Karlsruhe Institute of Technology

Entwicklung eines Exakten Algorithmus für das Directed Feedback Vertex Set Problem

Bachelorarbeit
von

Ruben Götz

An der Fakultät für Informatik
Institute of Theoretical Informatics

Erstgutachter:	T.T.-Prof. Dr. Thomas Bläsius
Zweitgutachter:	PD Dr. Torsten Ueckerdt
Betreuende Mitarbeiter:	Christopher Weyand

Bearbeitungszeit: 1. März 2022 – 1. Juli 2022

Selbstständigkeitserklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 21. August 2022

Zusammenfassung

In dieser Arbeit wurde ein Algorithmus zum Lösen des *Directed Feedback Vertex Set* Problems entwickelt. Der Algorithmus folgt einem *Branch&Bound* Ansatz. Den Hauptbestandteil dieser Arbeit bildet die Reduktion von Probleminstanzen. Hierfür werden 10 Reduktionsregeln vorgestellt. Auch die Implementierung des Algorithmus wird in dieser Arbeit diskutiert. Der hier behandelte Algorithmus wurde in der Programmiersprache C++ implementiert und steht öffentlich zur Verfügung. Die Testergebnisse der Implementierung wurden in dieser Arbeit zusammengefasst.

Inhaltsverzeichnis

1. Einführung

Das *(Directed) Feedback Vertex Set* (DFVS) Problem besteht daraus, für einen (gerichteten) Graphen G eine Knotenmenge S zu finden, für die gilt: Werden S sowie alle an einen Knoten aus S angrenzenden Kanten aus G entfernt, ist der übrige Graph azyklisch. Das *Feedback Vertex Set* Problem ist für ungerichtete, wie auch gerichtete Graphen, NP-vollständig [Coo71], [Kar72]. Ein optimales *Feedback Vertex Set* zeichnet sich dadurch aus, dass kein *Feedback Vertex Set* existiert, welches weniger Knoten beinhaltet.

Das *Feedback Vertex Set* Problem findet unter anderem Anwendung im Bereich der *deadlock recovery*, beispielsweise in Betriebssystemen [SGG06] oder Datenbanksystemen [GS76] sowie für die Entwicklung und Testung von Schaltkreisen [CA90],[CBA95],[SW75]. Auch zum Lösen von *Constraint Satisfaction Problems* sowie im Bereich der *bayesian inference* findet das *Feedback Vertex Set* Problem Anwendung [BYGNR98].

Ziel dieser Arbeit ist es, einen Algorithmus zu entwickeln, durch dessen Anwendung ein optimales *Directed Feedback Vertex Set* gefunden werden kann. Der Algorithmus wurde in C++ implementiert und bei der PACE-Callenge 2022 in der Kategorie für exakte Lösungen eingereicht [pac22]. Die Implementierung ist *open source* und steht auf GitLab [Gö22a] sowie zendo.org [Gö22b] zur Verfügung und ist nicht parallelisiert. Der Algorithmus verfolgt dabei einen direkten Ansatz, berechnet also ein DFVS, ohne das Problem auf ein anderes abzubilden. Ähnliche Ansätze finden sich z.B. in den Arbeiten [SW75], [CBA95] und [LJ00]. Alternative Herangehensweisen sind beispielsweise das Abbilden eines Graphen auf eine Bool'sche Funktion, deren Lösung einem minimalen *Feedback Vertex Set* entspricht [AM94], das Lösen des Problems mit Hilfe relationaler Algebra [BF06], sowie das Lösen des komplementären Problems: Finden einer maximalen Knotenmenge, die einen azyklischen Teilgraphen induziert [Raz07]. Letztere findet die derzeit beste bekannte Laufzeit zum Lösen des *Directed Feedback Vertex Set* Problems mit $O(1,9977^n)$.
Wie erstmals von [CLL$^+$08] gezeigt, ist das DFVS Problem *fixed-parameter tractable* (FPT). Seither wurden parametrisierte Algorithmen weiter erforscht: [FWY09], [KP14] sowie [LRS16], um nur einige Arbeiten zu nennen.

In den folgenden Abschnitten werden zunächst einige nützliche Definitionen festgelegt (Abschnitt 2). Anschließend wird der entwickelte Algorithmus aus theoretischer Sicht erläutert (Anschnitt 3) sowie dessen Implementierung diskutiert (Abschnitt 4). Abschnitt

5 fasst experimentelle Ergebnisse zusammen. Zuletzt werden in Abschnitt 6 die Ergebnisse der Arbeit zusammengefasst.

2. Präliminarien

Im Folgenden werden die nötigen Grundlagen der Arbeit gelegt. Abschnitt 2.1 führt die hierfür notwendige Notation sowie wichtige Begriffe ein. Abschnitt 2.2 stellt Algorithmen vor, die in dieser Arbeit an einigen Stellen Teilprobleme lösen.

2.1. Definitionen

Im Folgenden bezeichnet der Begriff Graph ein Paar $G = (V, E)$ aus einer Knotenmenge V und einer Kantenmenge E. $V(G)$ bezeichnet die Knotenmenge des Graphen G, analog dazu $E(G)$ die Kantenmenge von G. Eine Kante ist ein Paar von Knoten. In einem gerichteten Graphen ist die Richtung einer Kante relevant. Für jede Kante (u, v) gilt also: u ist der Vorgänger von v und v der Nachfolger von u. In einem ungerichteten Graphen wird diese Unterscheidung nicht gemacht, die Paare (u, v) und (v, u) können also die selbe Kante beschreiben. Existiert eine Kante (u, v), so heißen u und v Nachbarn. Für diese Arbeit sind vor allem gerichtete Graphen interessant. Im Folgenden werden also Graphen, sofern nicht anders spezifiziert, als gerichtet angenommen.
Eine Kante (v, v) wird als Schleife bezeichnet. Existieren zwei Kanten (u, v) und (i, j), für die gilt $u = i$ und $v = j$, so werden diese als Mehrfachkante bezeichnet. Für eine Kante (u, v) wird die Kante (v, u) als Rückkante bezeichnet, falls diese existiert. Ein Pfad ist eine Folge von Knoten $(v_0, ..., v_n)$, für die gilt $(v_i, v_{i+1}) \in E$ für $i \in \mathbb{Z}_{n+1}$. Ein Kreis ist ein Pfad, für den gilt $v_0 = v_n$. Ein einfacher Kreis ist ein Kreis dessen Knoten sich paarweise unterscheiden. Zwei Kreise heißen knotendisjunkt, wenn kein Knoten in beiden Kreisen enthalten ist. Ein Kreis k heißt minimal, wenn kein Kreis k' existiert, für den gilt $k' \neq k$ und alle Knoten aus k auch in k' enthalten sind. Gilt für zwei Knoten u, v dass v in jedem Kreis enthalten ist, in dem u enthalten ist so wird u von v dominiert. Ein Knoten $v \in V(G)$ bricht einen Kreis k, wenn das Entfernen von v sowie von allen an v angrenzenden Kanten aus G dazu führt, dass k nicht mehr existiert.
Ein ungerichteter Graph heißt Baum, wenn er kreisfrei und zusammenhängend ist, also einen Pfad zwischen beliebigen verschiedenen Knoten existiert. Ein gerichteter Graph heißt Baum, wenn sein ungerichtetes Gegenstück ein Baum ist und ein Knoten existiert, von dem aus alle anderen Knoten erreicht werden können.
Zur besseren Lesbarkeit wird in dieser Arbeit DFVS an Stelle von *Directed Feedback Vertex Set* verwendet. Eine Instanz I bezeichnet eine Instanz des DFVS-Problems und wird synonym zu gerichteter Graph verwendet. Für einen Knoten v bezeichnet $I \setminus v$ die Instanz I, aus welcher der Knoten v sowie alle an v angrenzenden Kanten entfernt wurden.

Für eine Kante e bezeichnet $I \setminus e$ die Instanz I, aus welcher die Kante e entfernt wurde. Einen Knoten v in I zu ignorieren bedeutet, für jeden Pfad (i, v, j) die Kante (i, j) in I einzufügen und v sowie alle an v angrenzenden Kanten aus I zu entfernen. Eine Instanz $I' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$ heißt eine Teilinstanz von $I = (V, E)$.
Eine starke Zusammenhangskomponente bezeichnet eine maximale Teilinstanz, in der für jedes Knotenpaar u, v mit $u \neq v$ ein Pfad $(u, \ldots, v)$ existiert.
$\Pi(I) \subseteq I$ sei eine Instanz für die gilt $E(\Pi(I)) = \{(u, v) \in E(I) | (v, u) \in E(I) \wedge v \neq u\}$, $V(\Pi(I)) = \{v \in V(I) | (v, u) \in E(\Pi(I))\}$. $\Pi(I)$ ist also eine Teilinstanz von I, die genau die Kanten aus I enthält, für welche eine Rückkante in I existiert. Eine Kante $e \in \Pi(I)$ wird als Π-Kante bezeichnet. $\Pi(I)$ enthält nur die Knoten, die an eine Π-Kante angrenzen. Ein Knoten $v \in V(\Pi(I))$ wird als Π-Knoten bezeichnet. Aus $\Pi(I)$ kann eine ungerichtete Instanz I' konstruiert werden. Hierfür muss nur jedes Kantenpaar (u, v) und (v, u) durch eine ungerichtete Kante ersetzt werden. $\Pi(I)$ kann also als ungerichteter Graph interpretiert werden.
Auf ungerichteten Graphen bezeichnet eine Clique eine Knotenmenge, in der jeder Knoten zu allen anderen Knoten der Clique benachbart ist. Ein Matching auf einem ungerichteten Graphen bezeichnet eine Menge an Kanten, in der keine zwei Kanten an den selben Knoten angrenzen.

2.2. Grundlegende Algorithmen

2.2.1. Ford-Fulkerson

Der Ford-Fulkerson Algorithmus [FF56] findet in einem Flussgraphen einen maximalen Fluss zwischen dem Startknoten s und dem Zielknoten t, indem iterativ verstärkende Pfade gefunden und Residualgraphen aufgebaut werden (vgl. Algorithmus 2.1).
Ein Flussgraph bezeichnet hier einen gerichteten Graphen G, welcher zwei Knoten s und t beinhaltet. Der Knoten s besitzt keine eingehenden Kanten, t keine ausgehenden. Jede Kante $e \in E(G)$ besitzt eine Kapazität $c(e) > 0$.
Ein Fluss auf einem Flussgraphen ist eine Funktion $f : E(G) \mapsto \mathbb{N}$ und muss dabei folgende Bedingungen erfüllen:

1. Für alle Knoten $v \in V(G) \setminus \{s, t\}$ ist die Summe aller Kantenflüsse über die eingehenden Kanten gleich der Summe aller ausgehenden Kantenflüsse.
2. $f(e) \leq c(e)$

Die Größe eines Flusses bezeichnet die Summe der Kantenflüsse der in t eingehenden Kanten. Ein Fluss heißt maximal, wenn kein größerer Fluss in G existiert.

Eine Flussdekomposition teilt einen Fluss in einem Flussgraphen G in eine Menge (gewichteter) Pfade von s nach t auf. Ist die Kapazität jeder Kante 1, so entspricht der maximale Fluss in G hier also der maximalen Anzahl an kantendisjunkten Pfaden.

Der Residualgraph R von G bezüglich eines Flusses f beschreibt G mit der Möglichkeit, einen Fluss zurück zu führen. Genauer ist R ein Flussgraph mit dem Fluss f und den Kapazitäten c'. Für jede Kante $(u, v) \in V(G)$ mit Kapazität $c(v)$ beinhaltet R die Kanten (u, v) mit $c'((u, v)) = c((u, v)) - f((u, v))$ sowie (v, u) mit $c((v, u)) = f((u, v))$. Ist ein Flussgraph G mit einem Fluss f gegeben, so bezeichnet ein verstärkender Pfad einen Pfad p von s nach t im Residualgraphen von G, für den gilt: Für alle Kanten e auf p gilt $(f(e) < c(e))$.

Der Ford-Fulkerson Algorithmus findet einen maximalen Fluss in $O(F|E|)$ für den maximalen Flusswert F. Werden immer kürzeste Pfade verstärkt, wird die Laufzeit des Algorithmus

Algorithm 2.1: FORD-FULKERSON

Input: Graph G, s, t

```
1 flow ← 0
2 R ← G
3 while augmenting Path p = (s, ..., t) ∈ G exists do
4 |   b ←bottleneck capacity of path
5 |   increase flow on p by b
6 |   flow ← flow + b
7 |   update Residual Graph R
8 RETURN flow
```

Algorithm 2.2: TARJAN

Input: Graph G

```
1 vertex_count ← 0
2 scc_count ← 0
3 forall v ∈ V(I) do
4 |   ids[v] ← −1
5 |   on_stack[v] ← false
6 |   low[v] ← 0
7 forall v ∈ V(I) do
8 |   if ids[v] = −1 then
9 |   |   DFS(v)
10 RETURN(scc)
```

durch $O(|V||E|^2)$ begrenzt [EK72]. Ein Ford-Fulkerson Algorithmus, der verstärkende Pfade mit Breitensuche findet, wird daher Edmonds-Karp Algorithmus genannt.

