3.4, no flow will pass through w. Since $D = D' - \{w\}$, the maximum s–t-flow is the same in both graphs, and hence, the two instances are equivalent, and constructing I' takes $\mathcal{O}(|I|)$ time. $\square$

It is similarly easy to see that Maximum Flow is GP-hard with respect to the bisection width of the underlying undirected graph $G(D)$.

Proposition 4.4. Maximum Flow *is* 0*-GP-hard with respect to the bisection width of the underlying undirected graph $G(D)$ of D.*

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow. Let D' be the disjoint union of two copies of D, and let $c' \colon V(D)^2 \to \mathbb{N}_0$ be the capacity function that assigns the same capacities to the copies of D as c. Let $s', t' \in V(D')$ be the vertices corresponding to the vertices s and t in one of the two copies of D. Then instance $I' = (D', s', t', c')$ has bisection width zero. Since the flow can only be sent within one of the two components of D', and each of the components is a copy of D with the same capacities as c, the instances I and I' are equivalent. Clearly, constructing I' takes $\mathcal{O}(|I|)$ time. □

Next, we are going to show that every instance of Maximum Flow can be modified such that it has degeneracy two. Recall that planar graphs have degeneracy five, and Maximum Flow on planar graphs can be solved in $\mathcal{O}(n \log n)$ time. Note that the converse is not necessarily true (a K_5 with subdivided edges has degeneracy two and is not planar), and hence, the following result does not imply that every instance of Maximum Flow can be made planar.

Proposition 4.5. Maximum Flow *is* 2*-GP-hard with respect to the degeneracy d of the underlying undirected graph $G(D)$ of D.*

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow. We create an instance $I' = (D', s, t, c')$ that has degeneracy two by subdividing each arc. Formally, we set

$$\begin{aligned} V(D') &:= V(D) \cup \{v \mid (u, w) \in A(D)\}, \\ A(D') &:= \{(u, v), (v, w) \mid (u, w) \in A(D)\}, \text{ and} \\ c'(u, v) &:= c'(v, w) := c(u, w), \text{ for each } (u, w) \in A(D). \end{aligned}$$

See Figure 4.1b for an illustration.

As each of the newly introduced vertices v has degree two, each subgraph $H \subseteq G(D')$ has at least one vertex of degree at most two, and hence, the degeneracy of $G(D')$ is at most two. Also note that the Deg-2-Rule applies to every such vertex v. Hence, it follows from Observation 3.6 that instances I and I' are equivalent. Constructing I' takes $\mathcal{O}(|I|)$ time. □

Note that the reduction of Proposition 4.5 produces bipartite graphs: The original vertices $V(D)$ are the one partition, and the vertices that subdivide the arcs are the other partition. It follows that Maximum Flow is GP-hard with respect to the distance to bipartite graphs of the underlying undirected graph.

Corollary 4.6. Maximum Flow *is* 0*-GP-hard with respect to the distance to bipartite graphs of the underlying undirected graph.*

4.2 Low Maximum Degree Does Not Help

In this section we will show that a linear-time fixed-parameter algorithm for Maximum Flow with respect to the maximum degree $\Delta(D)$ would imply that there exists a quasilinear-time algorithm for the general problem. This would provide an algorithm for Maximum Flow that improves on the running time of the currently best algorithm for Maximum Flow—the $\mathcal{O}(m\sqrt{n} \log^2 C)$-time algorithm by Lee and Sidford [LS14]. Our

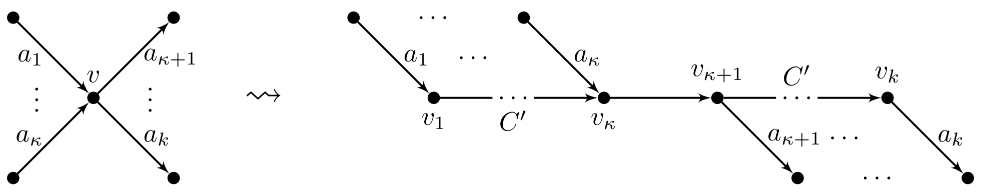


Figure 4.2: How a vertex v of degree higher than three is replaced by a path gadget. The path arcs have capacity $C' \coloneqq C \cdot \Delta(D)$.

result is based on the observation by Weihe [Wei97] that any vertex of high degree can be replaced by a cycle in which each vertex has degree three. We first state the main theorem of this section.

Theorem 4.7. *Let $f\colon \mathbb{N} \to \mathbb{N}$ be a computable function and let $\gamma \geq 0$. If there exists an $\mathcal{O}(f(\Delta(D)) \cdot |I|^{1+\gamma})$-time algorithm for* Maximum Flow*, then there exists an $\mathcal{O}((|I| + m \cdot \log(\Delta(D))^{1+\gamma})$-time algorithm for* Maximum Flow.

We will start off by showing how to construct an instance with maximum degree three from any instance of Maximum Flow. Next, we will prove that the constructed instance can be computed in quasilinear time, and increases only quasilinearly in size. Finally, we will show that it is equivalent to the original instance, that is, the value of the maximum flow of the two instances is the same.

Construction 4.8. We assume that the instance I is reduced with respect to the BFS-Rule, which removes any vertex that cannot be reached by s and cannot reach t. As shown in the previous chapter, this rule gives us an instance equivalent to the input instance (see Observation 3.4) and can be applied in $\mathcal{O}(n + m)$ time (see Lemma 3.8).

We create instance $I' = (D', s', t', c')$ from $I = (D, s, t, c)$: Let $C \coloneqq \max\{c(a) \mid a \in A(D)\}$ be the highest capacity of D. We initialize $c' \coloneqq c$ and

$$D' = (V(D) \cup \{s', t'\}, A(D) \cup \{(s', s), (t, t')\}).$$

The capacities of the new arcs are $c'(s', s) = C \cdot \deg_D(s)$ and $c'(t, t') = C \cdot \deg_D(t)$. We replace each vertex $v \in V(D') \setminus \{s', t'\}$ of degree higher than three by a gadget called *path gadget* (as depicted in Figure 4.2), which is constructed as follows:

Let $\kappa = \deg^-(v)$, let $k = \deg(v)$ and let $a_1, \ldots, a_\kappa$ be the incoming arcs of v and let $a_{\kappa+1}, \ldots, a_k$ be the outgoing arcs of v. Introduce *path vertices* $v_1, \ldots v_k$ and *path arcs* (v_i, v_{i+1}) with capacities $c'(v_i, v_{i+1}) = C \cdot \deg(v)$ for every $1 \leq i < k$. For $1 \leq i \leq k$, change the endpoint v of a_i to v_i and set $c'(a_i) = c(a_i)$. We call arcs a_i *altered arcs*. We call an arc *standard arc* if it is neither an altered arc, nor a path arc, nor any of the two arcs (s', s) and (t, t'). ◇

Having presented the construction we now show how to apply it efficiently, and how large the resulting instance becomes.

Lemma 4.9. *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow *and let $I' = (D', s', t', c')$ be the instance obtained when applying Construction 4.8 on I. Then D' has $\mathcal{O}(m_D)$ vertices and arcs and the highest capacity C' in I' is $C \cdot \Delta(D)$. Construction 4.8 can be performed in $\mathcal{O}(|I| + m \cdot \log(\Delta(D)))$ time.*

Proof. Let $n = |V(D)|$, $m = |A(D)|$, and let $n' = |V(D')|$ and $m' = |A(D')|$. Consider a vertex $v \in V(D)$ of degree greater than three. Then, the path gadget replacing v has $\deg(v)$ vertices and $\deg(v) - 1$ arcs. Hence, there are at most $2 + \sum_{v \in V(D)} \deg(v) = 2m + 2 \in \mathcal{O}(m)$ new vertices and arcs.

Let s^* and t^* be the unique neighbors of s' and t', respectively. Note that for a path arc $a \in A(D')$ corresponding to $v \in V(D)$ we have $c(a) = C \cdot \deg(v) > C$, and also, $c(s', s^*) = C \cdot \deg(s) > C$ and $c(t^*, t') = C \cdot \deg(t) > C$. Since all other arcs have capacity at most C, the highest capacity in I' is $C' := C \cdot \Delta(D)$. Note that $|I| \in \mathcal{O}(n + m \log C)$. Since more space is required to store the capacities, we have

$$\begin{aligned} |I'| &\in \mathcal{O}(m \log(C \cdot \Delta(D)) \\ &\subseteq \mathcal{O}(m \cdot (\log C + \log(\Delta(D)))) \\ &\subseteq \mathcal{O}(|I| + m \log(\Delta(D))). \end{aligned}$$

Adding the vertices s' and t' and the corresponding arcs requires constant time. For each $v \in V(D) \setminus \{s', t'\}$, $\deg(v) > 3$, we introduce $\deg(v)$ new vertices and $\deg(v) - 1$ new arcs of capacity $C \cdot \deg(v)$. Hence, replacing all vertices of degree higher than three by path gadgets requires $\mathcal{O}(\sum_{v \in V(D)} \deg(v) \cdot \log(C \cdot \deg(v))) \subseteq \mathcal{O}(m \log(C \cdot \Delta(D)))$ time. Together with Reduction Rule 3.2, we need $\mathcal{O}(n + m \cdot \log(C \cdot \Delta(D))) \subseteq \mathcal{O}(|I| + m \cdot \log(\Delta(D)))$ time to create the new instance I'. □

Next, we show that Construction 4.8 is correct.

Lemma 4.10. *Let $I = (D, s, t, c)$ be an instance of* Maximum Flow. *Then the instance $I' = (D', s', t', c')$ obtained from applying Construction 4.8 on I is equivalent to I.*

Proof. To show that instance I' has the same maximum flow as I, we will construct an s'–t'-flow f' on I' from an s–t-flow f on I. We will then show that f' is valid if f is valid. Lastly, we will discuss that f' is a maximum flow for I', assuming that f is a maximum flow for I.

By construction, vertices s' and t' have exactly one neighbor. We denote by s^* and t^* the unique neighbors of s' and t' respectively, and set

$$f'(s', s^*) = \sum_{(s,w) \in A(D)} f(s, w) \qquad \text{and} \qquad f'(t^*, t') = \sum_{(u,t) \in A(D)} f(u, t).$$

For each vertex v of indegree κ and combined degree $k > 3$ replaced by a path gadget, we do the following. Let $v_1, \ldots, v_k$ be the path vertices, and, for $1 \leq i \leq k$, let a_i be the (unique) altered arc incident to v_i. For each altered arc a_i, $1 \leq i \leq k$, set $f'(a_i) = f(a_i)$. For each path arc $(v_i, v_i + 1)$, $1 \leq i < k$, set

$$f'(v_i, v_{i+1}) = \sum_{j=1}^{\min\{i,\kappa\}} f'(a_j) - \sum_{j=\kappa+1}^{i} f'(a_j), \tag{4.1}$$

that is, the sum of the flows of the arcs whose source are $v_1, \ldots, v_i$ minus the sum of the flows of the arcs whose destination is $v_1, \ldots, v_i$. Lastly, for all standard arcs $(u, w) \in A(D')$, set $f'(u, w) = f(u, w)$. Note that $\mathrm{val}(f) = \sum_{(s,w) \in A(D)} f(s, w) = \mathrm{val}(f')$.

As one can see, the flow f' is the same as f on all altered and standard arcs. The capacities of those arcs are also the same in D and D'. Let $v \in V(D)$ be of degree $k > 3$. Then, it follows from equation (4.1) that for $1 \leq i < k$ we have $f'(v_i, v_{i+1}) \leq k \cdot C$. Lastly we have $f(s', s^*) \leq C \cdot \deg(s)$ and $f(t^*, t') \leq C \cdot \deg(t)$. Thus, f' follows the capacity constraint. We also obtain that f' follows the conservation constraint for all vertices except for those incident to the path arcs, that is, the path vertices. To show that the conservation constraint holds for path vertices as well, consider a vertex v with $\deg_D(v) > 3$ that is replaced by a path gadget with vertices $v_1, \dots, v_k$. We know exactly which arcs are incident to each of the path vertices. Thus, for $1 \leq i \leq \kappa$ we have

$$\begin{aligned}
&\sum_{(x,v_i)\in A(D')} f'(x, v_i) - \sum_{(v_i,x)\in A(D')} f'(v_i, x)\\
&= f'(a_i) + \sum_{j=1}^{\min\{i-1,\kappa\}} f'(a_j) - \underbrace{\sum_{j=\kappa+1}^{i-1} f'(a_j)}_{=0} - \Big(\sum_{j=1}^{\min\{i,\kappa\}} f'(a_j) - \underbrace{\sum_{j=\kappa+1}^{i} f'(a_j)}_{=0} \Big)\\
&= f'(a_i) + \sum_{j=1}^{i-1} f'(a_j) - \sum_{j=1}^{i} f'(a_j)\\
&= f'(a_i) - f'(a_i) = 0.
\end{aligned}$$

Analogously, for $\kappa < i \leq k$ we have

$$\begin{aligned}
&\sum_{(x,v_i)\in A(D')} f'(x, v_i) - \sum_{(v_i,x)\in A(D')} f'(v_i, x)\\
&= \sum_{j=1}^{\min\{i-1,\kappa\}} f'(a_j) - \sum_{j=\kappa+1}^{i-1} f'(a_j) - \Big(\sum_{j=1}^{\min\{i,\kappa\}} f'(a_j) - \sum_{j=\kappa+1}^{i} f'(a_j) + f'(a_i) \Big)\\
&= \underbrace{\sum_{j=1}^{\kappa} f'(a_j) - \sum_{j=1}^{\kappa} f'(a_j)}_{=0} - \sum_{j=\kappa+1}^{i-1} f'(a_j) + \sum_{j=\kappa+1}^{i} f'(a_j) - f'(a_i)\\
&= f'(a_i) - f'(a_i) = 0,
\end{aligned}$$

and hence, f' also follows the conservation constraints for the path vertices, giving us that f' is a valid flow under the assumption that f is valid.

Next, we will show that f' is a maximum flow if and only if f is a maximum flow. As in the construction, we denote by $v_1, \dots, v_{\deg(v)} \in V(D')$ the path vertices replacing $v \in V(D)$. As a simplification, for a vertex $v \in V(D)$ that is not replaced by a path gadget, we denote the corresponding vertex in D' by v_1.