2.2.2. Tarjans Algorithmus

Tarjans Algorithmus [Tar72] wird verwendet, um einen Graphen G in seine starken Zusammenhangskomponenten aufzuteilen (vgl. Algorithmus 2.2, 2.3), indem ein Spannbaum mit topologischer Ordnung für G aufgebaut wird. Für jeden Knoten wird dann der niedrigste erreichbare Knoten ermittelt. Alle Knoten mit dem selben niedrigsten erreichbaren Knoten bilden eine starke Zusammenhangskomponente. Hierfür wird der Graph mit einer Tiefensuche durchlaufen. Für alle Knoten wird festgehalten, wann diese gefunden wurden. Auch wird für jeden Knoten v berechnet, welcher der am frühsten gefundene Knoten ist, der von v aus erreichbar ist. In dieser Anwendung ordnet der Algorithmus jedem Knoten seine starke Zusammenhangskomponente zu.

Tarjans Algorithmus findet alle starken Zusammenhangskomponenten in $O(|V| + |E|)$.

Als Alternative zu Tarjans Algorithmus sei an dieser Stelle noch Kosaraju's Algorithmus [Sha81] genannt, der starke Zusammenhangskomponenten in der asymptotisch gleichen Laufzeit findet.

2.2.3. Disjoint Set Forest

Ein *Disjoint Set Forest* (DSF) ist eine *union/find* Datenstruktur. Solche Datenstrukturen werden dazu verwendet, disjunkte Mengen von Objekten zu verwalten. Hierbei wird jede

Algorithm 2.3: DFS

```
Input: Vertex u
1  stack.PUSHBACK(u)
2  on_stack[u] ← true
   // ids contains the number each visited vertex gets when it is
      discovered
3  ids[u] ← vertex_count
   // low contains the lowest numbered vertex currently reachable
4  low[u] ← vertex_count
5  vertex_count ← vertex_count + 1
   // visit all successors
6  forall successors v of u do
7      if ids[v] = −1 then
8          DFS(v)
9          low[u] ← MIN(low [v],low [u])
10     else if on_stack[v] then
11         low[u] ← MIN(low [v],ids [u])
   // if SCC found
12 if ids[v] = low[v] then
13     while true do
14         x ← stack.TOP
15         on_stack[x] ← false
16         coloration[x] ← scc_count
17         stack.POP
18         if x = u then
19             break
20     ++scc_count
```

Menge durch einen sogenannten Repräsentanten identifiziert. Ein Repräsentant ist ein in der Menge enthaltenes Objekt. Eine *union/find* Datenstruktur beinhaltet zwei Operationen: Eine *union* Operation, um Mengen effizient zu vereinigen (vgl. Algorithmus 2.4) und eine *find* Operation, um den eindeutigen Repräsentanten einer solchen Menge zu finden (vgl. Algorithmus 2.5). In einem DSF werden Mengen in einer Baumstruktur gespeichert.
Um Operationen auf einem DSF effizient zu machen, werden klassischer Weise zwei Optimierungen implementiert: *Union by rank* stellt sicher, dass die *union* Operation immer kleinere Bäume an größere Bäume anhängt. Auf diese Weise bleiben die Bäume möglichst flach und damit die Pfade zum Repräsentanten einer Menge kurz. Die Pfadkompression stellt sicher, dass jedes Objekt, welches von der *find* Operation besucht wird, zu einem direkten Kind des Repräsentanten wird. Auf diese Weise werden die Pfade verkürzt, was die *find* Operation effizienter werden lässt. In diesem Anwendungsfall ist es wichtig, welches Objekt der Repräsentant einer Menge ist. Daher muss hier auf *union by rank* verzichtet werden.

Für eine Menge von n Objekten, auf denen eine Sequenz von m *union* oder *find* Operationen ausgeführt wird, gelten folgende Laufzeiten [TVL84]:

Ohne Optimierungen: $O(mn)$

Union By Rank: $O(m \log n)$

Algorithm 2.4: UNION

Input: int u, int v

```
1 parent [FIND(u)] = FIND(v)
```

Algorithm 2.5: FIND

Input: int u, int v

```
1 if parent[v] = v then
2     RETURN(v)
3 RETURN parent[v] = FIND(parent [v ])
```

Pfadkompression: $O(m \log n)$

***Union By Rank* und Pathcompression:** $O(m\alpha(m+n, n))$, wobei α die inverse Ackermann Funktion darstellt [Sun71].

3. Der Algorithmus

Der Algorithmus folgt im Allgemeinen einem *Branch&Bound* Ansatz. Die Funktion SOLVE (vgl. Algorithmus 3.1) bekommt als Eingabe eine aktuelle Instanz I, eine partielle Lösung **dfvs** für die ursprüngliche Instanz sowie deren beste bisher gefundene Lösung **upper_bound** als globales Objekt. Zunächst wird eine Instanz des DFVS-Problems reduziert (vgl. Abschnitt 3.1). Ist die reduzierte Instanz leer und **dfvs** beinhaltet weniger Knoten als **upper_bound**, so wird **upper_bound** durch **dfvs** ersetzt. Ist dies nicht der Fall, wird die reduzierte Instanz in starke Zusammenhangskomponenten aufgeteilt, von denen jede unabhängig gelöst werden kann. In der größten der starken Zusammenhangskomponenten werden für einen ausgewählten Knoten v zwei Lösungen berechnet: Eine der Lösungen folgt der Annahme, dass v in einem optimalen DFVS enthalten ist. Die andere folgt der Annahme, dass ein optimales DFVS v nicht enthält.

Die folgenden Abschnitte beschreiben die essentiellen Schritte des Algorithmus: Reduktion der Instanz (vgl. Abschnitt 3.1), die Aufteilung der Instanz in starke Zusammenhangskomponenten (vgl. Abschnitt 3.2) und das *branching* (vgl. Abschnitt 3.3).

3.1. Reduktionen

Um eine Instanz des DFVS-Problems zu lösen, kann diese zunächst reduziert werden. Das bedeutet, eine Instanz I auf eine kleinere Instanz I' abzubilden. Außerdem ist aus einer optimalen Lösung von I' eine optimale Lösung von I konstruierbar. Hierzu werden die folgenden Reduktionsregeln sukzessive angewandt, bis keine der Regeln mehr anwendbar ist.
Die Regeln werden in aufsteigender asymptotischer Laufzeit angewandt. Nach jeder erfolgreichen Anwendung wird das Verfahren erneut begonnen. Durch diesen Ablauf werden teure Reduktionen auf einer bereits reduzierten Instanz aufgerufen.
Die Reduktionsregeln **IN0**, **OUT0**, **LOOP**, **IN1** und **OUT1** sind in ähnlicher Form in vielen Publikationen zu finden und gehören zum Standard der Graphreduktion für das DFVS-Problem [LJ00], [CBA95], [FWY09]. **PIE**, **CORE** und **DOME** basieren auf [LJ00], wo sich auch deren formelle Beweise finden.

IN0 und OUT0

Die Reduktionsregeln **IN0** und **OUT0** besagen, dass Knoten mit Ein- bzw. Ausgangsgrad 0 aus der Instanz entfernt werden können. Dies ist möglich, da Knoten ohne ein- bzw. ausgehende Kanten nicht Teil eines Kreises sein können.

Algorithm 3.1: SOLVE

```
Input: Instance I, Set dfvs, Set upper_bound
dfvs' ← ∅
if REDUCE(I,dfvs',upper_bound) then
    RETURN
dfvs ← dfvs ∪ dfvs'
if I is empty then
    if |dfvs| < |upper_bound| then
        upper_bound ← dfvs
    RETURN
SCC ← Set of all SCCs in I
c' ← biggest c ∈ SCC
forall c ∈ SCC \ c' do
    dfvs' ← ∅
    SOLVE(c, dfvs', INITIAL_SOLUTION(c))
    dfvs ← dfvs ∪ dfvs'
v ← some vertex in c'
SOLVE(c' \ v, dfvs ∪ v, upper_bound)
ignore v in c'
SOLVE(c', dfvs, upper_bound)
```

LOOP

Die Reduktionsregel *LOOP* besagt, dass Knoten mit einer Schleife in jedem DFVS vorkommen müssen, da Schleifen Kreise der Länge 1 sind und nur so gebrochen werden können.

IN1

Die Reduktionsregel **IN1** besagt, dass ein Knoten v mit Eingangsgrad 1 in seinen eindeutigen Vorgänger vereinigt werden kann. Dies ist möglich, da jeder Kreis, der v enthält, auch dessen Vorgänger enthalten muss.

OUT1

Die Reduktionsregel **OUT1** besagt, dass ein Knoten v mit Ausgangsgrad 1 in seinen eindeutigen Nachfolger vereinigt werden kann. Dies ist möglich, da jeder Kreis, der v enthält, auch dessen Nachfolger enthalten muss.

PIE

Kanten zwischen verschiedenen starken Zusammenhangskomponenten können nicht auf Kreisen liegen. Die Reduktionsregel **PIE** führt diese Beobachtung noch weiter und besagt, dass alle Kanten $e \in E(I)$ zwischen verschiedenen starken Zusammenhangskomponenten in $I \setminus E(\Pi(I))$ aus I entfernt werden können, denn: Wenn eine solche Kante $e = (u, v)$ Teil eines Kreises k in I ist, so enthält k eine Kante $(i, j) \in \Pi(I)$. Da jedes DFVS i oder j enthalten muss, wird k unabhängig von e von jedem DFVS gebrochen. Die Kante e muss also nicht beachtet werden.

CORE

Für eine Instanz I kann $\Pi(I)$ als ungerichteter Graph interpretiert werden. Um alle Kreise in einer Clique $C \subset \Pi(I)$ der Größe n zu brechen, sind genau $n-1$ Knoten nötig. Gilt für einen Knoten $v \in C$, dass für alle seine Nachbarn $u \in V(I)$ auch gilt $u \in C$, so können alle Knoten $C \setminus v$ für ein optimales DFVS gewählt werden.

DOME

Die Reduktionsregel **DOME** basiert auf dem Prinzip dominierter Kanten [LJ00]. Kanten heißen hier dominiert, wenn sie nicht Teil eines minimalen Kreises sein können. Da nur minimale Kreise von einem DFVS gebrochen werden müssen, besagt das Prinzip, dass dominierte Kanten nicht zum Finden eines DFVS beachtet werden müssen.
Um eine Kante $(u, v) \in E(I)$ als dominiert zu klassifizieren, darf für sie keine Rückkante existieren und sie muss eine der folgenden Bedingungen erfüllen:

1. $P_u \subseteq P_v$, wobei $P_u = \{i \in V(I) | (i, u) \in E(I) \wedge (i, u) \notin E(\Pi(I))\}$ und $P_v = \{j \in V(I) | (j, v) \in E(I)\}$
2. $S_v \subseteq S_u$, wobei $S_u = \{i \in V(I) | (u, i) \in E(I)\}$ und $S_v = \{j \in V(I) | (v, j) \in E(I) \wedge (v, j) \notin E(\Pi(I))\}$

Intuitive Erklärung der Bedingungen: Jeder Kreis k, der die Kante (u, v) enthält, enthält auch eine Kante (i, u) für einen Vorgänger i von v. Gilt Bedingung 1 und $(i, u) \in E(\Pi(I))$, so kann k kein minimaler Kreis sein und muss daher nicht beachtet werden. Gilt Bedingung 1 und $(i, u) \notin E(\Pi(I))$, so ist i auch ein Vorgänger von v. Statt dem Kreis k muss also nur der minimale Kreis k' gebrochen werden, der aus k hervorgeht, indem u aus k entfernt wird. Bedingung 2 lässt sich analog hierzu begründen.

SHORTCUT

Die Reduktionsregel **SHORTCUT** besagt, dass jeder Knoten v ohne Schleife, der auf einem Kreis aber auf keinen zwei Kreisen liegt, die ausschließlich v als gemeinsamen Knoten haben, ignoriert werden kann (vgl. Abschnitt 4.1.4) [FWY09]. Dies ist möglich, da in diesem Fall ein Knoten u existieren muss, der v dominiert. In diesem Fall kann für ein DFVS immer u anstatt v gewählt werden.
Für Knoten, die auf einem Kreis liegen, handelt es sich hierbei um eine Verallgemeinerung der **IN1** und **OUT1** Regeln. Diese sind aber schneller als **SHORTCUT** und daher dennoch relevant.

FLOWER

Die Reduktionsregel **FLOWER** basiert auf [FWY09] und besagt, dass gegeben ein DFVS S der Kardinalität k ein Knoten v in einem minimalen DFVS enthalten sein muss, wenn v auf mindestens $k+1$ knotendisjunkten Kreisen liegt. Dies gilt, da ein DFVS aus jedem Kreis mindestens einen Knoten enthalten muss, um alle Kreise zu brechen. Um ein DFVS S' zu finden, das kleiner ist als S, müssen also insbesondere alle Kreise gebrochen werden, die v enthalten. Gilt $v \notin S'$, werden hierfür mindestens $k+1$ Knoten benötigt. S' kann also nicht kleiner sein als S.

3.2. Aufteilung in Komponenten

Die starken Zusammenhangskomponenten einer Instanz können zur Berechnung eines DFVS unabhängig voneinander betrachtet werden, denn die Kanten zwischen verschiedenen Zusammenhangskomponenten der Instanz I können nicht Teil eines Kreises sein. Auf diese Weise bleibt eine nicht zusammenhängende Instanz I' zurück. Ein optimales DFVS einer solchen Instanz I' ist die Vereinigung der DFVS ihrer Komponenten und auch ein optimales DFVS der Instanz I.

3.3. Branching

Kann eine starke Zusammenhangskomponente nicht weiter reduziert werden, so wird ein Knoten v aus dieser Komponente gewählt. Da nicht klar ist, ob v in einer optimalen Lösung der Komponente enthalten ist, werden beide Optionen ausprobiert. Die Komponente wird also zweimal gelöst: Mit und ohne v in der Lösung.

4. Implementierung

Instanz

Instanzen des DFVS-Problems beinhalten in der Implementierung einen Graphen G sowie eine *map*. G wird in Form von ausgehenden Adjazenzlisten gespeichert, $Adj(v)$ bezeichnet im Folgenden die Adjazenzliste des Knoten v. Die *map* wird dazu verwendet, um Knoten der Instanz auf Knoten einer anderen Instanz abzubilden. Genauer wird die ursprüngliche Instanz, auf welcher der Algorithmus aufgerufen wird, durch verschiedene Operationen verändert, wobei sich die Indizes der Knoten ändern. Die *map* wird verwendet, um zu einem Knoten v den Index zu finden, den dieser Knoten v im ursprünglichen Graphen hatte. Die *map* ist als eine Liste von $|V|$ Zahlen implementiert, deren i-ter Eintrag den Index des Knoten i im ursprünglichen Graphen speichert. Wird im Folgenden ein Knoten v in das aktuelle **dfvs** eingefügt, bedeutet dies $map[v]$ in **dfvs** einzufügen, um eine Lösung für die ursprüngliche Instanz zu erhalten. Während der gesamten Ausführung des Algorithmus gelten die folgenden Invarianten, um einige Verfahren zu beschleunigen:

1. In der Instanz existieren keine Mehrfachkanten.
2. Alle Adjazenzlisten der Instanz sind sortiert.

Heuristiken zum Wählen von Knoten

Im Folgenden werden Knoten benötigt, die mit hoher Wahrscheinlichkeit Element eines optimalen DFVS sind. Um aus einer Instanz I einen solchen Knoten zu wählen, sind zwei verschiedene Heuristiken implementiert:

Heuristik 1: Knotengrad. Wählt den Knoten mit dem höchsten Knotengrad, da ein solcher Knoten v mit hoher Wahrscheinlichkeit auf einem Kreis liegt. Das Entfernen von v führt zusätzlich zum Entfernen vieler Kanten, was an einigen Stellen Möglichkeiten zur weiteren Reduktion der Instanz eröffnen kann.

Heuristik 2: Produkt aus Ein- und Ausgangsgrad. Wählt einen Knoten mit dem höchsten Produkt aus Eingangsgrad und Ausgangsgrad. Ein solcher Knoten v liegt mit hoher Wahrscheinlichkeit auf vielen Kreisen. Desweiteren ist das Produkt aus Ein- und Ausgangsgrad für einen Knoten v auch die Anzahl an Kanten, die in einer Instanz entstehen, wenn v ignoriert wird. Entstehen viele Kanten, entstehen auch mit hoher Wahrscheinlichkeit Schleifen, die anschließend zu weiteren Reduktionen führen.

Beide Heuristiken finden einen Knoten v in $O(|V| + |E|)$.

4.1. Grundlegende Operationen

4.1.1. Knoten löschen

Um einen Knoten $v \in V(I)$ zu löschen werden zunächst dessen Adjazenzliste und damit all seine ausgehenden Kanten gelöscht. Beinhaltet die Instanz eine *map*, wird auch hier das v-te Element entfernt. Hierbei verändern sich die Indizes aller Knoten $j > v$. Um alle eingehenden Kanten von v zu löschen und alle Kanten (i, j) mit $j > v$ zu korrigieren, geht die Implementierung wie folgt vor:
In jeder Adjazenzliste wird mit Hilfe einer binären Suche das erste Element e gefunden, für das gilt $e \geq v$. Da alle Adjazenzlisten sortiert sind, müssen diese nur an den Elementen an und hinter der Stelle von e angepasst werden. Eine binäre Suche findet e schneller, als die gesamte Liste zu durchsuchen. Falls gilt $e = v$ wird e gelöscht. Alle auf e folgenden Elemente der Adjazenzliste werden dekrementiert. Dieses Vorgehen korrigiert alle Kanten, die sich durch das Löschen des Knoten v verändert haben, da sich nach Invariante 2 alle betroffenen Kanten an der Stelle von oder hinter e befinden. Außerdem kann wegen Invariante 1 nur für das Element e gelten, dass $e = v$. Es müssen also keine anderen Elemente gelöscht werden.

Das Löschen von $Adj(v)$ sowie das Löschen von v in *map* erfolgt in $O(|V|)$. Das Korrigieren der Kanten erfolgt in $O(|E|)$. Das Löschen eines Knotens aus einer Instanz benötigt also $O(|V| + |E|)$ Zeit.

4.1.2. Knotenmenge löschen

In den meisten Fällen soll aus einer Instanz I nicht nur ein einziger Knoten, sondern eine Menge S von Knoten auf einmal gelöscht werden. Hierfür wird zunächst ein *alive* Array erstellt. An dessen i-ter Stelle steht eine 0, wenn der Knoten i gelöscht werden soll, ansonsten eine 1. Für *alive* wird eine Präfixsumme erstellt. In der Präfixsumme steht an der Stelle i die Anzahl der Knoten j, die nicht gelöscht werden sollen und für die gilt $j < i$. Also der neue Index aller Knoten i mit $alive[i] = 1$.
Mit Hilfe des *alive* Arrays und dessen Präfixsumme werden neue Adjazenzlisten und eine neue *map* aufgebaut, welche die alten ersetzen. Hierfür wird für jede Kante (v, w) eine Kante $(Präfixsumme(v), Präfixsumme(w))$ erstellt, falls gilt $alive[v] = alive[w] = 1$. In die neue *map* werden alle Einträge der nicht gelöschten Knoten übernommen.

Die Berechnung von *alive*, der Präfixsumme sowie der neuen *map* erfolgt jeweils in $O(|V|)$. Das Erstellen der neuen Adjazenzlisten kann in $O(|V| + |E|)$ erfolgen, da hier nur für jeden Knoten $u \in V(I)$ $alive[u] = 1$ und gegebenenfalls für jedes Element v in $Adj(u)$ $alive[v]$ überprüft werden muss. Da die Adjazenzlisten bereits vor dieser Operation sortiert waren (Invariante 2), kann die Reihenfolge in den Adjazenzlisten beibehalten werden. Das Löschen einer Knotenmenge aus einer Instanz benötigt also $O(|V| + |E|)$ Zeit.

4.1.3. Knoten vereinigen

Diese Operation nimmt für eine Instanz I eine Menge **merges** von Knotenpaaren (u, v) entgegen, die vereinigt werden sollen. Um aus **merges** die neue Knotenmenge V' der neuen Instanz I' zu berechnen, wird ein DSF verwendet (vgl. Abschnitt 2.2.3).

Alle Repräsentanten des DSF werden als *alive* markiert und eine Präfixsumme berechnet. Auf diese Weise ist die Präfixsumme eines Knoten $v \in V(I)$ sein Index in der neuen Knotenmenge V'.
Um die neue Kantenmenge E' der neuen Instanz I' zu berechnen, wird für jede Kante

$(u, v) \in E(I)$ eine neue Kante zwischen den Repräsentanten von u und v in E' erstellt. Mehrfachkanten in E' werden entfernt.

Um die neue Instanz I' zu erstellen, wird aus E' für jeden Knoten $v \in V'$ eine neue Adjazenzliste berechnet. Enthält I eine *map*, so werden nur die Einträge übernommen, die zu Knoten aus V' gehören.

Die Verwendung des DSF findet hier als eine Sequenz von e *union* Operationen gefolgt von $|V(I)|$ *find* Operationen statt. Um Mehrfachkanten aus E' zu entfernen, wird hier sortiert. Damit beläuft sich die Laufzeit dieser Operation auf $O((e + |V|) \log |V| + |E| \log |E|)$. Da Kanten durch Paare natürlicher Zahlen dargestellt werden, kann hier auch in Linearzeit sortiert werden. Die Vereinigung von Knoten kommt jedoch ausschließlich in Reduktionsregeln zum Einsatz, die selbst von komplexeren Regeln dominiert werden. Der zusätzliche logarithmische Faktor fällt hier also asymptotisch nicht ins Gewicht.

4.1.4. Knoten ignorieren

Wenn ein Knoten v nicht Teil eines minimalen DFVS sein kann, muss dieser nicht weiter beachtet werden. Die an v angrenzenden Kanten können allerdings nicht gelöscht werden. Um alle potentiellen Kreise beizubehalten, kann der Knoten v gelöscht werden, wenn jeder Pfad (i, v, j) durch eine Kante (i, j) ersetzt wird.

Die Implementierung dieser Operation geht dafür folgendermaßen vor: Soll ein Knoten v ersetzt werden, so wird für jede Kante (i, j) überprüft, ob $j = v$ oder $j > v$. Im Fall von $j = v$ wird die Kante gelöscht und i als ein Vorgänger von v gespeichert. Gilt $j > v$, so wird die Kante (i, j) durch die Kante $(i, j - 1)$ ersetzt. Auf diese Weise werden bereits beim Suchen der Vorgänger von v die Adjazenzlisten aller anderen Knoten $u \in V(I)$ aktualisiert. Da alle Knoten $u > v$ dekrementiert werden und alle Vorkommen von v gelöscht wurden, bleiben alle Adjazenzlisten sortiert und damit Invariante 2 erhalten.
Anschließend werden in alle Adjazenzlisten der Vorgänger von v die Nachfolger von v (genau die Einträge in $Adj(v)$) eingefügt (vgl. Algorithmus 4.1). Um die Invarianten 1 und 2 beizubehalten, wird eine neue Adjazenzliste aufgebaut, welche die Adjazenzliste des Vorgängers von v ersetzt. Hier werden jeweils die ersten Elemente der beiden zu vereinigenden Listen verglichen. Das kleinere der beiden Elemente wird in die neue Adjazenzliste übernommen. Sind beide Elemente gleich, wird eines davon übersprungen. Zuletzt werden die Adjazenzliste von v und der zu v gehörige *map*-Eintrag gelöscht.

Das Finden aller Vorgänger von v erfolgt in $O(|E|)$. Das Einfügen der Nachfolger von v in alle Adjazenzlisten seiner Vorgänger erfolgt in $O(|V|^2)$, da v höchstens $|V|$ Nachfolger haben kann (vgl. Invariante 1). Das Löschen von $Adj(v)$ sowie des zugehörigen Eintrags in *map* erfolgt jeweils in $O(|V|)$. Einen Knoten in einer Instanz zu ignorieren benötigt also $O(|V|^2)$ Zeit.

4.1.5. Berechnung von $\Pi(I)$

Die Implementierung (vgl. Algorithmus 4.2) erzeugt hierbei eine Instanz, in der alle Knoten $V(I)$ enthalten sind, um die Indizes der Knoten beizubehalten. Die Π-Knoten sind hier alle Knoten mit angrenzenden Kanten. Sind im folgenden nur die Knoten in $\Pi(I)$ interessant, so werden die Knoten ohne angrenzende Kanten übersprungen. Zur Berechnung von $\Pi(I)$ werden hier die Invarianten 1 und 2 ausgenutzt.
Für jeden Knoten u wird für jede Kante (u, v) überprüft, ob $(v, u) \in E(I)$, indem $Adj(v)$

Algorithm 4.1: MERGE

Input: Array array_1, Array array_2

```
1  pointer_1 ← 0
2  pointer_2 ← 0
3  new_array ← ∅
4  while pointer_1 ≠ |array_1| ∧ pointer_2 ≠ |array_2| do
5      if pointer_1 = |array_1| then
6          new_array.PUSH_BACK(array_2[pointer_2])
7          pointer_2 ← pointer_2 + 1
8      else if pointer_2 = |array_2| then
9          new_array.PUSH_BACK(array_1[pointer_1])
10         pointer_1 ← pointer_1 + 1
11     else if pointer_1 > pointer_2 then
12         new_array.PUSH_BACK(array_2[pointer_2])
13         pointer_2 ← pointer_2 + 1
14     else if pointer_1 < pointer_2 then
15         new_array.PUSH_BACK(array_1[pointer_1])
16         pointer_1 ← pointer_1 + 1
17     else
18         new_array.PUSH_BACK(array_1[pointer_1])
19         pointer_1 ← pointer_1 + 1
20         pointer_2 ← pointer_2 + 1
21 array_1 ← new_array
```

bis zum ersten Element $e \geq u$ durchlaufen wird. Gilt $e = u$, wird die Kante (u, v) in $\Pi(I)$ übernommen. Werden die Knoten in aufsteigender Reihenfolge betrachtet, kann eine Rückkante von w zu einem Knoten $i > j$ in keiner Adjazenzliste vor der Rückkante zu dem Knoten j stehen. Wurde $Adj(w)$ also schon bis zum ersten Element $e \geq j$ durchlaufen, kann für j die Suche ab dem Element e begonnen werden. $I \setminus E(\Pi(I))$ lässt sich analog zu $\Pi(I)$ berechnen, indem eine Kante (u, v) übernommen wird, wenn $(v, u) \notin E(I)$.

$\Pi(I)$ lässt sich auf diese Weise in $O(|V| + |E|)$ berechnen.