We claim that there is a path $v^1 \dots v^p$ in D if and only if there is a path

$$v^1_{i_1} v^1_{i_1+1} \dots v^1 j_1 v^2_{i_2} \dots v^2_{j_2} \dots v^{p-1}_{j_{p-1}} v^p_{i_p} \dots v^p_{j_p-1} v^p_{j_p}$$

in D', where $1 \leq i_\ell \leq j_\ell \leq \deg_D(v^k)$, $1 \leq \ell \leq p$. That is, D' contains a path going along subsets of those path vertices corresponding to $v^1, \dots, v^p$.

$\Rightarrow$: Assume that there is a path $v^1 \dots v^p$ in D. In D', we have a u_i–u_j-path for each vertex $u \in V(D)$, $\deg_D(u) > 3$, where $1 \leq i \leq j \leq \deg_D(u)$. If $\deg(u) \leq 3$, then we have only a single corresponding vertex $u_1 \in D'$, and we have a u_i–u_j-path for $i = j = 1$. Hence, we only need to show that we have the arcs $(v^\ell_{j_\ell}, v^{\ell+1}_{i_{\ell+1}}) \in A(D')$, for $1 \leq \ell < p$. If $\deg_D(v^\ell), \deg_D(v^{\ell+1}) \leq 3$, then the arc $(v^\ell_{j_\ell}, v^{\ell+1}_{i_{\ell+1}})$ exists for $j_\ell = i_{\ell+1} = 1$, since the arc and both of its endpoints were copied from D to D'. If $\deg_D(v^\ell) > 3$ and $\deg_D(v^{\ell+1}) \leq 3$, then the source of the arc $(v^\ell, v^{\ell+1}) \in A(D)$ is moved to one of the path vertices $v^\ell_{j_\ell}$. Thus, there is an arc $(v^\ell_{j_\ell}, v^{\ell+1}_{i_{\ell+1}}) \in A(D')$ for $\deg^-_D(v^\ell) < j_\ell \leq \deg_D(v^\ell)$ and $i_{\ell+1} = 1$. Note that j_ℓ cannot be smaller, since the path vertices $v^\ell_1, \dots, v^\ell_{\deg^-_D(v^\ell)}$ are only incident to path arcs and incoming altered arcs. Similarly, if $\deg_D(v^\ell) \leq 3$ and $\deg_D(v^{\ell+1}) > 3$, then there is an arc $(v^\ell_{j_\ell}, v^{\ell+1}_{i_{\ell+1}}) \in A(D')$ for $j_\ell = 1$ and $1 \leq i_{\ell+1} \leq \deg^-_D(v^{\ell+1})$. Lastly, if $\deg_D(v^\ell), \deg_D(v^{\ell+1}) > 3$, then both the source and the destination of arc $(v^\ell, v^{\ell+1}) \in A(D)$ are modified by the construction such that it starts at a path vertex $v^\ell_{j_\ell}$ and ends at a path vertex $v^{\ell+1}_{i_{\ell+1}}$. Hence, there is an arc $(v^\ell_{j_\ell}, v^{\ell+1}_{i_{\ell+1}}) \in A(D')$, where $\deg^-_D(v^\ell) < j_\ell \leq \deg_D(v^\ell)$ and $1 \leq i_{\ell+1} \leq \deg^-_D(v^\ell)$. Note that for each $v^\ell \in V(D)$, $1 \leq \ell \leq p$, we have $i_\ell = j_\ell = 1$ if $\deg_D(v) \leq 3$ and $i_\ell \leq \deg^-(v^\ell) < j_\ell$ if $\deg_D(v) > 3$. Hence, there is a path from $v^\ell_{i_\ell}$ to $v^\ell_{j_\ell}$ in D'. Altogether, there is a path $v^1_{i_1}, \dots, v^1_{j_1}, \dots, v^p_{i_p}, \dots v^p_{j_p}$ in D'.

$\Leftarrow$: Assume that there is a path $P = v^1_{i_1} \dots v^1_{j_1} \dots v^p_{i_p} \dots v^p_{j_p}$ in D'. Note that to undo the construction, that is, to obtain D from D', one needs to contract all path vertices $u_1, \dots, u_{\deg_D(u)}$ into a single vertex u. Hence, the subpaths going along the vertices $v^\ell_{i_\ell}, \dots, v^\ell_{j_\ell} \in V(D')$, $1 \leq \ell \leq p$, correspond to vertex $v^\ell \in V(D)$. Observe that for any $a \neq b \in \{1, \dots, p\}$ we have $v^a \neq v^b$ since P must visit $v^a_{\deg^-(v^a)}$ and $v^b_{\deg^-(v^b)}$, and if $a = b$, then P is not a path. Further, any altered arc $(v^\ell_{j_\ell}, v^{\ell+1}_{i_{\ell+1}}) \in A(D')$, $1 \leq \ell < p$, corresponds to arc $(v^\ell, v^{\ell+1}) \in A(D)$. Hence, there is a path $v^1 \dots v^p$ in D.

With this, we have shown the claim. This gives us that for each s–t-path in D, we have an s'–t'-path in D' that goes along the standard and altered arcs corresponding to the arcs of the path in D. Observe that each s'–t'-path in D' must contain at least one standard arc or one altered arc, or we have $s = t$ in D. These arcs in D' have the same capacity, and the same flow sent along them, as their corresponding arcs in D. This gives us that if an arc in D is saturated by f, then the corresponding arc in D' (a standard or altered arc) is saturated by f'. If f is a maximum flow, then every s–t-path in D contains at least one saturated arc. Simultaneously, in D', every s'–t'-path contains at least one arc saturated by f', and hence, f' is a maximum flow. $\square$

Combining the two lemmata, it follows directly that for every instance of Maximum Flow, we can create an equivalent instance with maximum degree three in linear time.

Proposition 4.11. *Given an instance $I = (D, s, t, c)$ of* Maximum Flow*, one can construct in $\mathcal{O}(|I| + m \cdot \log(\Delta(D)))$ time an equivalent instance $I' = (D', s', t', c')$ of size $|I'| \in \mathcal{O}(|I| + m \cdot \log(\Delta(D)))$, such that $\Delta(D') \leq 3$ and $C' = C \cdot \Delta(D)$, where C and C' are the maximum capacities in c and c' respectively.*

This proposition gives us that, if we have a linear fixed-parameter algorithm for Maximum Flow with respect to the maximum degree of the input graph, then we have a similarly efficient algorithm for the general problem.

Theorem 4.7 (Restated)**.** *Let $f\colon \mathbb{N} \to \mathbb{N}$ be a computable function and let $\gamma \geq 0$. If there exists an $\mathcal{O}(f(\Delta(D)) \cdot |I|^{1+\gamma})$-time algorithm for* Maximum Flow*, then there exists an $\mathcal{O}((|I| + m \cdot \log(\Delta(D))^{1+\gamma})$-time algorithm for* Maximum Flow*.*

Proof. By Proposition 4.11, we can compute an equivalent instance $I' = (D', s', t', c')$ with $\Delta(D') \leq 3$ in $\mathcal{O}(|I| + m \cdot \log(\Delta(D)))$ time. Since $|I'| \in \mathcal{O}(|I| + m \cdot \log(\Delta(D)))$, applying the $\mathcal{O}(f(\Delta(D)) \cdot |I|^{1+\gamma})$-time algorithm on I', $\gamma \geq 0$, gives us a running time of

$$\mathcal{O}\Big(f(\Delta(D') \cdot \big(|I| + m\log(\Delta(D))\big)^{1+\gamma}\Big) \subseteq \mathcal{O}\Big(\big(|I| + m\log(\Delta(D))\big)^{1+\gamma}\Big). \qquad \square$$

Recall that to this date the best known running time for solving Maximum Flow is $\tilde{\mathcal{O}}(m\sqrt{n}\log^2 C)$ [LS14]. They state that with the notation $\tilde{\mathcal{O}}(\cdot)$, they hide polylogarithmic factors in the number of arcs in the input graph; thus the running time of their algorithm is $\mathcal{O}(m\sqrt{n}\log^2 C \cdot \log^{\mathcal{O}(1)} m)$. Note that

$$\begin{aligned}\mathcal{O}\Big(\big(|I| + m\log(\Delta(D))\big)^{1+\gamma}\Big) &\subseteq \mathcal{O}\Big(\big((n + m\log C) \cdot \log(\Delta(D))\big)^{1+\gamma}\Big)\\ &\subseteq \mathcal{O}\Big(\big(n + m\log C\big)^{1+\gamma} \cdot \log^{\mathcal{O}(1)} m\Big),\end{aligned}$$

for $\gamma \geq 0$. Hence, if one finds an $\mathcal{O}(f(\Delta(D)) \cdot |I|^{1+\gamma})$-time algorithm for Maximum Flow with $0 \leq \gamma < 0.5$, then one would obtain an algorithm for Maximum Flow that is faster than the algorithm by Lee and Sidford [LS14] on sparse graphs, i.e., $m \in \mathcal{O}(n)$. If $0 \leq \gamma < 0.25$, then the algorithm would improve upon the algorithm by Lee and Sidford in general.

Since by applying Construction 4.8 the size of the resulting instance increases by a logarithmic factor, Proposition 4.11 gives us General Problem hardness with respect to $\Delta(D)$ only if $C \geq \Delta(D)$.

Corollary 4.12. Maximum Flow *with $C \geq \Delta(D)$ is 3-GP-hard with respect to $\Delta(D)$, where C is the highest capacity of the input and $\Delta(D)$ is the maximum degree of the input graph.*

Proof. If $C \geq \Delta(D)$, then we have $\mathcal{O}(\log(C \cdot \Delta(D))) \subseteq \mathcal{O}(\log(C^2)) \subseteq \mathcal{O}(\log C)$. Hence, by Proposition 4.11, for each instance $I = (D, s, t, c)$ with $C \geq \Delta(D)$, we can construct an equivalent instance $I' = (D', s', t', c')$ of size

$$|I'| \in \mathcal{O}(|A(D)| \cdot \log(C \cdot \Delta(D))) \subseteq \mathcal{O}(|A(D)| \cdot \log C) \subseteq \mathcal{O}(|I|),$$

where $\Delta(D') \leq 3$. This implies 3-GP hardness for all instances I' with $C > \Delta(D)$. $\square$

If instead $C < \Delta(D)$, then one cannot state GP hardness since the size of the new instance I' is required to be linear in the size of the original instance I. But on the other hand, there are as many distinct capacities as there are vertices in the instance. Thus assuming that the algorithm is run on a Random Access Machine with a word size of at least $\log n$, we can store and access capacities constant space and time. With this assumption we would obtain $|I'| \in \mathcal{O}(|I| + m\log(\Delta(D))) \subseteq \mathcal{O}(|I|)$, and thus, that the GP hardness result of Corollary 4.12 would hold in general.

4.3 High Minimum Degree Does Not Help Either

We have shown that a fixed-parameter linear-time algorithm for Maximum Flow with respect to the maximum degree would imply a faster algorithm for Maximum Flow. In other words, Maximum Flow does not become easier to solve when the maximum degree of the input graph is low. In this section we look into Maximum Flow on graphs with high minimum degree, specifically, on complete directed graphs and related graph classes.

Note first that Maximum Flow with unit capacities on complete directed graphs is trivial: There are $n-1$ arc-disjoint s–t-paths in a complete graph with n vertices, one path of length one from s to t, and $n-2$ paths of length two visiting a vertex other than s or t. Hence, the value of the maximum s–t-flow is $n-1$. But as soon as the restriction on the capacities is removed, Maximum Flow on complete directed graphs probably cannot be solved that efficiently. We show next that a linear-time algorithm for Maximum Flow on complete directed graphs would also yield an improvement on the best known running time of Maximum Flow on general graphs ($\mathcal{O}(m\sqrt{n}\log^2 C)$ [LS14]).

Theorem 4.13. *Let $\gamma \geq 0$. If there is an $\mathcal{O}(|I|^{1+\gamma})$-time algorithm for* Maximum Flow *on complete directed graphs, then there is an $\mathcal{O}((n^2 \log C)^{1+\gamma})$-time algorithm for* Maximum Flow *on general graphs.*

Proof. Let $I = (D, s, t, c)$ be an instance of Maximum Flow, where D is a general directed graph. We create an instance $I' = (D', s, t, c')$ from I as follows: The graph D' is a complete directed graph on $V(D)$, that is $A(D') = \{(u, v) \mid u \neq v \in V(D)\}$. We call an arc $a \in A(D')$ *old* if it is in $A(D)$, and *new* otherwise. For all old arcs $a \in A(D') \cap A(D)$ we set $c'(a) = c(a)$ and for all new arcs $a \in A(D') \setminus A(D)$ we set $c'(a) = 0$. The terminals s and t are copied from I. The construction of I' takes $\mathcal{O}(n^2 \log C)$ time, and the size of I' is $\mathcal{O}(n^2 \log C)$ as well.

Computing a maximum s–t-flow f' for instance I' can be done in $\mathcal{O}(|I'|^{1+\gamma}) \subseteq \mathcal{O}((n^2 \log C)^{1+\gamma})$ time since D' is complete. Clearly, the new arcs do not contribute to the value of f'. Thus we have $\mathrm{val}(f) = \mathrm{val}(f')$ I and I' are eqivalent. Therefore, it takes $\mathcal{O}((n^2 \log C)^{1+\gamma})$ time to compute f. □

Note that the running time implied by an $\mathcal{O}(|I|^{1+\gamma})$-time algorithm for Maximum Flow on complete directed graphs differs from the running time implied by an $\mathcal{O}(f(\Delta(D)) \cdot |I|^{1+\gamma})$-time algorithm in Theorem 4.7. This is an improvement over the fastest known algorithm for Maximum Flow ($\tilde{\mathcal{O}}(m\sqrt{n}\log^2 C)$ time [LS14]) whenever $m \in \omega(n^{1.5+2\gamma})$.

Clearly, Theorem 4.13 also holds if there is an $\mathcal{O}(|I|^{1+\gamma})$-time algorithm for Maximum Flow on graphs whose underlying undirected graphs are complete. One can generalize the result to all classes of (undirected) graphs that contain complete graphs, e.g., the classes of cluster graphs or interval graphs.