4.2. Lower und Upper Bounds

Lower und *upper bounds* werden verwendet, um möglichst früh im Suchbaum feststellen zu können, ob der aktuelle Ast schon gar nicht mehr zu einer optimalen Lösung führen kann. Das *upper bound* ist hierbei ein meist nicht optimales DFVS der ursprünglichen Instanz. Das *lower bound* ist die Anzahl an Knoten, die mindestens benötigt werden, um für die aktuelle Instanz ein DFVS zu bilden. Gilt also *lower bound* + $|\mathbf{dfvs}| \geq |\mathbf{upper\_bound}|$, so kann aus der aktuellen Instanz zusammen mit dem aktuellen **dfvs** kein DFVS berechnet werden, das besser ist als das bereits bekannte DFVS *upper bound*. In diesem Algorithmus findet das Überprüfen dieser Bedingung während der Reduktion einer Instanz statt (vgl. Abschnitt 4.3).

Die folgenden Abschnitte 4.2.1 und 4.2.2 beschreiben, wie in diesem Algorithmus die *upper* bzw. *lower bounds* zustande kommen.

Algorithm 4.2: CALCULATE_PIE

```
Input: Instance I
1 forall v ∈ V(I) do
2     pointer[v] = 0
3 for u = 0..(|V| − 1) do
4     for v ∈ Adj(u) in ascending order do
5         while pointer[v] ≠ |Adj(v)| ∧ Adj(v)[pointer[v]] < u do
6             ++ pointer[v]
7         if u ≠ v ∧ ¬pointer[v] ≠ |Adj(v)| ∧ Adj(v)[pointer[v]] = u then
8             Π ←Π ∪(u, v)
9 RETURN Π
```

4.2.1. Upper Bound

Ein *upper bound* kann hier auf drei verschiedene Weisen zustande kommen. Die erste ist ein naives *upper bound*: die gesamte Knotenmenge $V(I)$. Werden alle Knoten einer Instanz entfernt, so ist diese kreisfrei, also $V(I)$ ein DFVS. Mit diesem *upper_bound* wird der Algorithmus initial gestartet. Wird ein DFVS mit weniger Knoten als das aktuelle *upper bound* für die ursprüngliche Instanz gefunden, so wird *upper bound* durch diese Lösung ersetzt. Die letzte Art, wie ein *upper bound* in diesem Algorithmus zustande kommt, ist das Berechnen einer initialen Lösung. Wenn eine Teilinstanz unabhängig gelöst werden kann, so kann diese als eigenes Problem mit einer eigenen Lösung interpretiert werden. Für diesen Fall wird eine initiale, aber nicht optimale, Lösung für das Teilproblem berechnet und als *upper bound* in diesem Ast verwendet. Dieses Vorgehen wird beim Lösen von starken Zusammenhangskomponenten angewandt (vgl. Abschnitt 4.4).

Um initiale Lösungen zu berechnen, sind die folgenden zwei Ansätze implementiert:

Ansatz 1: Greedy. Dieser Ansatz entfernt so lange Knoten aus einer Instanz I, bis I azyklisch ist. Hierfür werden Knoten allerdings lediglich als gelöscht markiert und nicht tatsächlich entfernt. Um zu überprüfen, ob I azyklisch ist, werden so lange Tiefensuchen durchgeführt, bis alle Knoten in I gefunden wurden. Um auszuwählen, welcher Knoten als nächstes entfernt werden soll, wird eine der beiden zuvor vorgestellten Heuristiken (vgl. Abschnitt 4) verwendet.
Da hier für jede Iteration eine Heuristik sowie eine Tiefensuche durchgeführt wird, beläuft sich die Laufzeit dieses Ansatzes auf $O(|V|^2 + |V||E|)$.

Ansatz 2: Reduktion. Dieser Ansatz folgt dem Vorschlag von [LJ00] und reduziert eine Instanz I zu einer Instanz I' (vgl. Abschnitt 4.3). Dabei wird der Knoten mit der geringsten Heuristik (vgl. Abschnitt 4) in I' ignoriert. Anschließend werden Knoten gefunden, die in einem optimalen DFVS enthalten sein müssen, und in die Menge **dfvs** eingefügt. Dieses Vorgehen wird wiederholt, bis I vollständig reduziert wurde. Die Menge **dfvs** bildet die initiale Lösung.
Es wird ein Knoten mit der geringsten Heuristik gewählt, da so ein Knoten ignoriert wird, der wahrscheinlich nicht in einem optimalen DFVS enthalten ist. Außerdem entstehen durch das Ignorieren von Knoten neue Kanten, die zu erfolgreichen Reduktionen führen können. Würde dagegen der Knoten mit der höchsten Heuristik in **dfvs** aufgenommen und aus der Instanz entfernt, würden keine neuen Kanten entstehen.

In einer nicht finalen Version des Algorithmus führte Ansatz 2 zwar zu besseren Ergebnissen als Ansatz 1, benötigte hierfür allerdings unverhältnismäßig mehr Zeit. Ansatz 2 wurde daher in der Entwicklung des Algorithmus nicht weiter verfolgt.

Der Algorithmus berechnet für unabhängig gelöste starke Zusammenhangskomponenten eine initiale Lösung (vgl. Abschnitt 4.4). Der Gedanke liegt nahe, eine solche initiale Lösung auch zu Beginn für die gesamte Instanz zu berechnen. Laufzeitmessungen mit einer früheren Version des Algorithmus ergaben allerdings, dass der Zeitaufwand der initialen Lösung deren Nutzen übersteigt.
Eine mögliche Erklärung hierfür ist, dass die erste Reduktion bereits so viele Knoten und Kanten aus der Instanz entfernt, dass die initiale Lösung schon nicht mehr repräsentativ ist. Da die starken Zusammenhangskomponenten immer auf bereits reduzierten Instanzen berechnet werden, ist dieser Effekt hier deutlich schwächer.

4.2.2. Lower Bound

Um für eine Instanz I ein *lower bound* zu berechnen, sind zwei Ansätze implementiert:

Lower Bound 1: Matching. Dieser Ansatz interpretiert $\Pi(I)$ als ungerichteten Graphen und findet auf diesem ein Matching. Da jede Kante $e \in E(\Pi(I))$ einen minimalen Kreis der Länge 2 in I darstellt, findet dieser Ansatz eine Menge an minimalen Kreisen in I, die auf jeden Fall gebrochen werden müssen. Das Matching wird *greedy* berechnet, indem in $\Pi(I)$ eine Kante (u, v) ausgewählt und die Knoten u und v markiert werden. Sind alle Knoten in $\Pi(I)$ markiert, bilden die ausgewählen Kanten ein Matching.
Nach der Berechnung von $\Pi(I)$ werden hier solange Kanten gefunden, bis alle Knoten markiert wurden. Auf diese Weise kann also ein *lower bound* in $O(|V| + |E|)$ berechnet werden.

Lower Bound 2: Reduktion. Dieser Ansatz folgt dem von [LJ00] vorgeschlagenen Verfahren zum Finden eines *lower bound*. Hier wird die größte Clique $C \in \Pi(I)$ gefunden, aus I entfernt und *lower bound* um $|C| - 1$ inkrementiert. Kann keine Clique aus mindestens zwei Knoten gefunden werden, wird der kleinste Kreis K aus I entfernt und *lower bound* inkrementiert. Anschließend wird die Instanz reduziert. Werden dabei Knoten gefunden, die in einem optimalen DFVS enthalten sein müssen, wird das *lower bound* um die Anzahl dieser Knoten erhöht. Dieses Verfahren wird wiederholt, bis I leer ist. Die Reduktion in diesem Schritt unterscheidet sich zu dem in Abschnitt 4.3 beschriebenen Verfahren dadurch, dass keine *bounds* berechnet werden.
Da das Finden einer größten Clique selbst NP-vollständig ist, wird hier das Prinzip der gemeinsamen Nachbarn angewandt [TS86]. Hierbei werden zu bereits gefundenen Cliquen solange wie möglich einzelne Knoten hinzugefügt, die zu allen Knoten der Clique benachbart sind.

4.3. Reduktionen

Um eine Instanz I zu reduzieren (vgl. Algorithmus 4.3), wird zunächst ein *lower bound* (vgl. Abschnitt 4.2) für I berechnet. Wird die Berechnug an dieser Stelle nicht abgebrochen, so werden iterativ die Reduktionen angewandt. Die Reihenfolge der Reduktionen orientiert sich hierbei nicht ausschließlich an deren asymptotischer *worst case* Laufzeit, da einige Reduktionen vom Entfernen von Knoten dominiert werden. Einige der Reduktionen finden Knoten, die in einem optimalen DFVS enthalten sein müssen. Da das aktuelle **dfvs** so wächst, wird in jeder Iteration des Verfahrens überprüft, ob $|\mathbf{dfvs}| \geq |\mathbf{upper\_bound}|$. Ist dies der Fall, kann die Berechnung für diesen Zweig abgebrochen werden. Ein Durchlauf

Algorithm 4.3: REDUCE

```
Input: Instance I,dfvs,upper_bound
1  if LOWER_BOUND(I) + dfvs ≥ upper_bound then
2      RETURN 1
3  while true do
4      if dfvs ≥ upper_bound then
5          RETURN 1
6      if OUT0 is successful then
7          CONTINUE
8      if OUT1 is successful then
9          CONTINUE
10     if LOOP is successful then
11         CONTINUE
12     if IN0 is successful then
13         CONTINUE
14     if IN1 is successful then
15         CONTINUE
16     if PIE_OPERATION is successful then
17         CONTINUE
18     if CORE_OPERATION is successful then
19         CONTINUE
20     if DOME_OPERATION is successful then
21         CONTINUE
22     if SORTCUT is successful then
23         CONTINUE
24     if FLOWER is successful then
25         CONTINUE
26     BREAK
27 if LOWER_BOUND(I) + dfvs ≥ upper_bound then
28     RETURN 1
29 RETURN 0
```

einer solchen Reduktion wird asymptotisch von der **FLOWER** Reduktion dominiert und erfolgt in $O(|V|^2|E|)$.

IN0

Die Implementierung dieser Reduktion durchläuft alle Kanten $(u, v) \in E(I)$ und markiert dabei die gefundenen Knoten v. Alle Knoten, die auf diese Weise nicht markiert wurden, haben einen Eingangsgrad von 0 und werden entfernt (vgl. Abschnitt 4.1.2).

Das Durchlaufen aller Kanten erfolgt in $O(|E| + |V|)$. Mit dem Löschen der Knoten aus I ergibt sich also eine Laufzeit von $O(|E| + |V|)$ für diese Reduktion.

OUT0

Die Implementierung dieser Reduktion überprüft für jeden Knoten, ob dessen Adjazenzliste leer ist. Ist dies der Fall, so wird er markiert. Alle markierten Knoten werden anschließend

aus der Instanz entfernt (vgl. Abschnitt 4.1.2).

Das Markieren der Knoten erfolgt also in $O(|V|)$. Mit dem Löschen der markierten Knoten ergibt sich also eine Laufzeit von $O(|E| + |V|)$ für diese Reduktion.

LOOP

Die Implementierung dieser Reduktion überprüft für jeden Knoten v, ob er eine Schleife hat. Ist dies der Fall, so wird v markiert sowie zum aktuellen **dfvs** hinzugefügt. Nachdem alle Knoten überprüft wurden, wird S aus der Instanz entfernt (vgl. Abschnitt 4.1.2). Wegen Invariante 2 kann $v \in Adj(v)$ mit binärer Suche berechnet werden.

Der markieren der Knoten erfolgt also in $O(|V| \log |E|)$. Mit dem Löschen der markierten Knoten aus I ergibt sich also eine Laufzeit von $O(|V| \log |E| + |E|)$ für diese Reduktion.

IN1

Die Implementierung dieser Reduktion (vgl. Algorithmus 4.4) berechnet zunächst für alle Knoten ihren Eingangsgrad und einen Vorgänger. Für Knoten mit einem Eingangsgrad von 1 ist dieser Vorgänger eindeutig, für Knoten mit einem anderen Eingangsgrad ist der gefundene Vorgänger irrelevant. Auch eine leere Menge **merges** wird erstellt. Für alle Elemente dieser MEnge (i, j) werden am Ende der Reduktion der Knoten i in den Knoten j vereinigt. Da ein Knoten in dieser Reduktion in seinen Vorgänger vereinigt werden muss, ist die Reihenfolge innerhalb der Tupel hier wichtig.
Für jeden Knoten v mit einem Eingangsgrad von 1 und dem Vorgänger u wird das Knotenpaar (v, u) in **merges** eingefügt und v aus $Adj(u)$ entfernt, um das Entstehen von Schleifen zu verhindern (Z. 9,10). Liegt v allerdings auf einem Kreis, in dem jeder Knoten einen Eingangsgrad von 1 hat, so muss der Knoten im reduzierten Graphen, der diesen Kreis repräsentiert, eine Schleife haben. Um dieses Verhalten sicher zu stellen, werden die Knoten nicht in beliebiger Reihenfolge überprüft. Wird ein Knoten mit einem Eingangsgrad von 1 gefunden, wird als nächstes sein Vorgänger überprüft (Z. 11). Wird auf diese Weise ein Kreis festgestellt (Z. 7), wird eine Kante im Kreis erhalten, die später eine Schleife bilden wird. Da nur Knoten mit einem Eingangsgrad von 1 überprüft werden, wird der Eingangsgrad von bereits überprüften Knoten auf 0 gesetzt, um das mehrfache Überprüfen eines Knotens zu verhindern (Z. 6).

Diese Reduktion findet alle zu vereinigenden Knotenpaare in $O(|V| + |E|)$. Da bis zu $|V|$ solcher Knotenpaare gefunden werden können, liegt das Vereinigen anschließend in $O(|V| \log |V| + |E| \log |E|)$.

OUT1

Die Implementierung dieser Reduktion erfolgt analog zur Reduktion **IN1**. Hier werden allerdings Knoten mit einem Ausgangsgrad von 1 in ihren Nachfolger vereinigt, da auch hier die Reihenfolge wichtig ist. Auch wird hier immer der Nachfolger eines Knotens als nächstes überprüft, nicht der Vorgänger.

Die Laufzeit dieser Reduktion liegt analog zu **IN1** in $O(|V| + |E|)$ zum Finden der Knotenpaare und in $O(|V| \log |V| + |E| \log |E|)$, um diese zu vereinigen.

Algorithm 4.4: IN1

Input: Instance I

```
1  calculate indegree and predecessor
2  merges := ∅
3  forall {i ∈ V(I)|indegree[i] = 1 ∧ predecessor(i) ≠ i} do
4      v ← i
5      while indegree [v] = 1 do
6          indegree[v] ← 0
7          if predecessor[v] = i then
8              break
9          merges ← merges ∪ {(v, predecessor[v])}
10         delete v in Adj(predecessor[v])
11         v ← predecessor[v]
12 merge all merges in I
```

PIE

Die Implementierung dieser Reduktion berechnet zunächst $I' = I \setminus E(\Pi(I))$ (vgl. Abschnitt 4.1.5) sowie die starken Zusammenhangskomponenten von I'. Anschließend werden alle Kanten zwischen diesen starken Zusammenhangskomponenten aus I entfernt.

Da $\Pi(I)$ sowie die starken Zusammenhangskomponenten der Instanz I' in Linearzeit berechnet werden können, beläuft sich die Laufzeit der Reduktion auf $O(|V| + |E|)$.

CORE

Die Implementierung dieser Reduktion folgt dem von [LJ00] beschriebenen Algorithmus. Zunächst werden hier $\Pi(I)$ (vgl. Abschnitt 4.1.5) sowie der Knotengrad in $\Pi(I)$ für jeden Π-Knoten v berechnet und v als valide markiert. Anschließend wird für jeden Π-Knoten v nach aufsteigendem Knotengrad überprüft, ob v valide und der Kern einer Clique ist. Ist dies der Fall, werden v sowie all seine Nachbarn sowohl als zu löschen als auch als nicht valide markiert. Alle Nachbarn von v werden in das aktuelle **dfvs** eingefügt. Ist v valide, aber nicht Kern einer Clique, werden v sowie all seine Nachbarn als nicht valide markiert. Gibt es keine validen Knoten mehr, werden die zu löschenden Knoten aus I entfernt (vgl. Abschnitt 4.1.2). Zuletzt werden alle Duplikate aus **dfvs** entfernt.

Um zu überprüfen, ob ein Knoten v Kern einer Clique ist, wird für jeden Nachbarn u von v überprüft, ob $(v \cup Nachbarn(v)) \subset Adj(u)$. Wegen Invarianten 1 und 2 ist dies in $O(|E|)$ möglich. Damit beläuft sich die Laufzeit dieser Reduktion auf $O(|V|(\log|V| + |E|))$. Diese Reduktionsregel kann in $O(|V| \log|V| + E)$ implementiert werden, da die vollständige Reduktion einer Instanz aber von der **FLOWER** Reduktion dominiert wird, fällt der zusätzliche Faktor hier asymptotisch nicht ins Gewicht.

DOME

Die Implementierung dieser Reduktion berechnet zunächst, wie in 4.1.5 beschrieben, $\Pi(I)$ sowie $I \setminus E(\Pi(I))$. Diese werden verwendet, um für jede Kante ohne Rückkante die Bedingungen 2 und gegebenenfalls 1 zu prüfen. Erfüllt eine Kante mindestens eine der beiden Bedingungen, wird sie markiert.
Um die Mengen P_u bzw. P_v aufzubauen, wird für jeden Knoten $w \in I$ überprüft, ob

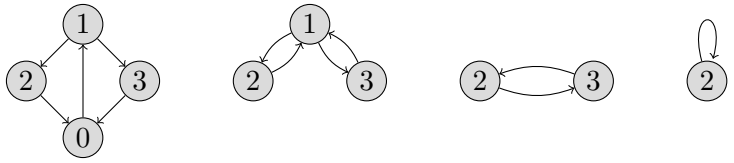
$(w, u) \in E(I \setminus E(\Pi(I)))$ bzw. $(w, v) \in E(I)$. Wegen Invariante 2 kann in $Adj(w)$ mit binärer Suche nach u bzw. v gesucht werden. Da die Knoten in aufsteigender Reihenfolge überprüft werden, sind die Mengen P_u und P_v sortiert und wegen Invariante 1 alle Elemente eindeutig. Für sortierte Mengen kann die Teilmengenrelation in Linearzeit überprüft werden.
Da nur eine der beiden Bedingungen zutreffen muss und Bedingung 2 schneller zu überprüfen ist als Bedingung 1, wird Bedingung 1 nur dann überprüft, wenn Bedingung 2 nicht zutrifft. Die Mengen S_u und S_v sind hier die Adjazenzlisten von u in I bzw. von v in $I \setminus E(\Pi(I))$.
Wurden alle Kanten überprüft, löscht die Reduktion alle markierten Kanten.

Da hier für jede Kante die beiden Bedingungen geprüft werden, ergibt sich für diese Reduktion eine Laufzeit von $O(|V||E| \log |E|)$. Theoretisch ist das Überprüfen beider Bedingungen in $O(|V|)$ zwar möglich. Dafür sind allerdings weitere Datenstrukturen nötig und da eine vollständige Reduktion von der **FLOWER** Reduktion (vgl. Abschnitt 7) dominiert wird, fällt der Faktor $\log |E|$ hier asymptotisch nicht ins Gewicht.

SHORTCUT

Die Implementierung dieser Reduktion folgt Algorithmus 4.5. Um zu überprüfen, ob ein Knoten auf genau einem Kreis liegt, wird zunächst aus der Instanz I eine Instanz **min_cut** aufgebaut. Hierfür werden für jeden Knoten $v \in V(I)$ in **min_cut** die Knoten v_{in} und v_{out} erstellt. Für alle Kanten $(u, v) \in E(I)$ werden die Kanten $(u_{out}, v_{in}) \in E(\textbf{min\_cut})$ erstellt. Zusätzlich beinhaltet **min_cut** für jeden Knoten $v \in V(I)$ die Kante (v_{in}, v_{out}). Dieses Vorgehen wird oft verwendet, um Knotenkapazitäten in einem Flussgraphen zu modellieren.
Um zu überprüfen, ob ein Knoten v auf genau einem Kreis liegt, reicht es zu überprüfen, ob ein einziger, von allen anderen Pfaden knotendisjunkter, Pfad von v_{out} nach v_{in} in **min_cut** existiert. Da jeder Knoten $v \in V(I)$ in **min_cut** durch v_{in}, v_{out} sowie die Kante (v_{in}, v_{out}) dargestellt wird, sind knotendisjunkte Pfade äquivalent zu kantendisjunkten Pfaden.
Die Anzahl der kantendisjunkten Pfade zwischen v_{out} und v_{in} kann daher durch einen Ford-Fulkerson Algorithmus (vgl. Abschnitt 2.2.1) berechnet werden.
Bei dieser Reduktion können nicht wie in den vorherigen Reduktionen erst alle zu ignorierenden Knoten gefunden und dann alle auf einmal bearbeitet werden, wie Beispiel 4.1 zeigt. Alle Knoten v werden von der **SHORTCUT** Regel markiert. Werden alle Knoten gleichzeitig ignoriert, würde G also auf einen leeren Graphen reduziert werden, womit die Reduktionsregel falsch angewendet wäre. Werden alle zu ignorierenden Knoten markiert und in beliebiger Reihenfolge ignoriert, außer es existiert eine Schleife in einem Knoten v, wird G zu einem Graphen $G' = (V', E')$ mit $V = v$, $E = (v, v)$ reduziert. Der Knoten v kann hier auch 2 oder 3 sein, was zu einem falschen reduzierten Graphen führt. Um mit dieser Eigenart der **SHORTCUT** Reduktion umzugehen, werden für jeden Knoten v die Instanzen I sowie **min_cut** direkt angepasst. Soll also v ignoriert werden, so werden v_{in} und v_{out} in **min_cut** auch ignoriert. Für alle nachfolgenden Kanten wird die **SHORTCUT** Reduktion dann auf die bereits reduzierten Instanzen angewendet.

Diese Reduktion wird vom Aufbau des Residualgraphen dominiert und beläuft sich damit auf $O(|V||E|)$. Die Laufzeit des Edmonds-Karp Algorithmus kann hier unterboten werden, da nicht nach der maximalen Anzahl an Pfaden, sondern lediglich nach zweien gesucht wird. Die Laufzeit des allgemeinen Ford-Fulkerson Algorithmus von $O(F|E|)$ wird also auf O(|E|) begrenzt.



a) b) c) d)

Abbildung 4.1.: Mögliche SHORTCUT Reduktion, wenn erst Knoten markiert und dann bearbeitet werden

Algorithm 4.5: SHORTCUT

Input: Instance I

```
1 create min_cut
2 forall v ∈ V(I) do
3     if v has self-loop then
4         continue
5     if (v, ..., v) is a path in min_cut and there are no other vertex disjoint paths
      (v,...,v) then
6         min_cut.IGNORE (v)
7         ignore v in I
```

FLOWER

Die Implementierung dieser Reduktion verhält sich analog zur **SHORTCUT** Reduktion. Zunächst wird also eine Instanz **min_cut** aus der Instanz I aufgebaut. Mit deren Hilfe wird für jeden Knoten v überprüft, ob v auf mindestens $k+1$ knotendisjunkten Kreisen liegt. Für den Parameter k wird $|\mathbf{upper\_bound}| - |\mathbf{dfvs}|$ gewählt, da ein DFVS mit $|\mathbf{upper\_bound}|$ Knoten existiert. **dfvs** bricht in der ursprünglichen Instanz bereits mindestens $|\mathbf{dfvs}|$ Kreise. Ist also $\mathbf{dfvs} \subset S$ für ein minimales DFVS S, so muss jeder Knoten, der auf mehr als $|\mathbf{upper\_bound}| - |\mathbf{dfvs}|$ Kreisen liegt, auch in **dfvs** sein.

Lemma 4.1. *Im Gegensatz zur SHORTCUT Reduktion können hier alle Knoten, die auf mehr als k Kreisen liegen, zunächst markiert und am Ende der Reduktion alle auf einmal gelöscht werden.*

Beweis. Annahme: Ein Konten v wird aus der Instanz I entfernt und zum aktuellen **dfvs** hinzugefügt, sobald er von der Reduktion gefunden wird. Er liegt also auf mindestens $k+1$ knotendisjunkten Kreisen. Sei u ein von v verschiedener Knoten, der von der Reduktion hätte markiert werden können, also vor dem Entfernen von v auf mindestens $k+1$ knotendisjunkten Kreisen in I lag.

Sei $I' = I \setminus v$ und $\mathbf{dfvs}' = \mathbf{dfvs} \cup v$. Die Reduktion kann mit I' und $\mathbf{dfvs}'$ wieder ausgeführt werden. Als das neue k ergibt sich $k' = |\mathbf{upper\_bound}| - |\mathbf{dfvs}'| = k - 1$. Der Knoten u kann auch in dieser neuen Situation wieder gefunden werden, da das Entfernen von v maximal einen knotendisjunkten Kreis in I bricht, auf dem u liegt. Jeder Knoten, der die Bedingungen der Reduktion erfüllt, wird also immer auch gefunden, wenn die Reduktion iterativ angewandt wird. Es können also alle Knoten, welche die Bedingung erfüllen, auf einmal gelöscht werden. □

Diese Reduktion sucht für jeden Knoten einen Fluss der maximalen Göße k in $O(k|E|)$. Da durch jeden Knoten, mit Ausnahme von s und t, nur ein maximaler Fluss von 1 fließen kann und keine Mehrfachkanten existieren (Invariante 1), wird k durch $|V|$ begrenzt. Daraus ergibt sich eine Laufzeit von $O(|V|^2|E|)$.

4.4. Aufteilung in Komponenten

Die starken Zusammenhangskomponenten werden mit Tarjans Algorithmus für starke Zusammenhangskomponenten (vgl. Abschnitt 2.2.2) berechnet. Die Implementierung berechnet dabei ein Array **scc**, das für jeden Knoten $v \in V(I)$ dessen starke Zusammenhangskomponente $\mathbf{scc}[v] \in \mathbb{Z}_i$ beinhaltet, wobei i die Anzahl der starken Zusammenhangskomponenten bezeichnet. Anschließend wird die größte starke Zusammenhangskomponente gefunden und deren Farbe c mit der Farbe $i-1$ getauscht. Auf diese Weise wird die größte starke Zusammenhangskomponente zuletzt und damit mit den meisten Informationen über *upper bound* und die aktuelle Lösung berechnet.
Nach dem Lösen des DFVS-Problems für eine starke Zusammenhangskomponente wird überprüft, ob die aktuelle Lösung bereits größer ist als das *upper bound*, um bereits hier die Berechnung abzubrechen. Um eine starke Zusammenhangskomponente C der Farbe c aus einer Instanz I zu einer eigenen Instanz zu machen, geht die Implementierung wie folgt vor: C wird zunächst als Kopie von I initialisiert. Alle Knoten $v \in V(I)$ mit $\mathbf{coloration}[v] = c$ werden aus I entfernt (vgl. Abschnitt 4.1.2). Alle Knoten $u \in V(C)$ mit $\mathbf{coloration}[u] \neq c$ werden analog dazu aus C entfernt. Für jede starke Zusammenhangskomponente wird eine Lösung berechnet, um C bereits mit einem möglichst kleinen initialen **upper bound** zu lösen (vgl. 4.2).