Corollary 4.14. *Let $\mathcal{C}$ be a class of undirected graphs such that $K_n \in \mathcal{C}$ for $n \in \mathbb{N}$, and let $\gamma \geq 0$. If* Maximum Flow *can be solved in $\mathcal{O}(|I|^{1+\gamma})$ time on graphs D for which $G(D) \in \mathcal{C}$, then* Maximum Flow *can be solved on general directed graphs in $\mathcal{O}((n^2 \log C)^{1+\gamma})$ time.*

What else might not be easy? We have shown that certain graph structures are not exploitable when solving Maximum Flow. Note though that the results of Sections 4.2 and 4.3 do not hold for Maximum Flow with unit capacities. If the maximum capacity is defined in the problem, that is, if the capacity function is $c\colon A(D) \to \{1, \dots, C\}$, then Theorem 4.13 and Corollary 4.14 still hold, but Theorem 4.7 does not. The results of Section 4.1 are independent of any capacity restrictions and thus also hold for Maximum Flow with constant and unit capcities. For the results for fixed-parameter algorithms with respect to maximum degree, it would be interesting whether these results can be extended to work with upper-bounded capacities or even unit capacities.

In their concluding remarks Kratsch and Nelles [KN18] mentioned that with the same reduction as used in Theorem 4.13 it can be shown that Maximum Flow does not become easier when parameterized by the modular width of the input graph. Without going into detail, this is easy to see since the modular width of a complete graph is constant.

Chapter 5

Fixed-Parameter Algorithms for Maximum Flow

Motivated by the gap in running times between Maximum Flow on planar graphs ($\mathcal{O}(n \log n)$, [BK09; Wei97]) and Maximum Flow on general graphs ($\mathcal{O}(nm)$, [Orl13]), Hochstein and Weihe [HW07] presented a parameterized algorithm for Maximum Flow running in $\mathcal{O}(k^3 n \log n)$ time, where k is the crossing number of the input graph. The crossing number is the minimum number k of edges that need to be removed from a (directed or undirected) graph such that it can be drawn in the plane [Zar55]. The algorithm which is named "nearly planar maxflow" by its authors is based on a generalization of the well-known *Push-Relabel method* introduced by Goldberg and Tarjan [GT88] (see Section 5.1). Herein, Hochstein and Weihe only operate on a subset of the vertices, and instead of pushing flow arc by arc towards terminal t, they push the flow along subgraphs on which Maximum Flow can be easily computed. For the implementation of the nearly planar maxflow algorithm, they split the graph into two parts, the first containing the k crossing edges, and the second containing the remaining planar graph. They choose the subset of vertices, the so-called stop vertices V^{stop}, to be the terminals s and t and the vertices incident to the crossing edges. Then, when "pushing" the flow from one stop vertex to another, one will find that either the two vertices are connected by a crossing edge, or by a subgraph of the planar graph. In the latter case, an algorithm for solving Maximum Flow on planar graphs is called as a subroutine.

Given a set of k crossing edges, the algorithm by Hochstein and Weihe [HW07] improves upon the best known running time for Maximum Flow that is independent of the maximum capacity ($\mathcal{O}(nm)$ [Orl13]) whenever $k \in \mathcal{O}(\sqrt[3]{m})$. Note at the same time that finding a minimum-cardinality set of crossing edges is NP-hard and that there are no constant-factor approximation algorithms for finding crossing edges of a graph unless $\mathsf{P} = \mathsf{NP}$ [Cab13].

Inspired by the work of Hochstein and Weihe [HW07], we present two main results in this chapter, one being a systematic approach for linear-time fixed-parameter algorithms for Maximum Flow based on the same idea as the generalized algorithm of Hochstein and Weihe, the other being an application of our approach yielding an algorithm for Maximum Flow parameterized by the feedback vertex set k of the underlying undirected input graph and running in $\mathcal{O}(k^4 m)$ time. Note that due to Bar-Yehuda et al.

[BY+98], given a graph with feedback vertex number k, one can compute an approximate feedback vertex set X of size at most $4k$ in linear time. Thus, our running time bound holds even if we are not given a feedback vertex set.

While the feedback vertex number of a graph is considered a rather large parameter, note that for the smaller parameter treewidth[1] there exists a fixed-parameter algorithm, but as opposed to our algorithm, its running time is exponential in the parameter [Hag+98]. Further, note that the crossing number and the feedback vertex number are incomparable (an $n \times n$-grid has a high feedback vertex number, but its crossing number is zero, and a complete bipartite graph $K_{3,n}$ has a high crossing number, but its feedback vertex number is three).

Chapter outline. In Section 5.1 we introduce the reader to the Push-Relabel method and state the lemmata and theorems used by Goldberg and Tarjan [GT88] to prove the correctness of the method. We then present in Section 5.2 an algorithm for Maximum Flow parameterized by the vertex cover number of the input graph as a warm-up to the systematic approach. Herein, we make use of the "proof structure" of Goldberg and Tarjan [GT88] to show the correctness and the running time of our algorithm. The systematic approach for linear-time fixed-parameter algorithms for Maximum Flow then is introduced in Section 5.3. The main theorem of Section 5.3 reuses of the theorems and lemmata proven in the previous section. Finally, in Section 5.4, we present an algorithm for Maximum Flow parameterized by the feedback vertex set as an application of our systematic approach.

5.1 Preflows and the Push-Relabel Method

In 1974, Karzanov [Kar74] took a new direction for computing Maximum Flow by introducing the concept of *preflows*. Informally, preflows have a relaxed capacity constraint compared with flows, demanding for each vertex the outgoing flow to be at most as large as the ingoing flow. The difference between incoming and outgoing flow is called the *excess* of a vertex. We state the following definitions:

Definition 5.1 (Preflow)**.** Let (D, s, t, c) be an instance of Maximum Flow. We call a function $f\colon A(D) \to \mathbb{N}_0$ a *preflow* on D, if

$$\forall a \in A(D)\colon 0 \leq f(a) \leq c(a) \qquad \text{(capacity constraint), and}$$

$$\forall v \in V(D) \setminus \{s,t\}\colon \sum_{u \in N^-(v)} f(u,v) - \sum_{w \in N^+(v)} f(v,w) \geq 0 \qquad \text{(conservation constraint).}$$

For better readability, we denote by $f(u,v)$ the flow on arc $(u,v) \in A(D)$.

The *excess* of $v \in V(D)$ with respect to preflow f is

$$\mathrm{ex}_f(v) := \sum_{u \in N^-(v)} f(u,v) - \sum_{w \in N^+(v)} f(v,w).$$

A vertex $v \in V(D) \setminus \{s,t\}$ with positive excess is called *active*.

[1] An undirected graph G with feedback vertex number k has a treewidth of at most $k+1$. For more information on treewidth we refer to Bodlaender [Bod05] and to Diestel [Die17, Chapter 12].

Note that for a flow, the excess of all vertices except s and t is zero. The definitions of the residual graph D_f and the residual capacities c_f with respect to f as stated in Definition 2.2 remain unchanged if f is a preflow.

In terms of preflow, the goal of Maximum Flow is to maximize the excess of t such that for all vertices v other than s or t, the excess is zero. The result of Karzanov [Kar74] is based on this formulation of the goal. The idea is to "push" the flow from s arc by arc towards t. Based on this, Goldberg and Tarjan [GT88] introduced the *Push-Relabel method*. Shortly put, in the Push-Relabel method one examines the active vertices other than s and t and pushes the excess towards what is "believed" to be close to t. If t is not reachable from a vertex, then the excess of the vertex cannot be pushed to t, and the algorithm pushes the excess back towards s. To determine which vertex is "believed" to be close to t, Goldberg and Tarjan introduced a labeling on the vertices.

Definition 5.2 (Distance Labeling)**.** Let (D, s, t, c) be an instance of Maximum Flow and let f be a preflow. A vertex labeling $d\colon V(D) \to \mathbb{N}_0$ is a *valid distance labeling* if

$$\begin{aligned} &d(s) = n,\ d(t) = 0, && \\ &d(v) \leq d(w) + 1, && \text{for all } (v, w) \in A(D_f). \end{aligned}$$

If for some $(v, w) \in A(D_f)$, $d(v) = d(w) + 1$, then we say that (v, w) is *admissible*.

The distance labeling allows us to approximate the distance from vertices $v \in V(G) \setminus \{s, t\}$ to t or s. We make the following observation, already indicated, but not formally proven by Goldberg and Tarjan [GT88].

Lemma 5.3. *Let $d\colon V(D) \to \mathbb{N}_0$ be a valid labeling and f be a preflow. If $d(v) < n$, then for $v \in V(D) \setminus \{s, t\}$, $d(v)$ is a lower bound on the distance from v to t in D_f. If $d(v) \geq n$, then $d(v) - n$ is a lower bound on the distance from v to s in D_f.*

Proof. Due to Reduction Rule 3.2 we may assume without loss of generality that every vertex $v \in V(D) \setminus \{s, t\}$ is reachable from s and reaches t. Let f be a preflow. We claim that if in the residual graph D_f vertex v does not reach t, then it reaches s. Towards this claim, consider a v–t-path P in D that does not exist in D_f. Then, there exists an arc $(u, w) \in A(P)$ such that u is reachable from v in D_f, and that f saturates (u, w), that is, $f(u, w) = c(u, w)$ and thus $(u, w) \notin A(D_f)$ and $(w, u) \in A(D_f)$. Since f is a preflow, we then have a path from u to s and thus a path from v to s in D_f.

Next, assume that d is a valid labeling on D_f and that that there exists a shortest v–t-path of length ℓ in D_f. Then, for each of its ℓ arcs (u, w) it holds that $d(u) \leq d(w) + 1$. By induction we can show that $d(v) \leq d(t) + \ell = \ell$, and thus $d(v)$ is a lower bound on the distance from v to t in D_f.

Clearly, any shortest path in a graph with n vertices has length at most $n - 1$. Thus if $d(v) \geq n$, then there cannot be a shortest v–t-path. Then, by the claim above, there exists an s–v-path. Assume that this path is a shortest path of length ℓ. Then, as above, for each of its ℓ arcs (u, w) we have $d(u) \leq d(w) + 1$, and thus $d(v) \leq d(s) + \ell = n + \ell$. Hence, if $d(v) \geq n$, then $d(v) - n$ is a lower bound on the distance from v to s. □

Algorithm 5.1 is an implementation of the Push-Relabel method as introduced by Goldberg and Tarjan [GT88] and uses two procedures called by a main routine. Procedure `Push`(v, w) takes an admissible arc $(v, w) \in D_f$ as input and pushes the excess of v towards w. There are two possible outcomes after a call to `Push`(v, w).

Algorithm 5.1: The Push-Relabel method by Goldberg and Tarjan [GT88].

```
Main Algorithm
    Call Initialize().
    while there is an active vertex $v$ do
        if there is a vertex $w$ such that arc $(v, w) \in A(D_f)$ is admissible then
            Call Push$(v, w)$.
        else
            Call Relabel$(v)$.
        end
    end
end
Procedure Initialize()
    Set $d(s) := n$, and set $d(v) := 0$ for all $v \neq s$.
    For each arc $(s, v) \in A(D)$ call Push$(s, v)$.
            (Assume $\text{ex}_f(s)$ to be $\infty$ during Initialize() and $-\infty$ afterwards.)
end
Procedure Push$(v, w)$
    Increase $f(v, w)$ by $\min\{c_f(v, w), \text{ex}_f(v)\}$.
end
Procedure Relabel$(v)$
    Set $d(v) := \min\{d(w) + 1 \mid (v, w) \in A(D_f)\}$.
end
```

(1) If $\text{ex}_f(v) \geq c_f(v, w)$, then the procedure pushes flow of value $c_f(v, w)$ along (v, w). After the call to `Push`(v, w), v has nonnegative excess, and arc $(v, w) \notin A(D_f)$. We call this a *saturating* push. (2) If $\text{ex}_f(v) < c_f(v, w)$, then the procedure pushes the excess of v towards w. Vertex v then has an excess of zero, and arc (v, w) still exists in D_f. We call this a *non-saturating* push. Procedure `Relabel`(v) takes an active vertex with no outgoing admissible arc as input. It increases label $d(v)$ such that there is at least one admissible arc (v, w). That is, after a call to `Relabel`(v), there is at least one arc (v, w) such that `Push`(v, w) can be called. Note that in procedure `Initialize`, we push from an inactive vertex s and along inadmissible arcs. To simplify the procedure, we assume during the initialization that $\text{ex}_f(s) = \infty$, and that all arcs leaving s are admissible. After the initialization, s acts as a sink. We thus assume that $\text{ex}_f(s) = -\infty$ after the call to `Initialize`. The algorithm terminates as soon as there is no active vertex, that is, as soon as there is no vertex other than s and t that has positive excess.

In the following we present the steps in which Goldberg and Tarjan [GT88] prove the correctness and running time of the Push-Relabel method, but we omit the proofs. We refer to Goldberg and Tarjan [GT88] and to Ahuja, Magnanti, and Orlin [AMO93, Chapter 7] for proof details.

Towards showing correctness, we first need to show that at no time d becomes invalid or there is a vertex with negative excess.

Lemma 5.4 ([GT88])**.** *Algorithm 5.1 maintains the invariant that d is a valid labeling and that f is a preflow.*

Since, in the initialization, we saturate all arcs leaving s, there are no s–t-paths in the residual graph D_f. If the algorithm terminates, then no vertex is active, and our preflow f is a flow. As t is not reachable from s, it follows that f is maximum.

Proposition 5.5 ([GT88])**.** *If Algorithm 5.1 terminates, then the computed preflow f is a maximum flow.*

Towards showing that the algorithm terminates within a given running time upper bound, Goldberg and Tarjan first observe that the sizes of the distance labels are bounded.

Lemma 5.6 ([GT88])**.** *For a vertex $v \in V(D)$, the label $d(v)$ never decreases. Each call to* `Relabel`*(v) increases $d(v)$. During the execution of Algorithm 5.1, for every vertex $v \in V(D)$, $d(v) \leq 2n - 1$.*

With this at hand, one can upper-bound the number of calls to the procedures.