Das Aufteilen der Instanz erfolgt also in $O(|V||E|)$. Das Aufteilen der Instanz ist in Linearzeit möglich. Da eine Instanz allerdings immer reduziert wird, bevor sie in ihre starken Zusammenhangskomponenten aufgeteilt wird, fällt die asymptotisch nicht optimale Laufzeit hier nicht ins Gewicht.

5. Experimente

Dieser Abschnitt beinhaltet Messdaten der Implementierung des Algorithmus (vgl. Abschnitt 4). Als Testinstanzen wurden die öffentlichen Graphen der PACE-Challenge verwendet [pac22]. Diese tragen die Namen e_XXX, wobei XXX die Nummer des Graphen in dreistelliger Dezimaldarstellung beschreibt. Der öffentliche Datensatz der PACE-Challenge besteht aus den ersten 100 Graphen mit ungeraden Nummern, also aus den Graphen e_001, e_003, ..., e_199. Die Instanzen bestehen aus zwischen 512 und 131072 Knoten. Alle Messungen wurden auf einer AMD Ryzen 7 4700U 2.0GHz CPU mit 16GB RAM durchgeführt.
Abbildung 5.1 zeigt die verschiedenen Berechnungszeiten aller Instanzen, die in unter 30 Minuten gelöst wurden. Die Ausführung des Algorithmus unterschied sich hierbei in der Auswahl des Knotens zum *branching* (vgl. Abschnitt 3.3): In einer Ausführung wurde die Heuristik (1) des Knotengrads, in der anderen Ausführung die Heuristik (2) des Produkts aus Ein- und Ausgangsgrad verwendet. Zweitere führte bei 30 der 43 gelösten Instanzen zu schnelleren Ergebnissen. Im Fall der Instanz e_035 führte die Heuristik 2 mit insgesamt ca. 4,66s sogar um ca. 2,3s schneller zu einem Ergebnis als Heuristik 1. In allen Fällen, in denen Heuristik 1 zu schnelleren Ergebnissen führte als Heuristik 2, unterschieden sich die Laufzeiten um weniger als 0,022s.

5.1. Initiale Lösungen

Die Abbildungen 5.2 und 5.3 zeigen für jede Testinstanz das Ergebnis bzw. die benötigte Berechnungsdauer zweier *greedy* Lösungen (vgl. Abschnitt 4.2.1). Die Berechnungen unterscheiden sich in der Heuristik (vgl. Abschnitt 4), mit der die zu löschenden Knoten ausgewählt werden.
Die Heuristik des höchsten Knotengrads führte zwar an vielen Stellen zu besseren, aber auch oft zu langsameren Ergebnissen als die Heuristik des Produkts aus Ein- und Ausgangsgrads. Die Größe der Lösungen unterscheiden sich im Durchschnitt um ca. 8%, mit einer maximalen Differenz von ca. 48% für die Instanz e_105. Diese Differenz ist allerdings weit größer als alle anderen und kann daher als Ausreißer betrachtet werden. Für einige wenige Instanzen führte die Heuristik des Produkts aus Ein- und Ausgangsgrad hier sogar zu besseren Ergebnissen als die Heuristik des Knotengrads.
Die Berechnungsdauern unterscheiden sich im Durchschnitt um ca. 37% mit einer maximalen Differenz von ca. 72%. Die Heuristik des Knotengrads führte hier ausschließlich für die Instanz e_121 zu einer marginal schnelleren Berechnung als die Heuristik des Produkts

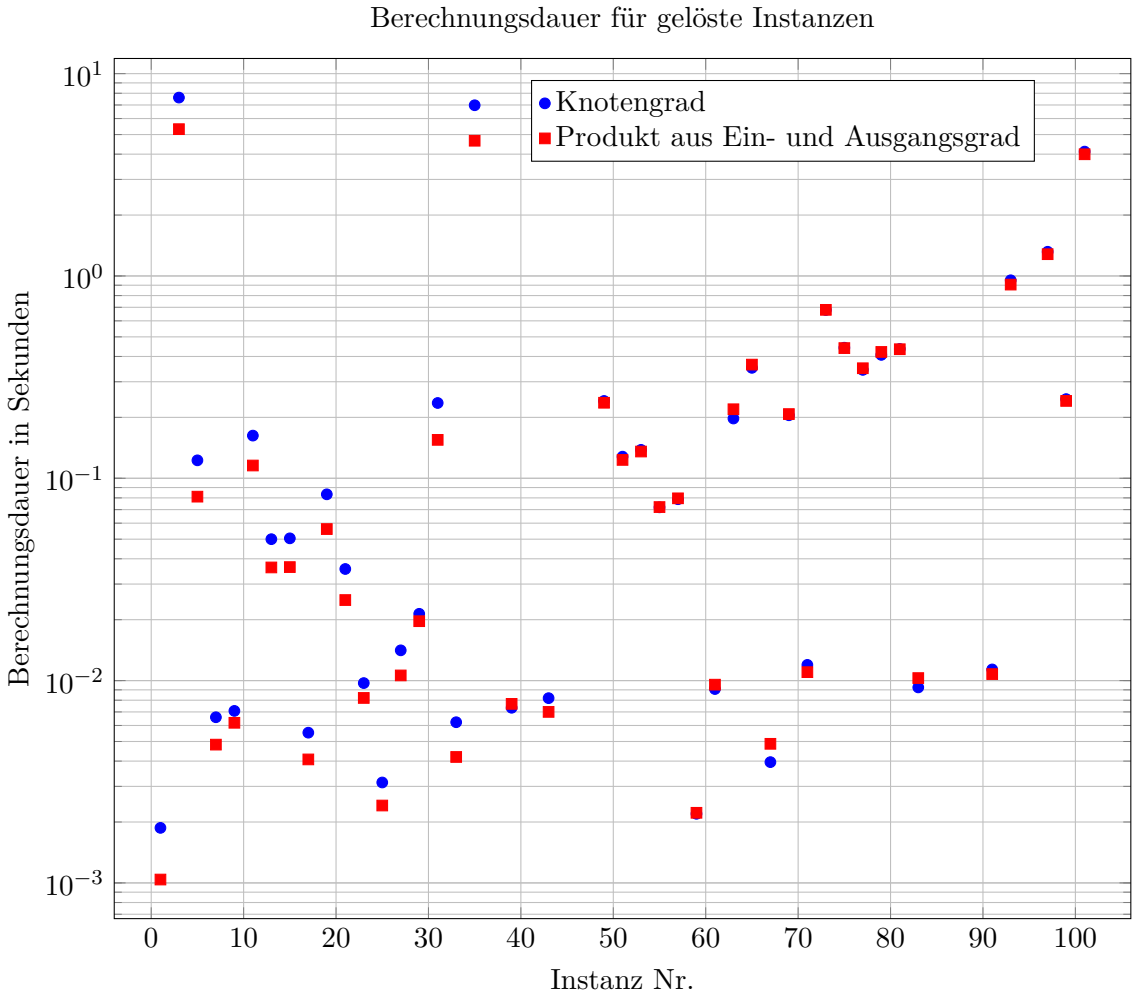


Abbildung 5.1.: Vergleich der Berechnungszeiten des Algorithmus für alle Instanzen, die in unter 30 Minuten gelöst werden konnten. In blau mit der Heuristik des Knotengrads, in rot mit der Heuristik des Produkts aus Ein- und Ausgangsgrad (vgl. Abschnitt 4).

aus Ein- und Ausgangsgrad.
Abbildung 5.4 zeigt für alle Instanzen, für die eine optimale Lösung berechnet werden konnte, den Vergleich der initialen Lösungen zu der berechneten optimalen Lösung.

5.2. Reduktion

Abbildung 5.5 zeigt für jede Testinstanz, wie viele Knoten in der reduzierten Instanz im Verhältnis zur ursprünglichen Knotenmenge vorhanden sind. Im Durchschnitt enthält die reduzierte Instanz noch ca. 46% der ursprünglichen Knotenmenge. Abbildung 5.6 zeigt analog hierzu die Verhältnisse der Knotenmengen von reduzierter und ursprünglicher Instanz mit einem Durchschnitt von ca. 44% der Kantenmenge.
Nur die Instanzen e_195 und e_147 konnten gar nicht reduziert werden.

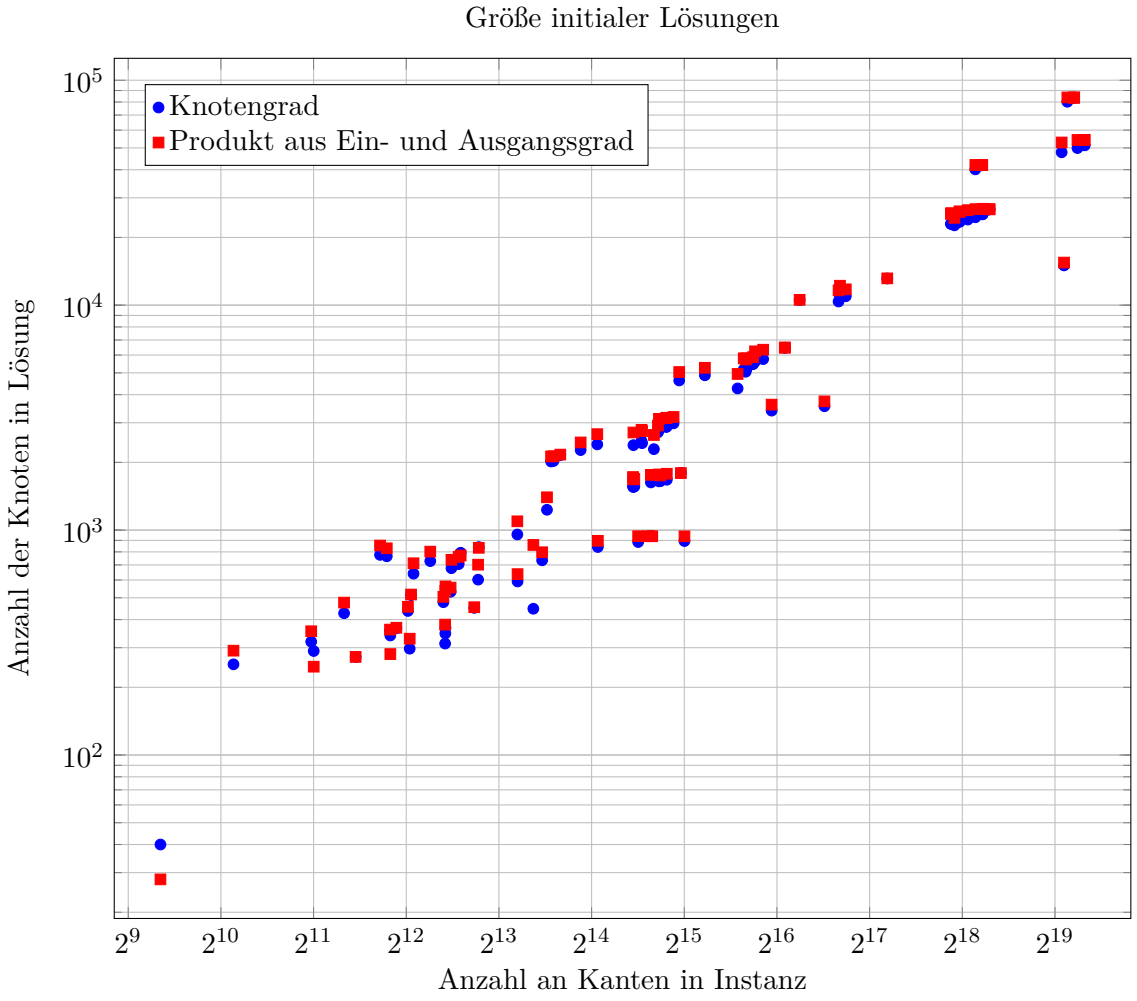


Abbildung 5.2.: Vergleich der Initialen Lösungen mit *greedy*-Ansatz (vgl. Abschnitt 4.2.1). In blau mit der Heuristik des Knotengrads, in rot mit der Heuristik des Produkts aus Ein- und Ausgangsgrad (vgl. Abschnitt 4).