Lemma 5.7 ([GT88])**.** *In Algorithm 5.1,*

(i) the number of calls to `Relabel` *is at most $2n-1$ per vertex and at most $2n^2$ overall,*
(ii) the number of calls to `Push` *resulting in a saturating push is at most $2nm$, and*
(iii) the number of calls to `Push` *resulting in a non-saturating push is at most $4n^2m$.*

Since the procedures `Push` and `Relabel` operate in constant time, we obtain the following.

Theorem 5.8 ([GT88])**.** *Algorithm 5.1 solves* Maximum Flow *in $\mathcal{O}(n^2m)$ time.*

This running time can be further improved if the active vertices are selected in a specific order. Goldberg and Tarjan [GT88] achieved a running time of $\mathcal{O}(n^3)$ by choosing the next active vertex in a first-in, first-out (FIFO) manner. Informally, by FIFO, we mean that we call procedures `Push` and `Relabel` on the vertices in the order in which they become active. Algorithm 5.2 is an implementation of the Push-Relabel method employing the FIFO method to choose the next vertex to operate on.

Shortly after the introduction of the Push-Relabel method, Cheriyan and Maheshwari [CM89] showed that by always selecting the active vertex with the highest distance label results in a running time for Maximum Flow of $\mathcal{O}(n^2m^{1/2})$.

5.2 Warm-Up: Maximum Flow with Vertex Cover Number

As a simple practical example of the systematic approach which is going to be introduced in the following section we present a parameterized algorithm for Maximum Flow with respect to the vertex cover number (introduced below). The algorithm shares many features with the parameterized algorithm with respect to the crossing number by Hochstein and Weihe [HW07]. Both algorithms are based on the Push-Relabel method

Algorithm 5.2: FIFO implementation of the Push-Relabel method [GT88].

```
1  Call Initialize().
2  Let $Q_1$ be a list containing the initially active vertices in arbitrary order.
3  Set $i := 1$.
4  while there are active vertices do
5      Set $\ell := |Q_i|$ and initialize $Q_{i+1}$ as an empty list.
6      for $j := 1, \dots, \ell$ do                    (Iterate through the elements of $Q_i$)
7          Let $v$ be the $j$-th element of $Q_i$.
8          repeat
9              if there is a vertex $w$ such that arc $(v, w) \in A(D_f)$ is admissible then
10                 Call Push$(v, w)$.
11                 if $w \notin Q_{i+1}$ then append $w$ to $Q_{i+1}$.
12             else
13                 Call Relabel$(v)$.
14             end
15         until $\text{ex}_f(v) = 0$ or $d(v)$ is increased.
16         if $v$ is active then append $v$ to $Q_{i+1}$.
17     end
18     Increase $i$ by one.
19 end
```

introduced in Section 5.1 and use a highly modified `Push` method. But this method is also the main difference of the algorithms: While the algorithm of Hochstein and Weihe calls a subroutine on the input graph minus the crossing edges, our algorithm calls a subroutine only on a small subgraph.

The algorithm presented in this section can be seen as an application of the systematic approach shown in the next section, but we prove the correctness and running time of the parameterized algorithm with respect to the vertex cover number independently of the systematic approach. Later on, when presenting the systematic approach in the next section, we make use of the proof structure employed in this section.

Let D be a directed graph together with a vertex cover $V' \subseteq V(D)$. We define a vertex cover of a directed graph as the vertex cover of the underlying undirected graph, that is, a vertex cover is a set of vertices V' such that the undirected graph $G(D) - V'$ has no edges. Observe that, given a vertex $u \in V'$ of degree at least one, either u is in the vertex cover, or every neighbor of u is in the vertex cover. This observation is the central idea of the algorithm presented in this section: If there is an u–w-path with $u, w \in V'$, then there is an u–v-path with $v \in V'$ of length at most two (it may be that $v = w$). In other words, if D is (weakly) connected, then from any vertex $u \in V'$, there is a path of length at most two to some vertex $w \in V'$.

We use this property of vertex covers for our algorithm. Instead of pushing excess along single arcs, we push excess only between vertices in the vertex cover, and s and t. In analogy to Hochstein and Weihe [HW07] we call these vertices *stop vertices*. We denote them by $V^{\text{stop}} := V' \cup \{s, t\}$, and the number of stop vertices by $k := |V^{\text{stop}}|$.

One can think of the procedure `Push`(v, w) as solving an instance of MAXIMUM FLOW

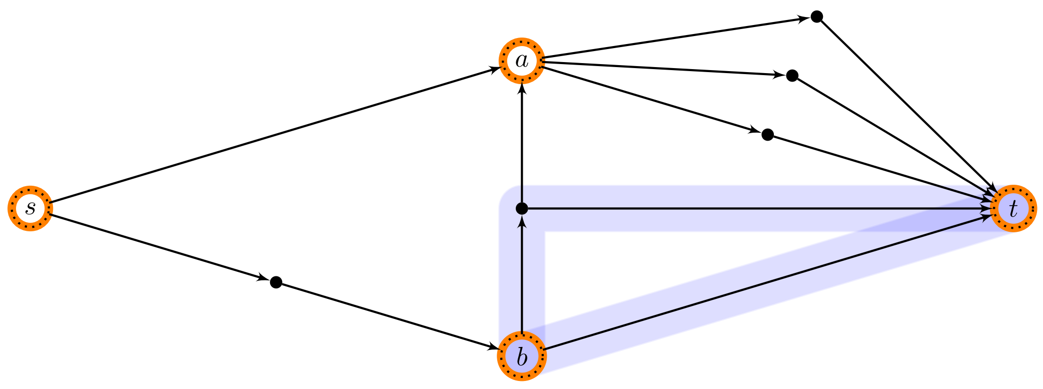


Figure 5.1: An example graph with stop vertices $V^{\text{stop}} = \{s, a, b, t\}$ (marked orange, dotted). A call to $\texttt{Push}(b, t)$ operates only on the atomic b–t-paths (marked blue), but not on any of the paths visiting a, since $a \in V^{\text{stop}}$.

on the subgraph "between" v and w, that is, on the graph consisting of all v–w-paths. To ensure that procedure Push operates efficiently we need the procedure to operate on paths that are as short as possible. Consider two vertices $v, w \in V^{\text{stop}}$. There may be arbitrarily many v–w-paths of arbitrary length. As noted above though, we can separate any of these v–w-paths into paths of length at most two in such a way that the endpoints of all paths are in V^{stop}. We call such paths *atomic*. See Figure 5.1 for an example.

Definition 5.9 (Atomic paths)**.** Let D be a graph and let $V^{\text{stop}} \subseteq V(D)$ be a vertex set. Then, an u–v-path is an *atomic path* if its endpoints are in V^{stop} and if its inner vertices are in $V(D) \setminus V^{\text{stop}}$.

By allowing $\texttt{Push}(v, w)$ to operate only on atomic v–w-paths, we can guarantee that the procedure operates in linear time. Since the labeling is only relevant for the procedure Push, it is sufficient if only the stop vertices carry labels. Note though that the definitions of a valid labeling changes: A labeling $d\colon V^{\text{stop}} \to \mathbb{N}_0$ is valid if $d(s) = k$, $d(t) = 0$, and $d(v) \leq d(w) + 1$ for every atomic v–w-path in the residual graph D_f. Note that this implies $d(v) \leq d(w) + 1$ for any v–w-path with $v, w \in V^{\text{stop}}$. We call v–w-paths *admissible* if $v, w \in V^{\text{stop}}$ and $d(v) = d(w) + 1$.

We may call $\texttt{Push}(v, w)$ if v is active and there is an atomic admissible v–w-path in D_f. Procedure $\texttt{Push}(v, w)$ then works in two steps:

(1) On the subgraph of D_f consisting of all atomic v–w-paths, compute a v–w-flow such that the value of the flow is maximized, but does not exceed the excess of v.

(2) Send the computed flow along the arcs of D_f.

The definition of saturating and non-saturating is adapted accordingly: We call a push resulting from $\texttt{Push}(v, w)$ saturating if afterwards there is no atomic v–w-path in D_f, and non-saturating otherwise.

Algorithm 5.3 gives a formal description of the algorithm described above. As with Algorithm 5.1, we assume that during Initialize, $\text{ex}_f(s) = \infty$, and that all atomic

Algorithm 5.3: The Push-Relabel method with a vertex cover as stop vertices.

```
1  Main Algorithm
       See Algorithm 5.2 (Replace "admissible arc $(v,w) \in A(D_f)$" with "admissible
        atomic $v$–$w$-path in $D_f$").
2  end
3  Procedure Initialize()
4      Set $d(s) := k$, and set $d(v) := 0$ for all $v \neq s$.
5      For each atomic $s$–$v$-path in $D_f$, call Push$(s,v)$
                   (Assume $\mathrm{ex}_f(s)$ to be $\infty$ during Initialize() and $-\infty$ afterwards.)
6  end
7  Procedure Push$(v,w)$
8      In $D_f$, find all atomic $v$–$w$-paths $\mathcal{P} = \{P_1,\dots,P_\ell\}$.
9      For $1 \le i \le \ell$, let $\delta_i := \min\{c_f(u,u') \mid (u,u') \in A(P_i)\}$.
10     Set $c_f^{\mathrm{stop}}(v,w) := \sum_{i=1}^{\ell} \delta_i$.          (maximum flow from $v$ to $w$ along $\mathcal{P}$)
11     Set $j := \min\{\mathrm{ex}_f(v), c_f^{\mathrm{stop}}(v,w)\}$, and set $i := 1$.
12     while $j > 0$ and $i \le \ell$ do                 (Send flow of $j$ along the paths in $\mathcal{P}$.)
13         Set $q := \min\{\delta_i, j\}$.
14         For all $(u,u') \in A(P_i)$ increase $f(u,u')$ by $q$.
15         Decrease $j$ by $q$ and increase $i$ by one.
16     end
17 end
18 Procedure Relabel$(v)$
19     Set $d(v) := \min\{d(w)+1 \mid \text{there is an atomic } v\text{–}w\text{-path in } D_f\}$.
20 end
```

paths starting in s are admissible. After the initialization, we assume $\mathrm{ex}_f(s) = -\infty$. Note that we use the first-in, first-out implementation of the Push-Relabel method as described in Algorithm 5.2.

We continue by proving correctness of the described algorithm. We start by showing that some properties that hold for the original procedure Push also hold for the new procedure. Afterwards we proceed similarly to the way of proving the correctness of the original Push-Relabel method. For this, we first show that at any time during the execution of the algorithm, f is a preflow and d is a valid labeling.

Lemma 5.10. *Algorithm 5.3 maintains the invariant that f is a preflow.*

Proof. Note that the only part of the algorithm to change f is the procedure Push. So we claim that if f is a valid preflow, then it remains a valid preflow after calling Push.

Let $\mathcal{P}$ be the set of all atomic v–w-paths in D_f. When calling Push(v,w), flow of value $\min\{\mathrm{ex}_f(v), c_f^{\mathrm{stop}}(v,w)\}$ is sent along the paths in $\mathcal{P}$. More specifically, for each $P \in \mathcal{P}$ and for each $a, a' \in A(P)$, we have $f(a) = f(a')$. The paths in $\mathcal{P}$ are pairwise arc-disjoint, since otherwise the graph would contain parallel arcs or loops. Thus, we have that the incoming flow of the internal vertices is the same as the outgoing flow. This gives us that the internal vertices have an excess of zero. Since all

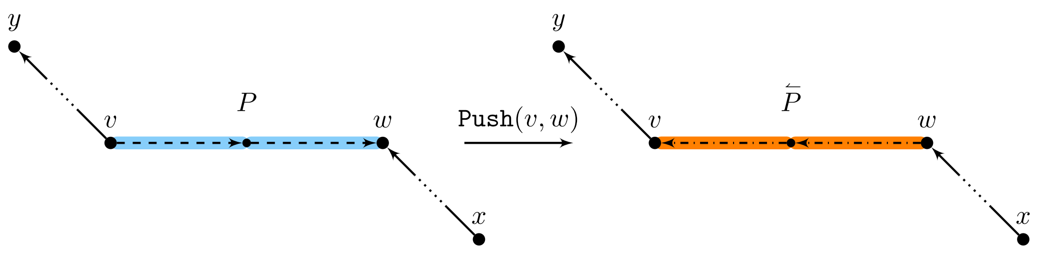


Figure 5.2: The effect of a call to Push along stop vertices. On the left-hand side, we can see that there is an atomic admissible v–w-path P (dashed, blue). If we call $\texttt{Push}(v,w)$, then we know that afterwards we have a w–v-path $\overleftarrow{P}$ (dash-dotted, orange) and hence, an x–y-path, which did not exist before.

paths in $\mathcal{P}$ start in v and end in w, the excess of v is pushed towards w, and $\mathrm{ex}_f(w)$ increases by $\min\{\mathrm{ex}_f(v), c_f^{\mathrm{stop}}(v,w)\}$. Note that after a call to $\texttt{Push}(v,w)$, the excess of v becomes zero if it is a non-saturating push, and nonnegative if it is a saturating push. Hence, f follows the relaxed conservation constraint as of Definition 5.1 after a call to Push, that is, for every vertex other than s and t, the incoming flow is at least as large as the outgoing flow. Due to the choice of δ_i the capacities of the arcs in $A(\bigcup \mathcal{P})$ are not exceeded by f. So, f also follows the capacity constraint, and thus, if f is a preflow, then it also is a preflow after calling $\texttt{Push}(v,w)$. □

For showing that d is a valid labeling at any step during the execution, we first need to prove the following claim. The proof is based on Hochstein and Weihe [HW07, Lemma 5.1].

Lemma 5.11. *Let d be a valid labeling, and let $v,w,x,y \in V^{\mathrm{stop}}$ be stop vertices such that there is an atomic admissible v–w-path P in D_f. After calling* $\texttt{Push}(v,w)$ *from Algorithm 5.3, there is no x–y-path in D_f, or $d(x) \le d(y)+1$.*

Proof. If after calling $\texttt{Push}(v,w)$ there is no x–y-path in D_f, then we are done. So assume that there is an arbitrary x–y-path Q. Calling $\texttt{Push}(v,w)$ creates a w–v-path $\overleftarrow{P}$ in D_f. If $\overleftarrow{P}$ and Q are arc-disjoint, then Q existed in D_f before the call to $\texttt{Push}(v,w)$. Since at that moment d was a valid labeling, we have $d(x) \le d(y)+1$, and we are done.

So assume that $\overleftarrow{P}$ and Q are not arc-disjoint. Let x' and y' be the vertices incident to the shared arcs $A(\overleftarrow{P}) \cap A(Q)$ that are closest to x and y respectively. Before the call to $\texttt{Push}(v,w)$, there was an x–x'-path in D_f. Since P existed at that time, there was an x'–w-path, and hence, an x–w-path as well. Since the labeling was valid at that point, we have $d(x) \le d(w)+1$. Analogously, there were paths from v to y' and from y' to y, yielding a v–y-path in D_f. Thus $d(v) \le d(y)+1$. See Figure 5.2 for an example. Since the v–w-path was admissible at that time, we have $d(v) = d(w)+1$. We obtain

$$d(x) \le d(w)+1 = d(v) \le d(y)+1.$$

This inequality also holds if $x = w$ or $v = y$. □

With Lemma 5.11 at hand, we can show that d is a valid labeling throughout the runtime of Algorithm 5.3.

Lemma 5.12. *Algorithm 5.3 maintains the invariant that d is a valid labeling.*

Proof. After the initialization there is no arc leaving s in D_f, and thus $d(s) = k$ is correct. For $v \in V^{\text{stop}} \setminus \{s, t\}$, label $d(v) = 0$ also fulfills the criteria of a valid labeling. Hence, after the initialization, d is valid.

Let $W \subseteq V^{\text{stop}}$ be the set of vertices such that for each $w \in W$ there is an atomic v–w-path. Then calling `Relabel`(v) increases $d(v)$ such that $d(v) \leq d(w) + 1$ for all $w \in W$. Given that d is valid, `Relabel`(v) yields a new valid labeling.

Lemma 5.11 gives us that after a call to `Push`(v, w) on a valid labeling, for each pair $x, y \in V^{\text{stop}}$ of stop vertices, there either is an x–y-path in D_f and $d(x) \leq d(y) + 1$, or there is no x–y-path. This gives us that `Push` maintains the invariant that d is a valid labeling. □

Having shown that labeling d remains valid during the execution of Algorithm 5.3, we are now ready to show that the algorithm is correct.

Proposition 5.13. *If Algorithm 5.3 terminates, then f is a maximum flow.*

Proof. By Lemmata 5.10 and 5.12, the algorithm maintains the invariants that f is a preflow and d is a valid labeling. The algorithm terminates if every vertex except s and t has an excess of zero. Thus, at termination, f is a flow. Assume towards contradiction that after termination there is an s–t-path P of length ℓ in D_f. Note that a path may visit every vertex (and thus every stop vertex) at most once. Thus there exist $j < k$ atomic paths $P_1, \dots, P_j$ such that $P = P_1 \cup \dots \cup P_j$, where P_1 is an atomic s–v_1-path, P_j is an atomic v_{j-1}–t-path, and P_i is an atomic v_{i-1}–v_i-path, for $1 < i < j$. Since by Lemma 5.12 the validity of d is maintained, we have $d(v) \leq d(w) + 1$ for each of the atomic paths P_i, and thus $d(s) \leq d(t) + j < k$ as $d(t) = 0$. This contradicts $d(s) = k$ and thus there is no s–t-path after the algorithm has terminated. Hence, at termination, f is a maximum flow. □

The next step is to show an upper bound for the running time of the algorithm. We do this by showing that the labels cannot increase too much, and by this, the number of calls to `Push` and `Relabel` are bounded.

First, we show that the label of a vertex cannot get smaller during the execution of the algorithm.

Lemma 5.14. *For a vertex $v \in V^{\text{stop}}$, the label $d(v)$ never decreases, and each call to `Relabel`(v) of Algorithm 5.3 increases $d(v)$.*

Proof. Since the labels are only updated by `Relabel`, we only need to show that $d(v)$ is increased by a call to `Relabel`(v). Let $W \subseteq V^{\text{stop}}$ be the set of vertices such that for each $w \in W$ there is an atomic v–w-path. Procedure `Relabel`(v) is only called if in D_f there is no admissible atomic path starting in v. Then we have $d(v) < d(w) + 1$ for all $w \in W$. Setting $d(v)$ to $\min\{d(w) + 1 \mid w \in W\}$ increases the value of $d(v)$. □

Having shown that the labels do not decrease at any time, we need to show that the labels cannot grow too large. Recall that $k = |\, V^{\text{stop}} \,|$.

Lemma 5.15. *During the execution of Algorithm 5.1, for every $v \in V^{\text{stop}}$, $d(v) \leq 2k-1$.*

Proof. This trivially holds for s and t. Further, the algorithm only changes labels of active vertices. Thus it suffices to show that the lemma holds for any active vertex $v \in V^{\text{stop}} \setminus \{s,t\}$. Let f be a preflow computed during the runtime of Algorithm 5.3. Let $v \in V^{\text{stop}}$ be a vertex with $\text{ex}_f(v) > 0$. Then there must be an s–v-path P in D such that for all $a \in A(P)$, $f(a) > 0$. Hence, in D_f, we have a v–s-path $\overleftarrow{P}$. The number ℓ of stop vertices visited by $\overleftarrow{P}$ excluding the start point v is at most $k-1$, since otherwise $\overleftarrow{P}$ would visit some vertex twice. Since d is valid, we have $d(u) \leq d(u') + 1$ for each atomic u–u'-path in $\overleftarrow{P}$. Hence, $d(v) \leq d(s) + \ell \leq k + k - 1 = 2k - 1$. □

After the initialization, $d(v) = 0$ for each $v \in V^{\text{stop}} \setminus \{s\}$, and $d(t)$ is never increased. Hence, it follows directly from Lemmata 5.14 and 5.15 that the number of calls to `Relabel` is bounded.

Lemma 5.16. *The number of calls to* `Relabel` *is at most* $2k - 1$ *per vertex and at most* $2k^2$ *overall.*

Next, we show that the number of calls to `Push` is bounded. Here, we need to differentiate between saturating and non-saturating pushes. Recall that we call the push resulting from `Push`(v, w) saturating if afterwards there is no atomic v–w-path in D_f, and non-saturating otherwise.

Lemma 5.17. *The number of calls to* `Push` *resulting in a saturating push is at most* $4k^3$.

Proof. We claim that for each atomic v–w-path, between two calls to `Push`(v, w) that result in a saturating push, there must be a call to `Relabel`(v). Recall that in order to call `Push`(v, w), we require that there is an atomic v–w-path, and that this path is admissible, that is,

$$d(v) = d(w) + 1. \tag{5.1}$$

Recall further that an atomic v–w-path has length at most two. If we have a saturating push along the path, then at least one of the two arcs becomes reversed by the push operation.

First, consider an atomic v–w-path of length one in D_f. That is, the path consists only of the arc (v, w). After a saturating push along (v, w), we only have the reverse arc (w, v) in D_f. Hence, to be able to push along (v, w) again, we need to restore the arc. This can only be achieved by calling `Push`(w, v). For this, the label of w must increase by two. Hence, to push along (v, w) again, the label of v must increase by two as well, since otherwise (v, w) is not admissible. Thus, for atomic v–w-paths of length one, there cannot be two calls to `Push`(v, w) without a call to `Relabel`(v) in between.

Now, consider an atomic v–w-path P of length two in D_f. Let P consist of the arcs (v, u) and (u, w). After a saturating push from v to w, at least one of the two arcs of P is saturated, that is, one of the two arcs is replaced by its reverse. Hence, P can be restored by pushing back from w to v. This case is analogous to the case above with paths of length one. However, it is also possible to restore only the saturated arc.

Consider first that (v, u) is saturated, that is, we have arc (u, v), but not (v, u) in D_f. Then, to restore arc (v, u), we require a push along an atomic path that contains arc (u, v). Suppose there is an atomic x–v-path Q in D_f, starting in $x \in V^{\text{stop}}$ and visiting u. Then, by calling `Push`(x, v), we can restore arc (v, u) in D_f. Such a path Q

exists only if arc (x, u) existed prior to the first push along P. Hence, before the first call to $\texttt{Push}(v, w)$ there was an atomic x–w-path. Since d is a valid labeling at all times, we have

$$d(x) \leq d(w) + 1. \tag{5.2}$$

To call $\texttt{Push}(x, v)$, we require Q to be an admissible atomic path. For Q to be admissible, we must have

$$d(x) = d(v) + 1 = d(w) + 2 > d(w) + 1,$$

which contradicts inequality (5.2). The second equality is due to equation (5.1). The contradiction gives us that Q can only be admissible if $\texttt{Relabel}(w)$ is called. Hence, in order to call $\texttt{Push}(v, w)$ again, v must be relabeled as well.

Consider now that (u, w) is saturated, that is, we only have arc (w, u) in D_f. To restore arc (u, w), we need to push along an atomic path that contains arc (w, u). Suppose there is an atomic w–x-path Q in D_f, ending in $x \in V^{\text{stop}}$ and visiting u. By calling $\texttt{Push}(w, x)$, we can restore arc (u, w) in D_f. We again have that arc (u, x) must have existed prior to the first push along P, hence, we have an atomic v–x-path, and since d is a valid labeling,

$$d(v) \leq d(x) + 1. \tag{5.3}$$

To push along Q, we require that Q is admissible, that is,

$$d(w) = d(x) + 1 \geq d(v) \Rightarrow d(v) < d(w) + 1,$$

which contradicts equation (5.1). Thus Q can only be admissible if $\texttt{Relabel}(w)$ is called. As above, this rules out another call to $\texttt{Push}(v, w)$ without calling $\texttt{Relabel}(v)$ first.

Concludingly, in all cases, between any two saturating pushes along a v–w-path, there must be a call to $\texttt{Relabel}(v)$. Since there are at most $2k$ calls to $\texttt{Relabel}(v)$, and there are at most k^2 atomic paths in D, we have that there are at most $4k^3$ calls to $\texttt{Push}$ resulting in a saturating push. □

Next we need to upper-bound the number of non-saturating pushes. Recall that Algorithm 5.3 employs the FIFO method to choose the next active vertex to operate on (see Algorithm 5.2). To bound the number of non-saturating pushes, we first need to discuss the number of iterations over the while-loop in line 4 of Algorithm 5.2.

Lemma 5.18. *The number of iterations over the while-loop in line 4 of Algorithm 5.2 is at most $4k^2$.*

Proof. Let $\Phi_i := \max(\{d(v) \mid v \in Q_i\} \cup \{0\})$, $i \geq 1$, be the highest label of all vertices in Q_i at the moment at which the algorithm starts iterating over Q_i (consider Q_i to be computed after line 5). First note that, by Lemma 5.6, $\Phi_i \leq 2k - 1$. Observe further that in iteration i, we iterate over the vertices in list Q_i.

There are two ways for a vertex $v \in V^{\text{stop}}$ to be appended to Q_{i+1}. The first way is if v became active during iterating over Q_i due to a call to $\texttt{Push}(u, v)$ from some $u \in Q_i$. Then, $d(v) < d(u) \leq \Phi_i$, where d is the labeling at the moment of the call to $\texttt{Push}(v, w)$. The other way is if $v \in Q_i$, and v remained active during iterating over Q_i. Then, there must have been a call to $\texttt{Relabel}(v)$, and $d(v)$ must have increased.

Suppose that during iterating over list Q_i there was at least one call to `Relabel`. Denote by $Q_i^R \subseteq Q_i$ the vertices that were relabeled. Let $d(v)$ be the label before the call to `Relabel`(v), and let $d'(v)$ be the label afterwards. Let $\varphi_i := \max\{d'(v) - d(v) \mid v \in Q_i^R\}$ be the highest increase of a label during iterating over Q_i. Hence, $\Phi_{i+1} \leq \Phi_i + \varphi_i$.

Suppose now that during iterating over Q_i there was no call to `Relabel`; thus there were only calls to `Push`. Then, Q_{i+1} contains all vertices to which excess was pushed. Thus, due to the admissibility constraint, for each $w \in Q_{i+1}$ we have $d(w) < \Phi_i$, and hence, $\Phi_{i+1} < \Phi_i$.

Since overall we have at most $2k^2$ calls to `Relabel`, we know that there are at most $2k^2$ iterations i such that $\Phi_i > \Phi_{i-1}$. Suppose there are j iterations over the while-loop. Then we have that $\Phi_1 = \Phi_{j+1} = 0$, since after the initialization all active vertices have label zero, and at termination there are no active vertices. Thus there are at most $2k^2$ iterations i such that $\Phi_i < \Phi_{i+1}$. Altogether, there are at most $4k^2$ and hence, at most $4k^2$ iterations over the while-loop. □

With this at hand, it is easy to see that the number of non-saturating pushes can be upper-bounded.

Lemma 5.19. *The number of calls to* `Push` *resulting in a non-saturating push is at most* $4k^3$.

Proof. After a non-saturating push from v to w, the excess of v is zero. Since in a list each vertex can appear at most once, there is at most one non-saturating push per vertex per list, which is $4k^2 \cdot k = 4k^3$. □

With the running times and the number of calls to the procedures at hand, we can state the running time of Algorithm 5.3.

Theorem 5.20. *Algorithm 5.3 solves* Maximum Flow *in* $\mathcal{O}(\tau^3 \cdot m)$ *time, where* τ *is the vertex cover number of* D.

Proof. For executing `Relabel`(v), we need to find all vertices $w \in V^{\text{stop}}$ such that there is an atomic v–w-path in D_f. We can do this by computing the first two layers of a breadth-first search from v. Computing the new value of $d(v)$ can be done on the fly during the breadth-first search. A call to `Relabel` thus takes $\mathcal{O}(m)$ time.

We can use the same principle to bound the time of a call to `Push`(v, w). Again, to find all v–w-paths, we need to compute the first two layers of a breadth-first search. Since there are at most $\mathcal{O}(m)$ v–w-paths and $\mathcal{O}(m)$ arcs of which the flow needs to be updated, the while-loop in procedure `Push` takes $\mathcal{O}(m)$ time to compute. Together with the breadth-first search, this gives us a running time of $\mathcal{O}(m)$.