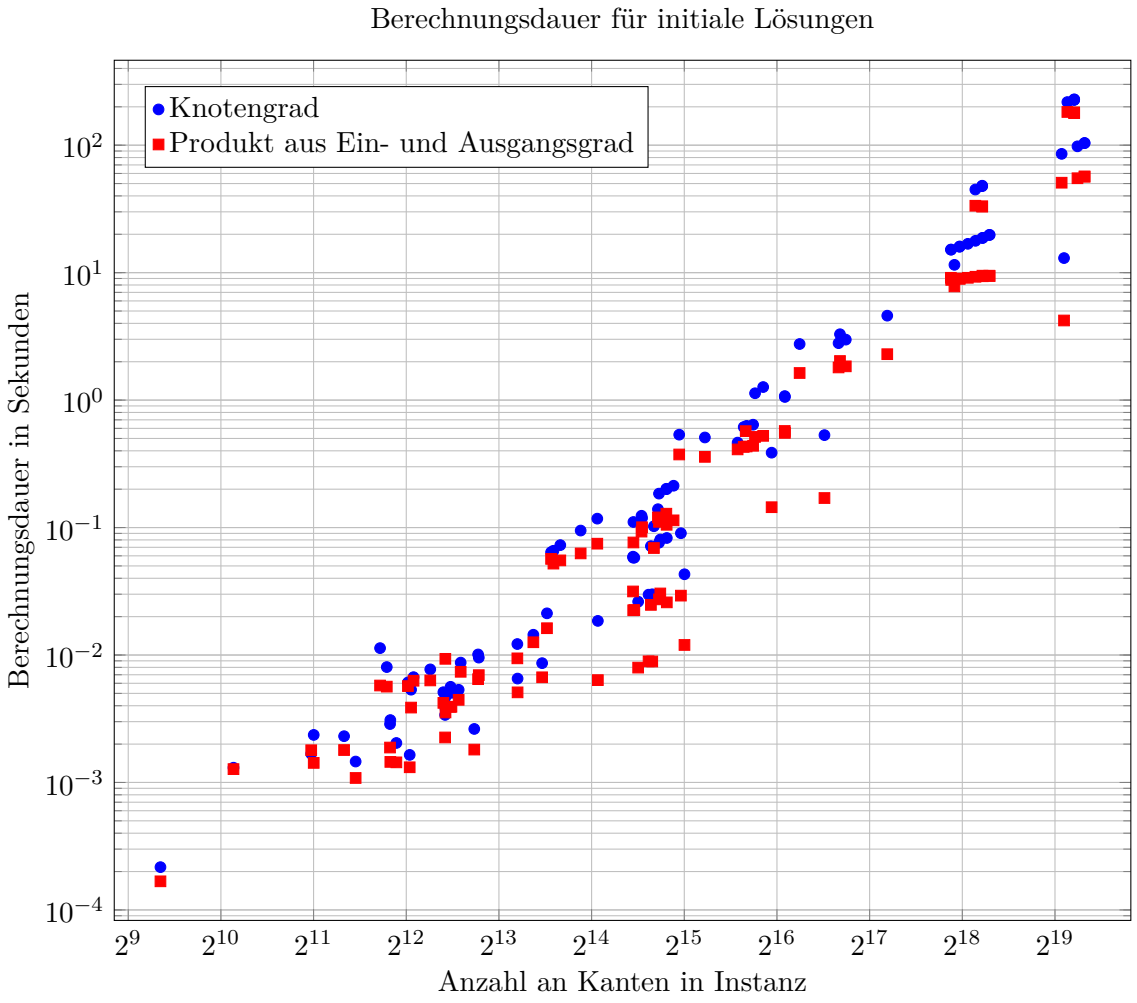


Abbildung 5.3.: Vergleich der Berechnungszeiten der Initialen Lösungen mit *greedy*-Ansatz (vgl. Abschnitt 4.2.1). In blau mit der Heuristik des Knotengrads, in rot mit der Heuristik des Produkts aus Ein- und Ausgangsgrad (vgl. Abschnitt 4).

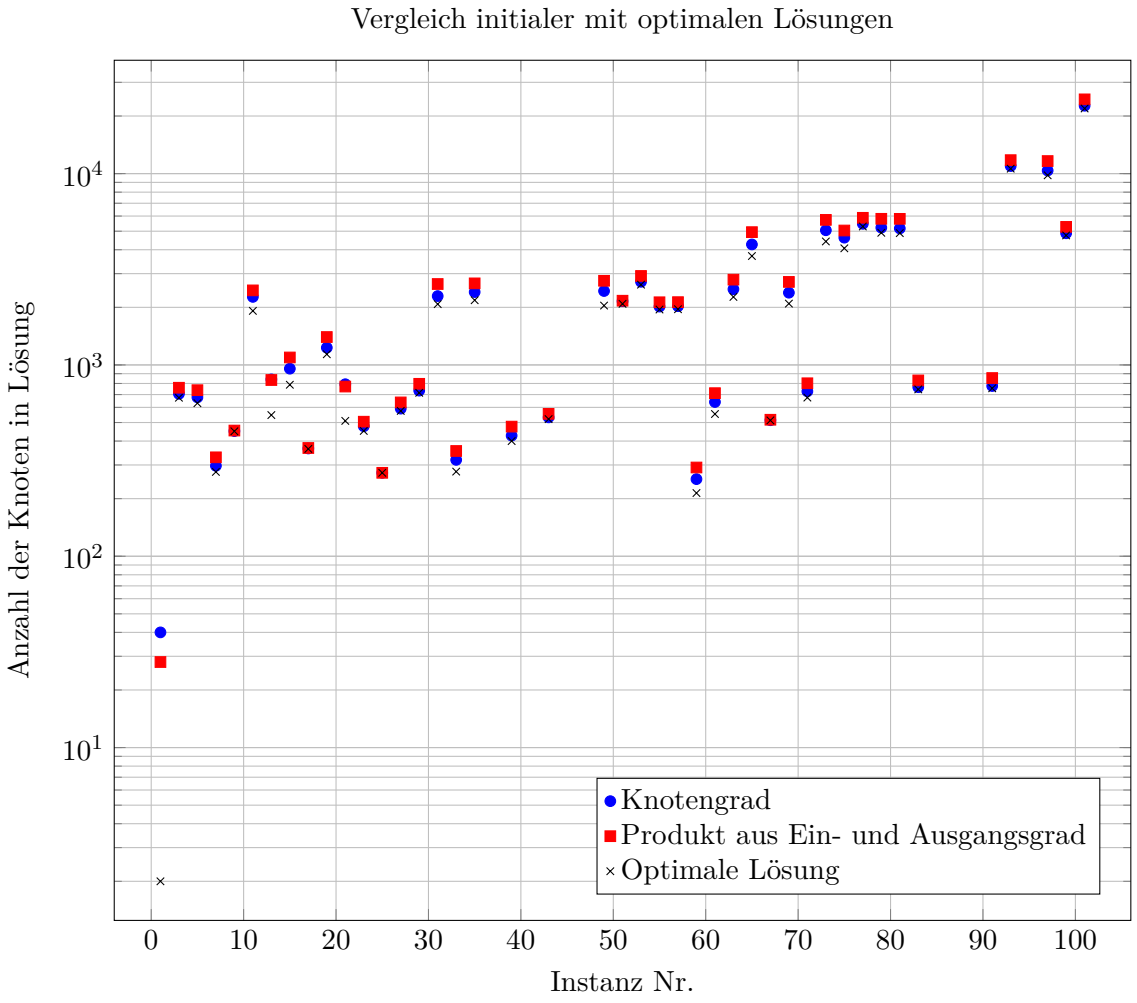


Abbildung 5.4.: Vergleich der Berechnungszeiten der Initialen Lösungen zu den entsprechenden optimalen Lösungen. In blau mit der Heuristik des Knotengrads, in rot mit der Heuristik des Produkts aus Ein- und Ausgangsgrad (vgl. Abschnitt 4), in grau optimale Lösung.

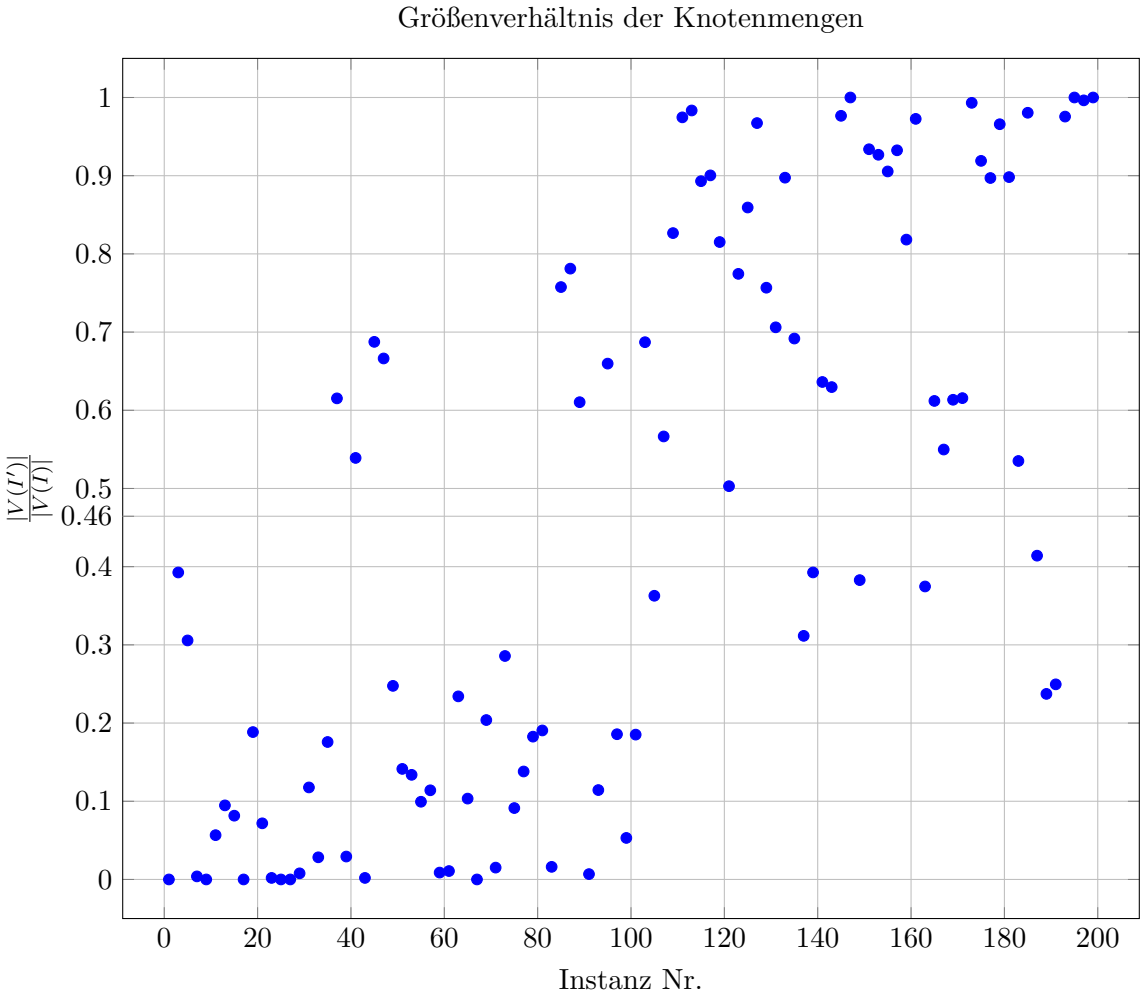


Abbildung 5.5.: Die Verhältnisse der Knotenmenge der reduzierten Instanz I' zur ursprünglichen Instanz I

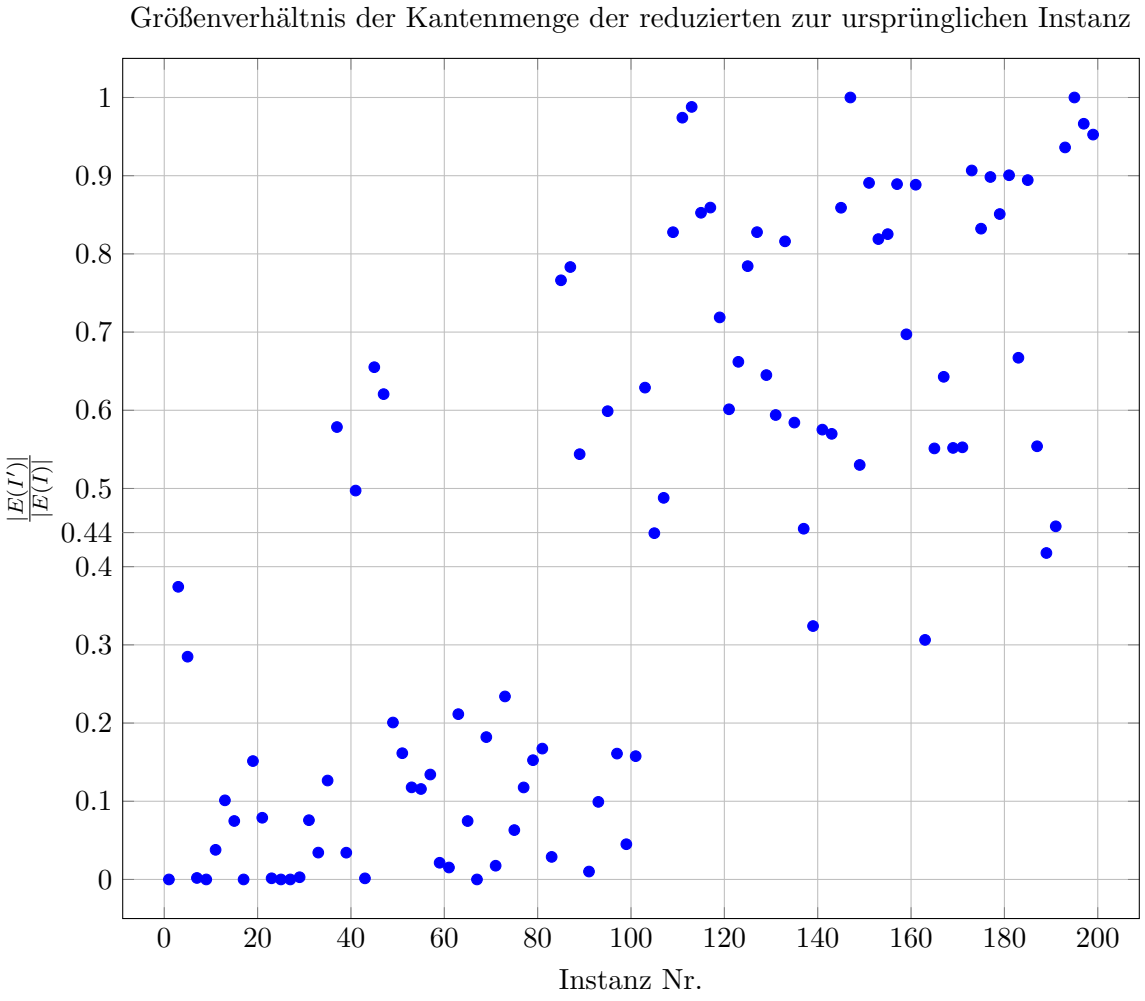


Abbildung 5.6.: Die Verhältnisse der Kantenmenge der reduzierten Instanz I' zur ursprünglichen Instanz I