Lastly, for each list Q_i, for each vertex $v \in Q_i$, we need to find all admissible atomic paths starting in a given vertex v. This can be done by computing the first two layers of a breadth-first search as well and again takes $\mathcal{O}(m)$ time.

Overall, by Lemmata 5.16 to 5.19, we know that there are at most $\mathcal{O}(k^2)$ calls to `Relabel`, at most $\mathcal{O}(k^3)$ calls to `Push` and at most $\mathcal{O}(k^2)$ lists, each of which may contain $\mathcal{O}(k)$ vertices, where $k := |V^{\text{stop}}|$ is the number of stop vertices. We have shown above that for each of these calls or vertices in lists, we need to do computations taking $\mathcal{O}(m)$ time. Hence, the overall running time is $\mathcal{O}(k^3 \cdot m)$.

Let τ be the vertex cover number of D. Since V^{stop} is a vertex cover plus the vertices s and t, we have $k \leq \tau + 2$. Thus the running time of Algorithm 5.3 can be bounded by $\mathcal{O}(\tau^3 \cdot m)$. $\square$

Comparing the algorithm with the algorithm by Hochstein and Weihe [HW07] which runs in $\mathcal{O}(k^3 n \log n)$ time, where k is the crossing number, one can see that the running times of the two algorithms are similar. The two main differences are the following: The first difference is, as mentioned before, that the procedure `Push` of our algorithm runs only on a small portion of the graph while the procedure `Push` of the algorithm by Hochstein and Weihe runs an instance of Maximum Flow on planar graphs on a subgraph containing all vertices of the input graph. Hence, it is likely that in practice the procedure `Push` of our algorithm outperforms the procedure `Push` of Hochstein and Weihe. The second difference is the parameter. While the parameters vertex cover number and crossing number are incomparable (an $n \times n$-grid has a high vertex cover number, but its crossing number is zero, and a complete bipartite graph $K_{3,n}$ has a high crossing number, but its vertex cover number is three), in practice, the crossing number tends to be smaller than the vertex cover number.

In the upcoming section we are going to make use of the proof structure employed for proving the running time and the correctness of Algorithm 5.3.

5.3 A Systematic Approach

In the previous section we presented a linear-time fixed-parameter algorithm for Maximum Flow with respect to the vertex cover number of the input graph. Our goal is to generalize the used approach to develop algorithms for Maximum Flow with respect to stronger parameters. Herein we focus on vertex deletion distances to special graph classes. In order to maintain the efficiency of the algorithm of our systematic approach as well as any algorithms derived from this approach we require that the procedure `Push` operates in linear time and that the number of stop vertices $|V^{\text{stop}}|$ is bounded by some parameter.

We now generalize the approach of the parameterized algorithm with respect to the vertex cover number. Observe that in procedure `Push`(v, w) of Algorithm 5.3 we need to compute the maximum v–w-flow on the subgraph consisting of all atomic v–w-paths in D_f, that is, an independent set plus two vertices, namely v and w. Our generalization is of such a kind that we exchange the graph class of independent sets by other graph classes Π. Analogously we adapt V^{stop} to be the deletion set to Π plus s and t, that is, we set $V^{\text{stop}} := V' \cup \{s, t\}$, where $V' \subseteq V(D)$ is the smallest set such that $D - V' \in \Pi$. In procedure `Push`(v, w) we then compute the maximum v–w-flow on the subgraph D' consisting of the vertices of all atomic v–w-paths in D_f. Note that the definition of atomic paths is dependent on the set V^{stop}. Further, note that D' consists of a subgraph $D'' \in \Pi$ together with the vertices $v, w \in V^{\text{stop}}$. So in order to fulfill that `Push` operates in linear time, we require the class Π to be of such kind that that Maximum Flow on a graph in Π plus two vertices can be solved efficiently. If Π fulfills this property, then we say it is *nifty*.

Before we formally define Π, we make a few observations that impact on how to design Π. Since we call `Push` as a subroutine, and since the orientation of the arcs in D_f

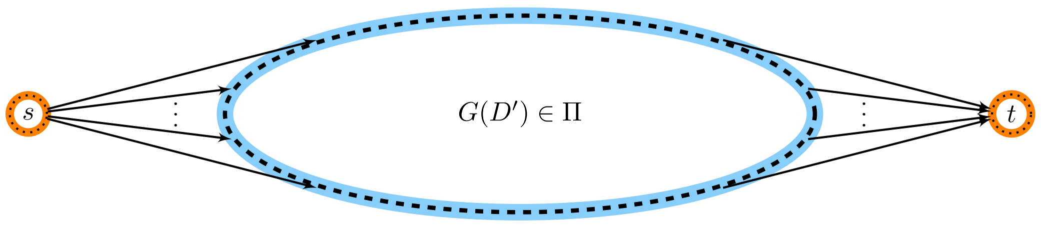


Figure 5.3: Let Π be a class of graphs that is *f-nifty*. If the subgraph $G(D') = G(D - \{s, t\})$ (blue, dashed) is in Π, then Maximum Flow can be solved in $f(|I|)$ time on D.

depends on (pre-)flow f, we require that a call to `Push` runs in linear time independently of the orientation of the arcs. Hence, we consider classes Π of undirected graphs. Further, since for a call to `Push` we only consider a subgraph of D_f we require Π to be hereditary. Having made these two observations, we call Π *nifty* if one can compute a maximum s–t-flow on a graph consisting of an arbitrary orientation of some $G \in \Pi$ plus arbitrarily connected vertices s and t efficiently. See Figure 5.3 for further intuition. Formally, we state the following definition.

Definition 5.21. Let Π be a hereditary class of undirected graphs and let $f \colon \mathbb{N} \to \mathbb{N}$ be a function. We call Π *f-nifty* if it holds that for every instance $I = (D, s, t, c)$ of Maximum Flow, if $G(D - \{s, t\}) \in \Pi$, then I is solvable in $f(|I|)$ time. If f is linear, then we say Π is nifty.

Since a vertex cover $V' \subseteq V(D)$ is the same as the vertex deletion set to an independent set, we obtain the following as an immediate product of Theorem 5.20.

Observation 5.22. *The class* Π *of independent sets is nifty.*

From the results of Chapter 4 we can also derive a few graph classes that are unlikely to be nifty, e.g., classes of bipartite graphs (Corollary 4.6), of bounded degree (Theorem 4.7) and classes of graphs that contain complete graphs (Corollary 4.14).

We now show the main result of this section, namely, that there exists a linear-time fixed-parameter algorithm for Maximum Flow with respect to the vertex deletion distance to Π, if Π is nifty.

Theorem 5.23. *Let f be a function, let Π be an f-nifty graph class and let $I = (D, s, t, c)$ be an instance of* Maximum Flow*, and let $V' \in V(D)$ be a vertex set of size k such that $G(D - V') \in \Pi$. Then instance I can be solved in $k^4 f(|I|)$ time.*

Proof. We first state the algorithm—hereafter called *general algorithm*—and afterwards prove its correctness and running time. Let $V^{\text{stop}} = V' \cup \{s, t\}$. Recall that a path is atomic if its endpoints are in V^{stop} and its inner vertices are in $V(D) \setminus V^{\text{stop}}$. For the general algorithm we reuse the main routine as shown in Algorithm 5.2 (replace "admissible arc (v, w)" with "admissible atomic v–w-path") and the procedures `Initialize` and `Relabel` from Algorithm 5.3, but state a new generalized procedure `Push`. All procedures and the main routine used are gathered in Algorithm 5.4.

Algorithm 5.4: The algorithm for Maximum Flow with parameter vertex deletion distance to Π.

```
Main Algorithm                                    (Restated from Algorithm 5.2)
    Call Initialize().
    Let $Q_1$ be a list containing the initially active vertices in arbitrary order.
    Set $i := 1$.
    while there are active vertices do
        Set $\ell := |Q_i|$ and initialize $Q_{i+1}$ as an empty list.
        for $j := 1, \dots, \ell$ do
            Let $v$ be the $j$-th element of $Q_i$.
            repeat
                if there is a vertex $w$ such that there is an atomic $v$–$w$-path in $D_f$
                 then
                    Call Push$(v, w)$.
                    if $w \notin Q_{i+1}$ then append $w$ to $Q_{i+1}$.
                else
                    Call Relabel$(v)$.
                end
            until $\mathrm{ex}_f(v) = 0$ or $d(v)$ is increased.
            if $v$ is active then append $v$ to $Q_{i+1}$.
        end
        Increase $i$ by one.
    end
end
Procedure Initialize()                            (Restated from Algorithm 5.3)
    Set $d(s) := k$, and set $d(v) := 0$ for all $v \neq s$.
    For each atomic $s$–$v$-path in $D_f$, call Push$(s, v)$
              (Assume $\mathrm{ex}_f(s)$ to be $\infty$ during Initialize() and $-\infty$ afterwards.)
end
Procedure Push$(v, w)$         (A generalization of procedure Push of Algorithm 5.3)
    In $D_f$, find all atomic $v$–$w$-paths $\mathcal{P} = \{P_1, \dots, P_\ell\}$.
    Compute $v$–$w$-flow $f_{v,w}$ on $\bigcup \mathcal{P}$ such that $\mathrm{val}(f)$ is maximized, but does not
     exceed $ex_f(v)$.
    Set $f := f + f_{v,w}$.
end
Procedure Relabel$(v)$                            (Restated from Algorithm 5.3)
    Set $d(v) := \min\{d(w) + 1 \mid \text{there is an atomic } v\text{–}w\text{-path in } D_f\}$.
end
```

Towards showing the correctness of the general algorithm we first show that procedure Push in Algorithm 5.4 maintains the invariant that f is a preflow. Consider a call to Push(v, w), $v, w \in V^{\mathrm{stop}}$, and let $\mathcal{P}$ be the set of all atomic v–w-paths. Let f be the preflow before, $f_{v,w}$ be the flow computed during, and $f' = f + f_{v,w}$ be the preflow after the call to `Push`(v, w). Assume that f is a valid preflow and that $f_{v,w}$ is a valid v–w-flow in graph $\bigcup\mathcal{P} \subseteq D_f$ of maximal value such that $\operatorname{val}(f_{v,w}) \leq \operatorname{ex}_f(v)$. Since $\operatorname{val}(f_{v,w}) \leq \operatorname{ex}_f(v)$, preflow f' still fulfills the relaxed conservation constraint (see Definition 5.1) after a call to `Push`. Further, since $f_{v,w}$ is a valid flow on $\bigcup\mathcal{P} \subseteq D_f$, we have $f_{v,w}(a) \leq c_f(a)$ for every arc $a \in A(\bigcup\mathcal{P})$. By Definition 2.2 it then follows that f' also fulfills the capacity constraint, and thus f' is a valid preflow.

Having shown that procedure `Push` in Algorithm 5.4 maintains a valid preflow, by careful observation one can find that Lemmata 5.11 and 5.12 and Proposition 5.13 also hold for the general algorithm. Hence, the general algorithm is correct.

We now show that the general algorithm runs in $\mathcal{O}(k^4 m)$ time. Since the general algorithm uses the same procedure `Relabel` as Algorithm 5.3, Lemmata 5.14 to 5.16 trivially hold. Towards upper-bounding the number of calls to `Push`, note that Lemmata 5.18 and 5.19 hold as well for the general algorithm. Thus we have at most $4k^2$ iterations over the while-loop in line 5 of Algorithm 5.4. Recall that in each of these iterations, we iterate over a list of active vertices. Note that there are at most $|V^{\text{stop}}| \leq k+2$ active vertices. Let $v \in V^{\text{stop}}$ and let $W \subseteq V^{\text{stop}}$ be the set of vertices such that for each $w \in W$ there is an admissible atomic v–w-path. After a call to `Push`(v, w) resulting in a saturating push, neither $\operatorname{ex}_f(v) = 0$ nor $d(v)$ increased. Hence there are at most $(k+2) \cdot (k+2) \cdot (4k^2) \in \mathcal{O}(k^4)$ calls to `Push` resulting in a saturating push.

Towards determining the running time of procedure `Push`(v, w) of the general algorithm observe that $G(\bigcup\mathcal{P} - \{v, w\}) \subseteq G(D_f - V^{\text{stop}}) \in \Pi$, and since $G(\bigcup\mathcal{P}) \subseteq G(D_f) \subseteq G(D)$ and Π is nifty, flow $f_{v,w}$ can be computed in $\mathcal{O}(f(I|))$ time. Finding all atomic v–w-paths in D_f and increasing f by $f_{v,w}$ can be done in $\mathcal{O}(m)$ time by a breadth-first search starting at v and by iterating over the arcs of D_f, respectively. Thus procedure `Push`(v, w) operates in $f(|I|)$ time. As shown in the proof of Theorem 5.20, procedure `Relabel` operates in $\mathcal{O}(m)$ time, and for each iteration over the while-loop in line 5 of Algorithm 5.4, for each active vertex v in the list we require $\mathcal{O}(m)$ time to find all atomic admissible paths starting in v.

Overall, we have at most $\mathcal{O}(k^2)$ calls to `Relabel`, at most $\mathcal{O}(k^4)$ calls to `Push` and at most $\mathcal{O}(k^2)$ iterations over the while-loop in line 4 of Algorithm 5.2, in each of which we may iterate over $\mathcal{O}(k)$ vertices. Hence, the running time of the general algorithm is $k^4 f(|I|)$. □

The bottleneck in the running time of the general algorithm is the bound on the number of saturating pushes. As one can see we did not manage to bound the number of saturating pushes by $\mathcal{O}(k^3)$ as in Lemma 5.17 of the previous section. We leave it as an open question to whether the running time bound of the general algorithm, and thus, of our systematic approach, can be improved to $k^3 f(|I|)$.

5.4 Maximum Flow With Feedback Vertex Number

Having stated our systematic approach, we now present a parameterized algorithm for Maximum Flow with respect to the undirected feedback vertex number of the input graph based on our approach. Since a feedback vertex set is the same as a vertex deletion set to a forest, we only need to show that forests are nifty. In our main theorem we make use of the data reduction rules introduced in Chapter 3, so we restate them here.

Reduction Rule 3.2 (BFS-Rule; Restated)**.** *Let $I = (D, s, t, c)$ be an instance of* MAXIMUM FLOW *and let $v \in V(D)$ be a vertex such that there exists no s–v-path or no v–t-path. Then delete v.*

Reduction Rule 3.3 (Cut-Rule; Restated)**.** *Let $I = (D, s, t, c)$ be an instance of* MAXIMUM FLOW *where the BFS-Rule is not applicable, and let $w \in V(D)$ be a cut vertex. Let $v \in V(D) \setminus \{s, t\}$ be a vertex such that all s–v-paths and all v–t-paths visit w. Then delete v.*

Reduction Rule 3.4 (Deg-2-Rule; Restated)**.** *Let $I = (D, s, t, c)$ be an instance of* MAXIMUM FLOW *and let $v \in V(D) \setminus \{s, t\}$ be a vertex of combined degree two, and let u and w be the neighbors of v. If there exists a u–w-path, then ensure that $(u, w) \in A(D)$ (if necessary, add (u, w) with capacity zero), and increase $c(u, w)$ by $\min\{c(u, v), c(v, w)\}$. Proceed analogously if there exists a w–u-path. Then remove v from D.*

Reduction Rule 3.5 (SVT-Rule; Restated)**.** *Let $I = (D, s, t, c, k)$ be an instance of k-*FLOW *and let $v \in V(D) \setminus \{s, t\}$ such that there is a path $P = svt$ in D. Then decrease k by $\min\{c(s, v), c(v, t)\}$, and decrease $c(s, v)$ and $c(v, t)$ by the same value. If arc $a \in A(P)$ has capacity zero, delete it.*

We can now state the main theorem of this section.

Theorem 5.24. *Forests are nifty.*

Proof. Let $I = (D, s, t, c)$ be an instance of MAXIMUM FLOW such that $G := G(D - \{s, t\})$ is a forest. Since every s–t-path in D can only visit vertices of at most one of the components of G, we assume without loss of generality that G is a tree. We prove first that by applying Reduction Rules 3.2 to 3.5 exhaustively we can reduce I to a constant-size kernel, second, that we can apply the reduction rules in $\mathcal{O}(n_G)$ time, and third, that we then can compute a maximum s–t-flow f for I in $\mathcal{O}(n_G)$ time.

We first apply Reduction Rules 3.2 and 3.4 to remove sources and sinks as well as all degree-two vertices $v \in V(G)$ whose neighbors u and w are in the tree. Clearly, arcs (u, w) and (w, u) cannot exist, since otherwise G would contain a cycle $uvwu$, contradicting the assumption that it is a tree. Thus, we can introduce arcs (u, w) and (w, u) in constant time. Next, by iterating through the neighborhoods of s and t we remember for each vertex $v \in V(G)$ whether the arc (s, v) or the arc (v, t) exists. This allows us to check in constant time whether arcs (s, v) or (v, t) exist.

The next step is to show that we can delete a leaf v of tree G with a constant number of applications of Reduction Rules 3.2 to 3.5, requiring constant time overall. Let u be the unique neighbor of v in G. Observe that we have at least one of the arcs (u, v) and (v, u), and that there may exist arcs (s, v) and (v, t) in D. If neither arc (s, v) nor arc (v, t) exists, then v is connected to the rest of the graph only via u and by Reduction Rule 3.3 we can delete v. If one of the two arcs exists, then it is adjacent to two vertices. Assume that arc (s, v) exists. If arc (v, u) exists, then we can apply Reduction Rule 3.4. Since we can check in constant time whether arc (s, u) exists, the application of the reduction rule on v takes constant time. If arc (v, u) does not exist, then v is a sink and can be removed by Reduction Rule 3.2. The case that arc (v, t) exists can be approached analogously. Lastly, if v is connected to both s and t, then we have a path $P = svt$, and by Reduction Rule 3.5 we can delete at least one of the arcs (s, v) and (v, t). Then, v is adjacent to one or two vertices, so one of the previous case applies and v can be deleted. Since the degree of v is at most three, it is easy to see that removing the vertex by taking the steps above can be done in constant time.

Trees can be eliminated by consequently deleting leaves, hence, applying the four reduction rules exhaustively reduces instance I to an instance I' that consists of only the vertices s and t which may or may not be connected by an arc (s, t). Instance I' can be solved trivially. Towards computing f, we keep a list of actions $A_1, \dots A_\ell$, each representing an application of a reduction rule, to remember in which order we applied which reduction rule on which vertex. For each action we remember

(1) the vertex v that was removed and the capacities of the arcs incident to v, if we apply Reduction Rule 3.2 or 3.3,

(2) the vertex v that was removed, its incident arcs to neighbors u and w and their capacities, and the capacities $c(u, w)$ and $c(w, u)$, if the corresponding arcs existed, if we apply Reduction Rule 3.4, and

(3) the removed arc and its capacity, if we apply Reduction Rule 3.5.

Let $I_0 = I$, and let instance I_j be the instance that we obtain from applying action A_j to instance I_{j-1}, for $1 \leq j \leq \ell$. Thus $I_\ell = I'$. Now, given an instance $I_j = (D_j, s, t, c_j)$, a corresponding solution f_j and action A_j, we can reconstruct the previous instance $I_{j-1} = (D_{j-1}, s, t, c_{j-1})$ and compute a solution f_{j-1} as follows:

(1) If A_j is an application of Reduction Rule 3.2, then add the remembered vertex v and its incident arcs with their capacities. Set the flow f_{j-1} of those arcs to zero.

(2) If A_j is an application of Reduction Rule 3.4, then add the remembered vertex v and its incident arcs to D. Set the capacities c_{j-1} of the arcs between the vertices u, v and w to the remembered values. If $f_j(u, w) \leq c_{j-1}(u, w)$, then set flow $f_{j-1}(u, w) = f_j(u, w)$ and set $f_{j-1}(u, v) = f_{j-1}(v, w) = 0$. If $f_j(u, w) > c_{j-1}(u, w)$, then set $f_{j-1}(u, w) = c_{j-1}(u, w)$ and set $f_{j-1}(u, v) = f_{j-1}(v, w) = f_j(u, w) - f_{j-1}(u, w)$. Proceed analogously with the arcs (w, u), (w, v), and (v, u) in the opposite direction, if existing.

(3) If A_j is an application of Reduction Rule 3.5, then assume that the arc removed by A_j is (s, v). Add arc (s, v) with the remembered capacity $c(s, v)$, set $c_{j-1}(v, t) = c_j(v, t) = c(s, v)$, set $f_{j-1}(s, v) = c(s, v)$ and set $f_{j-1}(v, t) = f_j(v, t) + c(s, v)$. If arc (v, t) is removed by A_j, then proceed analogously.

The restored instance then is equal to the original instance since we only introduce tree leaves in cases (1) and (2), and we add arcs to tree leaves in case (3).

Next, we need to show that given instance I_j and action A_j for $1 \leq j \leq \ell$ and the corresponding maximum flow f_j, the computed flow f_{j-1} for I_{j-1} is a maximum flow. To this end, observe that in cases (1) and (2) we do not add any s–t-paths, and that $\operatorname{val}(f_{j-1}) = \operatorname{val}(f_j)$. In case (3) we introduce a saturated arc incident to either s or t. So, while $\operatorname{val}(f_{j-1}) \geq \operatorname{val}(f_j)$, we do not obtain any additional s–t-paths in D_f. Thus in all three cases the computed flow f_{j-1} is a maximum flow for I_{j-1}. Since instance I_ℓ

can be trivially solved, the flow f_0 computed in the last step is a maximum flow for instance $I_0 = I$.

Finally, observe that for every action $A_1, \ldots, A_\ell$ reconstructing the previous instance takes constant time. The exhaustive application of Reduction Rules 3.2 and 3.4 on tree vertices takes linear time, a tree elimination order is found in linear time, and, as shown above, each leaf of tree G is eliminated by applying a constant number of data reduction rules, and thus actions. This gives us that we require $\mathcal{O}(n_G)$ time overall to solve instances I as described above, and concludes the proof. □

Using Theorems 5.23 and 5.24 and the linear-time constant-factor approximation for finding an undirected feedback vertex set by Bar-Yehuda et al. [BY+98], we obtain a parameterized algorithm for Maximum Flow with respect to the feedback vertex number. Since we only need $\mathcal{O}(n_G)$ time to solve Maximum Flow on forests plus two vertices, we can improve the running time upper bound for a call to `Push` in the generalized algorithm from $\mathcal{O}(|I|)$ to $\mathcal{O}(m)$.

Corollary 5.25. Maximum Flow *can be solved in $\mathcal{O}(k^4 m)$ time, where k is the undirected feedback vertex number of the input graph.*

Are other graph classes nifty? Apart from forests we also investigated into whether other graph classes may be nifty. As already mentioned, it is unlikely that the class bipartite graphs, the class of graphs of bounded degree or any class of graphs that contain complete graphs are nifty (see Corollaries 4.6 and 4.14 and Theorem 4.7). Recall that Maximum Flow on planar graphs can be solved in $\mathcal{O}(n \log n)$ time [BK09; Wei97]. This does not imply though that planar graphs would become f-nifty for a quasilinear function f. This is easy to see since one cannot add two arbitrarily connected vertices to a planar graph while maintaining the planarity of the graph. The same also holds for outerplanar graphs: Adding one (arbitrarily connected) vertex to an outerplanar graph maintains planarity, but adding the second vertex may break the planarity of the graph.

Note that grid graphs are planar and thus Maximum Flow on grid graphs can be solved in $\mathcal{O}(n \log n)$ time. Hence, it would be interesting to know whether grid graphs are f-nifty for a quasilinear f, or whether it is unlikely for then to be f-nifty.

Chapter 6

Conclusion

This work provides the first systematic study on parameterized algorithms for Maximum Flow in spirit of the FPT in P program [GMN17]. On the one hand it turns out that the restrictions of several graph parameters, i.e., any parameter that is upper-bounded by maximum degree or distance to cliques, do not help to solve Maximum Flow more efficiently. That is, a fixed-parameter algorithm cannot be more efficient than an algorithm for the general problem. However, we also provide positive results: Maximum Flow admits a quasilinear-time linear-size kernelization when parameterized by the feedback edge number of the underlying undirected graph, being—to the best of our knowledge—the first nontrivial kernelization for the problem. Further, we introduced a systematic approach for designing fixed-parameter algorithms for Maximum Flow. This is based on the idea that the operation `Push` of the Push-Relabel method [GT88] can be generalized to push along subgraphs instead of arcs. Applications of this approach yield fixed-parameter algorithms with respect to the vertex cover number τ of the underlying undirected input graph running in $\mathcal{O}(\tau^3 m)$ time, and with respect to the feedback vertex number k of the underlying undirected graph running in $\mathcal{O}(k^4 m)$ time. While there exists a (conceptually different) algorithm for Maximum Flow with respect to the smaller parameter treewidth [Hag+98], its running time is $2^{\mathcal{O}(\mathrm{tw}^2)}n$ and thus exponential in the parameter.

Comparing the outcome of this work with the results for fixed-parameter algorithms for Maximum Matching [Fom+17; IOO17; KN18; MNN17; Yus13] it seems that Maximum Flow is slightly harder to solve than Maximum Matching. Both problems admit a linear kernel with respect to the feedback edge number, and for both problems there are fixed-parameter algorithms with respect to the vertex cover number and the feedback vertex number. But the algorithms for Maximum Matching with respect to the latter two parameters run in $\mathcal{O}(k(n+m))$ time [MNN17], while the running times of the corresponding algorithms for Maximum Flow depend polynomially on the parameter.

Further research opportunities. Throughout this work we already stated a few open questions gathered here:

– Can we show that the parameter maximum degree does not help to solve Maximum Flow more quickly, even if the maximum capacity of the input is upper-bounded by a constant?

– Can we improve the upper bound on the running time of our systematic approach for fixed-parameter algorithms (Theorem 5.23) from $k^4 f(|I|)$ to $k^3 f(|I|)$ in order to match the running time of our algorithm with respect to the vertex cover number?
– Are grid graphs f-nifty for quasilinear functions f, that is, can Maximum Flow on grid graphs plus two arbitrarily connected terminals be solved in quasilinear time (see Section 5.3 for a definition of f-nifty graph classes)? This would yield a third application for our systematic approach with f being at most quasilinear.

We conclude this work with further research directions.

Treewidth and other width parameters. While there exists a fixed-parameter algorithm for Maximum Flow with respect to treewidth [Hag+98], it requires time exponential in the parameter. Fomin et al. [Fom+17] state in their work fully polynomial fixed-parameter algorithms with respect to treewidth for unit-capacitated Maximum Vertex Flow and for Maximum Matching, and state as an open question whether one can find a fully polynomial fixed-parameter algorithm for unit-capacitated Maximum Flow with respect to treewidth. We supplement this question by also asking for fully polynomial fixed-parameter algorithms for Maximum Flow with (or without) unit capacities with respect to the parameters pathwidth or bandwidth, which upper-bound the treewidth.

Kratsch and Nelles [KN18] presented a fixed-parameter algorithm for Maximum Matching with respect to modular width, and stated in their concluding remarks that Maximum Flow parameterized by modular width is not easier than general Maximum Flow. But this does not necessarily hold for Maximum Flow with unit capacities. Since the modular width of a graph can be computed in linear time it would be interesting to see whether one can find a fixed-parameter algorithm for Maximum Flow with unit capacities with respect to modular width.

Implementation. In this work we provide two fixed-parameter algorithms for Maximum Flow. Investigating whether they prove efficient also in practice and comparing them with the best known practical algorithms for Maximum Flow on real-world instances would be an interesting next step. Also, it would be interesting to see how our fixed-parameter algorithms practically compete with the algorithm parameterized by the crossing number due to Hochstein and Weihe [HW07].

Possibly even more promising for running time improvements in practice are the four data reduction rules introduced in Chapter 3. We state a few questions with respect to the practicality of our rules: How far can the instance size of real-world instances be reduced by our reduction rules? How long does an exhaustive application of the four reduction rules take in practice? How big is the time improvement of running the best known practical Maximum Flow algorithms on instances reduced by our rules versus running the algorithms on the non-reduced instances?