6. Fazit

In dieser Arbeit wurde ein *Branch&Bound* Algorithmus zum Lösen des *Directed Feedback Vertex Set* Problems entwickelt und implementiert. Die Hauptmerkmale des Algorithmus sind die Reduktion der Probleminstanzen sowie das unabhängige Lösen starker Zusammenhangskomponenten. Für die Implementierung des Algorithmus sowie der Reduktion wurden Messungen vorgenommen und präsentiert. Innerhalb von 30 Minuten konnte die Implementierung 43 der 100 verwendeten Testinstanzen vollständig lösen. Der Algorithmus wurde für die PACE-Challenge 2022 eingereicht [pac22]. Da die PACE-Challenge zum aktuellen Zeitpunkt noch nicht ausgewertet ist, kann an dieser Stelle leider weder ein Vergleich mit anderen Teilnehmenden noch eine Evaluation der eigenen Abgabe erfolgen. Der Algorithmus findet, auch wenn er nicht vollständig ausgeführt wird, immer eine Lösung. Diese ist die bisher beste gefundene. Eine solche Lösung besitzt allerdings keine Gütegarantien und kann alle Knoten der Instanz beinhalten. Wird statt einer optimalen also nur eine mögliche Lösung benötigt, so kann der Algorithmus zwar verwendet werden, ein zu diesem Zweck entwickelter heuristischer Algorithmus ist diesem allerdings vorzuziehen.

Der Algorithmus bietet Potential, um weiter optimiert zu werden. An erster Stelle sei hier die Parallelisierung genannt: Da starke Zusammenhangskomponenten ohnehin unabhängig voneinander gelöst werden, bietet sich hier die Aufspaltung in einen Prozess für jede Komponente an. Auch die Reduktion einer Instanz lässt sich auf verschiedene Weisen parallelisieren. Zum Einen könnten alle Reduktionsregeln in jeder Iteration der Reduktion gleichzeitig geprüft werden. Hierbei werden zu löschende Knoten bzw. Kanten von jeder Regel zunächst nur markiert und dann gleichzeitig gelöscht. Das Vereinigen und Ignorieren von Knoten müsste an dieser Stelle allerdings auf andere Weise gelöst werden. Um die Reduktion auf eine andere Art zu parallelisieren, kann jeder Knoten bzw. jede Kante gleichzeitig auf eine Reduktionsregel überprüft werden. Hierfür müssen ggf. aber zunächst einige Datenstrukturen, wie beispielsweise $\Pi(I)$, aufgebaut werden.
Auch könnten neben der Parallelisierung noch weitere Experimente für die Reduktionsregel **FLOWER** (vgl. Abschnitt 7) zu einer Optimierung führen. Diese Reduktion ist die asymptotisch langsamste aller hier implementierten Reduktionen. Daher könnte experimentell ein minimales k' ermittelt werden, so dass die Reduktion nur angewendet wird, wenn $k < k'$ gilt. Ein gut gewähltes k' könnte den Algorithmus beschleunigen, indem **FLOWER** nur auf Instanzen angewendet wird, die schon klein genug sind, um eine schnelle Ausführung zu garantieren. Auch steigt für ein kleines k die Wahrscheinlichkeit, dass ein Knoten tatsächlich auf mehr als k Kreisen liegt. Die Reduktion ist für kleines k also mit höherer

Wahrscheinlichkeit erfolgreich.
Des Weiteren seien hier noch potentielle Optimierungsmöglichkeiten bei den Heuristiken, der Berechnung von initialen Lösungen sowie dem *lower bound* erwähnt: Die eben genannten Konzepte beeinflussen maßgeblich die Laufzeit des Algorithmus. Hier könnten also weiterhin verschiedene Ansätze implementiert und experimentell getestet werden. Beispielsweise könnte der *greedy* Algorithmus zur Berechnung initialer Lösungen noch optimiert werden, indem zunächst eine *Priority Queue* für die Löschreihenfolge der Knoten erstellt wird. Mit Hilfe dieser *Priority Queue* kann anschließend mit dem Prinzip der binären Suche die Anzahl von Knoten ermittelt werden, die gelöscht werden müssen, um die Instanz azyklisch zurück zu lassen.
Soll der Algorithmus auf Graphen mit spezieller Struktur angewendet werden, so kann natürlich versucht werden, sich diese Struktur in allen bisher genannten Bereichen zu Nutze zu machen. Beispielsweise, in dem spezielle Heuristiken oder Reduktionen gefunden werden.

Literaturverzeichnis

[AM94] Pranav Ashar und Sharad Malik: *Implicit computation of minimum-cost feedback-vertex sets for partial scan and other applications.* In: *31st Design Automation Conference*, Seiten 77–80. IEEE, 1994.

[BF06] Rudolf Berghammer und Alexander Fronk: *Exact computation of minimum feedback vertex sets with relational algebra.* Fundamenta Informaticae, 70(4):301–316, 2006.

[BYGNR98] Reuven Bar-Yehuda, Dan Geiger, Joseph Naor und Ron M Roth: *Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and Bayesian inference.* SIAM journal on computing, 27(4):942–959, 1998.

[CA90] K T Cheng und Vishwani D. Agrawal: *A partial scan method for sequential circuits with feedback.* IEEE Transactions on Computers, 39(4):544–548, 1990.

[CBA95] Srimat T Chakradhar, Arun Balakrishnan und Vishwani D Agrawal: *An exact algorithm for selecting partial scan flip-flops.* Journal of Electronic Testing, 7(1):83–93, 1995.

[CLL+08] Jianer Chen, Yang Liu, Songjian Lu, Barry O'sullivan und Igor Razgon: *A fixed-parameter algorithm for the directed feedback vertex set problem.* In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*, Seiten 177–186, 2008.

[Coo71] Stephen A Cook: *The complexity of theorem-proving procedures.* In: *Proceedings of the third annual ACM symposium on Theory of computing*, Seiten 151–158, 1971.

[EK72] Jack Edmonds und Richard M Karp: *Theoretical improvements in algorithmic efficiency for network flow problems.* Journal of the ACM (JACM), 19(2):248–264, 1972.

[FF56] Lester Randolph Ford und Delbert R Fulkerson: *Maximal flow through a network.* Canadian journal of Mathematics, 8:399–404, 1956.

[FWY09] Rudolf Fleischer, Xi Wu und Liwei Yuan: *Experimental study of FPT algorithms for the directed feedback vertex set problem.* In: *European Symposium on Algorithms*, Seiten 611–622. Springer, 2009.

[Gö22a] Ruben Götz: *DFVS Algo*, 2022. `https://gitlab.com/rubenGoetz/dfvs-algo`, besucht: 2022-06-17.

[Gö22b] Ruben Götz: *Ruben Bachelor Thesis (Code)*, Juni 2022. `https://doi.org/10.5281/zenodo.6604728`.

[GS76] Georges Gardarin und Stefano Spaccapietra: *Integrity of Data Bases: A General Lockout Algorithm with Deadlock Avoidance.* In: *IFIP Working*

Conference on Modelling in Data Base Management Systems, Seiten 395–412, 1976.

[HU72] Matthew S Hecht und Jeffrey D Ullman: *Flow graph reducibility.* In: *Proceedings of the fourth annual ACM symposium on Theory of computing*, Seiten 238–250, 1972.

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

[KP14] Tomasz Kociumaka und Marcin Pilipczuk: *Faster deterministic feedback vertex set.* Information Processing Letters, 114(10):556–560, 2014.

[LJ00] Hen Ming Lin und Jing Yang Jou: *On computing the minimum feedback vertex set of a directed graph by contraction operations.* IEEE Transactions on computer-aided design of integrated circuits and systems, 19(3):295–307, 2000.

[LRS16] Daniel Lokshtanov, MS Ramanujan und Saket Saurabh: *A linear time parameterized algorithm for directed feedback vertex set.* arXiv preprint arXiv:1609.04347, 2016.

[pac22] *PACE 2022*, 2022. `https://pacechallenge.org/2022/`, besucht: 2022-06-17.

[Raz07] Igor Razgon: *Computing minimum directed feedback vertex set in O*(1.9977 n).* In: *Theoretical Computer Science*, Seiten 70–81. World Scientific, 2007.

[SGG06] Abraham Silberschatz, Peter B Galvin und Greg Gagne: *Operating system concepts.* John Wiley & Sons, 2006.

[Sha81] Micha Sharir: *A strong-connectivity algorithm and its applications in data flow analysis.* Computers & Mathematics with Applications, 7(1):67–72, 1981.

[Sun71] Yngve Sundblad: *The Ackermann function. a theoretical, computational, and formula manipulative study.* BIT Numerical Mathematics, 11(1):107–119, 1971.

[SW75] G Smith und R Walford: *The identification of a minimal feedback vertex set of a directed graph.* IEEE Transactions on Circuits and Systems, 22(1):9–15, 1975.

[Tar72] Robert Tarjan: *Depth-first search and linear graph algorithms.* SIAM journal on computing, 1(2):146–160, 1972.

[TS86] Chia Jeng Tseng und Daniel P Siewiorek: *Automated synthesis of data paths in digital systems.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 5(3):379–395, 1986.

[TVL84] Robert E Tarjan und Jan Van Leeuwen: *Worst-case analysis of set union algorithms.* Journal of the ACM (JACM), 31(2):245–281, 1984.

Anhang

A. Verworfene Ideen

Während der Entwicklung des Algorithmus wurden einige Ideen in Betracht gezogen, jedoch aus unterschiedlichen Gründen wieder verworfen. Im Folgenden werden die interessantesten dieser Ideen sowie die Gründe gegen deren Implementierung aufgeführt.

A.1. Intervall-Reduktion

Die Intervall-Reduktion ist eine Reduktionsregel, basierend auf der *Flow*-Graph Reduktion nach [HU72]. Ein *Flow*-Graph bezeichnet hier einen gerichteten endlichen Graphen G mit einem Startknoten i, von dem aus jeder andere Knoten $v \in V(G)$ erreichbar ist.
Die *Flow*-Graph Reduktion sortiert $V(G)$ in disjunkte Knotenmengen (genannt Intervalle) mit eindeutigen Repräsentanten. Für jedes dieser Intervalle gilt: 1) Jeder Kreis innerhalb eines Intervalls beinhaltet dessen Repräsentanten und 2) jede Kante, die von außerhalb in ein Intevall geht, geht in den Repräsentanten des Intervalls. Um ein DFVS für einen *Flow*-Graphen zu berechnen, können also alle Knoten eines Intervalls in dessen Repräsentanten vereinigt werden (vgl. Abschnitt 4.1.3). Jede starke Zusammenhangskomponente erfüllt die Bedingungen ein Flowgraph zu sein. Hier kann jeder Knoten als Startknoten gewählt werden. Die Intervall-Reduktion kann im Algorithmus also für jede starke Zusammenhangskomponente angewandt werden.
In Reduzierten Graphen hat jeder Knoten mindestens einen Ein- sowie Ausgangsgrad von 2, wegen den Reduktionsregeln **IN0**, **OUT0**, **LOOP**, **IN1** und **OUT1**. Die Intervallbildung beginnt mit einem Knoten r als Repräsentanten und fügt jeden Knoten v in das Intervall von r ein, dessen Vorgänger alle im Intervall von r sind. In einer reduzierten starken Zusammenhangskomponente findet die Intervall-Reduktion also für jeden Knoten ein Intervall und reduziert den Graphen daher nicht weiter.

A.2. Prioritätsbasierter Aufbau des Suchbaums

Der Algorithmus baut den Suchbaum in Form einer Tiefensuche auf. Eine Alternative dazu wäre es, zu jedem Zeitpunkt den Ast mit dem aktuell kleinsten **dfvs** weiter zu berechnen. Auf diese Weise wird immer in Richtung der aktuell kleinsten möglichen Lösung gesucht. Dies erfordert zusätzlichen Speicher, da auf diese Weise viele Zustände gleichzeitig gehalten werden müssen. Allerdings hat eine frühe Version dieses Algorithmus bereits das Speicherlimit der testweisen Einreichung der PACE-Challenge (vgl. [pac22]) überschritten. Eine prioritätsbasierte Suche wäre für diese Abgabe also unrealistisch.