Further kernelization results and data reduction rules. Liers and Pardella [LP11] presented some data reduction rules that work well in practice but have a worst-case running time of $\mathcal{O}(n^5)$. Is it possible to simplify these rules towards a running time improvement, or even find further data reduction rules for Maximum Flow? Ideally these rules would yield further kernelizations for the problem.

We have shown that the four data reduction rules can be applied exhaustively together in $\mathcal{O}(nm)$ time (Theorem 3.20), and by carefully analyzing the dependencies of the rules, we have seen that applying the rules in quasilinear time turns out to be challenging; see Section 3.3. Similarly, with Weihe's Rule (Reduction Rule 3.1 [Wei97]), it remains open how one can apply the rule in quasilinear time. Clearly, an algorithm that applies these rules exhaustively in quasilinear time would yield a highly efficient preprocessing for Maximum Flow; but we would be surprised if such an algorithm existed. If one further does not succeed in finding such an algorithm, can one state conditional running time lower bounds à la Vassilevska Williams [VW15] for the exhaustive application of our four reduction rules, or of Weihe's Rule? Such a lower bound result would be of special interest given the lack of lower bound results for the Maximum Flow problem.

Literature

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993 (cit. on pp. 9, 10, 52).

[AS05] A. Atamtürk and M. W. Savelsbergh. “Integer-Programming Software Systems”. *Annals of Operations Research* 140.1 (2005), pp. 67–124 (cit. on p. 23).

[AWW16] A. Abboud, V. V. Williams, and J. Wang. “Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs”. *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’16)*. SIAM. 2016, pp. 377–391 (cit. on p. 11).

[Bag+12] M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres. “Fast Exact Computation of Betweenness Centrality in Social Networks”. *Proceedings of the 4th IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM ’12)*. IEEE Computer Society. 2012, pp. 450–456 (cit. on p. 23).

[BBN14] N. Betzler, R. Bredereck, and R. Niedermeier. “Theoretical and Empirical Evaluation of Data Reduction for Exact Kemeny Rank Aggregation”. *Autonomous Agents and Multi-Agent Systems* 28.5 (2014), pp. 721–748 (cit. on p. 23).

[Ben+17] M. Bentert, T. Fluschnik, A. Nichterlein, and R. Niedermeier. “Parameterized Aspects of Triangle Enumeration”. *Proceedings of the 21st International Symposium on Fundamentals of Computation Theory (FCT ’17)*. Springer. 2017, pp. 96–110 (cit. on pp. 11, 13, 23, 37, 38).

[BK04] Y. Boykov and V. Kolmogorov. “An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision”. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.9 (2004), pp. 1124–1137 (cit. on p. 26).

[BK09] G. Borradaile and P. Klein. “An $\mathcal{O}(n \log n)$ Algorithm for Maximum *st*-Flow in a Directed Planar Graph”. *Journal of the ACM (JACM)* 56.2 (2009), pp. 524–533 (cit. on pp. 49, 68).

[BN18] M. Bentert and A. Nichterlein. “Parameterized Complexity of Diameter”. *arXiv preprint arXiv:1802.10048* (2018) (cit. on p. 11).

[Bod05] H. L. Bodlaender. “Discovering Treewidth”. *Proceedings of the 31st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM ’05).* Springer. 2005, pp. 1–16 (cit. on p. 50).

[Bru+12] S. Bruckner, F. Hüffner, C. Komusiewicz, R. Niedermeier, S. Thiel, and J. Uhlmann. “Partitioning Into Colorful Components by Minimum Edge Deletions”. *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM ’12).* Springer. 2012, pp. 56–69 (cit. on p. 23).

[BY+98] R. Bar-Yehuda, D. Geiger, J. Naor, and R. M. Roth. “Approximation Algorithms for the Feedback Vertex Set Problem With Applications to Constraint Satisfaction and Bayesian Inference”. *SIAM Journal on Computing* 27.4 (1998), pp. 942–959 (cit. on pp. 13, 49, 68).

[Cab13] S. Cabello. “Hardness of Approximation for Crossing Number”. *Discrete & Computational Geometry* 49.2 (2013), pp. 348–358 (cit. on pp. 13, 49).

[Car+16] M. L. Carmosino, J. Gao, R. Impagliazzo, I. Mihajlin, R. Paturi, and S. Schneider. “Nondeterministic Extensions of the Strong Exponential Time Hypothesis and Consequences for Non-Reducibility”. *Proceedings of the 7th ACM Conference on Innovations in Theoretical Computer Science (ITCS ’16).* ACM. 2016, pp. 261–270 (cit. on p. 37).

[CEN12] E. W. Chambers, J. Erickson, and A. Nayyeri. “Homology Flows, Cohomology Cuts”. *SIAM Journal on Computing* 41.6 (2012), pp. 1605–1634 (cit. on pp. 11, 12, 14).

[Chr+11] P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S.-H. Teng. “Electrical Flows, Laplacian Systems, and Faster Approximation of Maximum Flow in Undirected Graphs”. *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC ’11).* ACM. 2011, pp. 273–282 (cit. on p. 9).

[CM89] J. Cheriyan and S. Maheshwari. “Analysis of Preflow Push Algorithms for Maximum Network Flow”. *SIAM Journal on Computing* 18.6 (1989), pp. 1057–1086 (cit. on p. 53).

[Die17] R. Diestel. *Graph Theory.* Springer Publishing Company, Incorporated, 2017 (cit. on pp. 15, 50).

[Din70] E. A. Dinic. “Algorithm for Solution of a Problem of Maximum Flows in Networks with Power Estimation”. *Soviet Mathematics Doklady* 11 (1970), pp. 1277–1280 (cit. on p. 11).

[Din73] E. A. Dinic. “Bitwise Redidual Decreasing Method and Transportation Type Problems (in Russian)”. *Studies in Discrete Mathematics* (1973), pp. 46–57 (cit. on p. 11).

[EK72] J. Edmonds and R. M. Karp. “Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems”. *Journal of the ACM)* 19.2 (1972), pp. 248–264 (cit. on p. 11).

[Eve11] S. Even. *Graph Algorithms.* Cambridge University Press, 2011 (cit. on pp. 28, 30).

[FF56] L. R. Ford and D. R. Fulkerson. "Maximal Flow Through a Network". *Canadian Journal of Mathematics* 8.3 (1956), pp. 399–404 (cit. on pp. 9, 11, 20, 22, 24, 27).

[FHW80] S. Fortune, J. Hopcroft, and J. Wyllie. "The Directed Subgraph Homeomorphism Problem". *Theoretical Computer Science* 10.2 (1980), pp. 111–121 (cit. on p. 24).

[Flu+17] T. Fluschnik, C. Komusiewicz, G. B. Mertzios, A. Nichterlein, R. Niedermeier, and N. Talmon. "When can Graph Hyperbolicity be Computed in Linear Time?" *Proceedings of the 15th Algorithms and Data Structures Symposium (WADS '17)*. Springer. 2017, pp. 397–408 (cit. on pp. 11, 23).

[Fom+17] F. V. Fomin, D. Lokshtanov, M. Pilipczuk, S. Saurabh, and M. Wrochna. "Fully Polynomial-Time Parameterized Computations for Graphs and Matrices of Low Treewidth". *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '17)*. Society for Industrial and Applied Mathematics. 2017, pp. 1419–1432 (cit. on pp. 11, 69, 70).

[Gab85] H. N. Gabow. "Scaling Algorithms for Network Problems". *Journal of Computer and System Sciences* 31.2 (1985), pp. 148–168 (cit. on p. 11).

[GG11] K. Genova and V. Guliashki. "Linear Integer Programming Methods and Approaches—A Survey". *Journal of Cybernetics and Information Technologies* 11.1 (2011) (cit. on p. 23).

[GMN17] A. C. Giannopoulou, G. B. Mertzios, and R. Niedermeier. "Polynomial Fixed-Parameter Algorithms: A Case Study for Longest Path on Interval Graphs". *Theoretical Computer Science* 689 (2017), pp. 67–95 (cit. on pp. 9, 11, 69).

[GR98] A. V. Goldberg and S. Rao. "Beyond the Flow Decomposition Barrier". *Journal of the ACM* 45.5 (Sept. 1998), pp. 783–797 (cit. on pp. 10, 11).

[GT14] A. V. Goldberg and R. E. Tarjan. "Efficient Maximum Flow Algorithms". *Communications of the ACM* 57.8 (2014), pp. 82–89 (cit. on p. 10).

[GT88] A. V. Goldberg and R. E. Tarjan. "A New Approach to the Maximum-Flow Problem". *Journal of the ACM* 35.4 (1988), pp. 921–940 (cit. on pp. 11–13, 49–54, 69).

[Hag+98] T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde. "Characterizing Multiterminal Flow Networks and Computing Flows in Networks of Small Treewidth". *Journal of Computer and System Sciences* 57.3 (1998), pp. 366–375 (cit. on pp. 12–14, 50, 69, 70).

[Hal+15] K. G. Hales, C. A. Korey, A. M. Larracuente, and D. M. Roberts. "Genetics on the fly: a primer on the Drosophila model system". *Genetics* 201.3 (2015), pp. 815–842 (cit. on p. 11).

[HL11] M. Hilbert and P. López. "The World's Technological Capacity to Store, Communicate, and Compute Information". *Science* 332.6025 (2011), pp. 60–65 (cit. on p. 9).

[HW07] J. M. Hochstein and K. Weihe. “Maximum s–t–Flow with k Crossings in $\mathcal{O}(k^3 n \log n)$ Time”. *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’07)*. Society for Industrial and Applied Mathematics. 2007, pp. 843–847 (cit. on pp. 12, 13, 37, 49, 53, 54, 57, 62, 70).

[IOO17] Y. Iwata, T. Ogasawara, and N. Ohsaka. “On the Power of Tree-Depth for Fully Polynomial FPT Algorithms”. *arXiv:1710.04376* (2017) (cit. on pp. 11, 69).

[IPZ01] R. Impagliazzo, R. Paturi, and F. Zane. “Which Problems Have Strongly Exponential Complexity?” *Journal of Computer and System Sciences* 63.4 (2001), pp. 512–530 (cit. on pp. 11, 37).

[Kar74] A. V. Karzanov. “Determining the Maximal Flow in a Network by the Method of Preflows”. *Soviet Mathematics Doklady* 15.2 (1974), pp. 434–437 (cit. on pp. 11, 50, 51).

[Kel+14] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford. “An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations”. *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’14)*. SIAM. 2014, pp. 217–226 (cit. on p. 9).

[KN18] S. Kratsch and F. Nelles. “Efficient and Adaptive Parameterized Algorithms on Modular Decompositions”. *arXiv preprint arXiv:1804.10173* (2018) (cit. on pp. 11, 47, 69, 70).

[KRT94] V. King, S. Rao, and R. Tarjan. “A Faster Deterministic Maximum Flow Algorithm”. *Journal of Algorithms* 17.3 (1994), pp. 447–474 (cit. on pp. 10, 11).

[KT18] R. Krauthgamer and O. Trabelsi. “Conditional Lower Bounds for All-Pairs Max-Flow”. *arXiv preprint arXiv:1702.05805* (2018) (cit. on pp. 10, 11).

[LP11] F. Liers and G. Pardella. “Simplifying Maximum Flow Computations: The Effect of Shrinking and Good Initial Flows”. *Discrete Applied Mathematics* 159.17 (2011), pp. 2187–2203 (cit. on pp. 12, 23, 70).

[LRS13] Y. T. Lee, S. Rao, and N. Srivastava. “A New Approach to Computing Maximum Flows Using Electrical Flows”. *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC ’13)*. ACM. 2013, pp. 755–764 (cit. on p. 9).

[LS14] Y. T. Lee and A. Sidford. “Path Finding Methods for Linear Programming: Solving Linear Programs in $\tilde{\mathcal{O}}(\sqrt{\text{rank}})$ Iterations and Faster Algorithms for Maximum Flow”. *Proceedings of the 55th Annual Symposium on Foundations of Computer Science (FOCS ’14)*. IEEE. 2014, pp. 424–433 (cit. on pp. 10, 11, 32, 36, 37, 40, 45, 46).

[MNN17] G. B. Mertzios, A. Nichterlein, and R. Niedermeier. “The Power of Linear-Time Data Reduction for Maximum Matching”. *Proceedings of the 42nd International Symposium on Mathematical Foundations of Computer Science (MFCS ’17)*. 2017, pp. 46:1–46:14 (cit. on pp. 11, 23, 69).

[Nie06] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006 (cit. on p. 11).

[Orl13] J. B. Orlin. “Max Flows in $\mathcal{O}(nm)$ Time, or Better”. *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC ’13).* 2013, pp. 765–774 (cit. on pp. 10, 11, 36, 37, 49).

[Sav94] M. W. P. Savelsbergh. “Preprocessing and Probing Techniques for Mixed Integer Programming Problems”. *ORSA Journal on Computing* 6.4 (1994), pp. 445–454 (cit. on p. 23).

[She13] J. Sherman. “Nearly Maximum Flows in Nearly Linear Time”. *Proceedings of the 54th Annual Symposium on Foundations of Computer Science (FOCS ’13).* IEEE. 2013, pp. 263–269 (cit. on p. 9).

[SW16] M. Sorge and M. Weller. “The Graph Parameter Hierarchy”. Unpublished Manuscript. 2016 (cit. on p. 13).

[VW15] V. Vassilevska Williams. “Hardness of Easy Problems: Basing Hardness on Popular Conjectures Such as the Strong Exponential Time Hypothesis”. *Proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC ’15).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2015, pp. 16–29 (cit. on pp. 11, 37, 71).

[Wei97] K. Weihe. “Maximum (s, t)-Flows in Planar Networks in $\mathcal{O}(|V| \log |V|)$ Time”. *Journal of Computer and System Sciences* 54.3 (1997), pp. 454–475 (cit. on pp. 12, 23–26, 37, 38, 41, 49, 68, 71).

[Yus13] R. Yuster. “Maximum Matching in Regular and Almost Regular Graphs”. *Algorithmica* 66.1 (2013), pp. 87–92 (cit. on pp. 11, 69).

[Zar55] C. Zarankiewicz. “On a Problem of P. Turán Concerning Graphs”. *Fundamenta Mathematicae* 41.1 (1955), pp. 137–145 (cit. on p. 49).