Untersuchung von Algorithmen für transitive Reduktionen und minimale Äquivalenzgraphen

Diplomarbeit

vorgelegt von
Christian Kasper

Aufgabenstellung
Prof. Dr. Rudolf Berghammer

Betreuung
Prof. Dr. Rudolf Berghammer
Thorsten Hoffmann

Christian-Albrechts-Universität zu Kiel
Technische Fakultät
Institut für Informatik und Praktische Mathematik

30. April 2001

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Graphentheorie ist ein wichtiges Forschungsgebiet der Informatik und Mathematik. Ihre Ergebnisse ermöglichen es, Probleme der Praxis durch geeignete Abstraktion und Modellierung zu lösen. Die Entwicklung von Algorithmen muß aber sowohl in der Theorie erfolgen, um die Korrektheit zu sichern, als auch in der Praxis, um schnelle und gute Lösungsmöglichkeiten zu erhalten.

Bei der Betrachtung von Graphen ist die Erreichbarkeit zwischen den Knoten eine wichtige Eigenschaft. Die Minimierung von Graphen bei Erhaltung aller Erreichbarkeitsinformationen führt auf den Problemkreis der minimalen Äquivalenzgraphen und transitiven Reduktionen. Diese speziellen Untergraphen bestehen nur aus „notwendigen“ Pfeilen und lösen das Problem anzahl- bzw. inklusionsminimal.

Da die Berechnung eines minimalen Äquivalenzgraphen $\mathcal{NP}$-hart ist, für die Anwendung aber nur praktikable Verfahren von Interesse sind, beschränkt sich die Arbeit auf die Entwicklung von Algorithmen zur Berechnung transitiver Reduktionen und Approximationen minimaler Äquivalenzgraphen. Diese werden sowohl theoretisch entwickelt als auch praktisch miteinander verglichen.

Die Anwendungsmöglichkeiten der theoretischen Betrachtungen sind vielfältig. So können Speicherplatzminimierung, Ablaufprobleme, Ressourcenverwaltung und die Möglichkeit übersichtlicherer und damit schönerer Zeichnungen von Graphen genannt werden.

Im Anschluß an eine Einführung der Grundbegriffe der Graphentheorie und anderer Grundlagen in Kapitel 2, wird im Kapitel 3 speziell in den Themenbereich der minimalen Äquivalenzgraphen eingeführt.
In den Kapiteln 4 bis 6 werden drei verschiedene Ansätze zur Berechnung transitiver Reduktionen bzw. Approximationen minimaler Äquivalenzgraphen entwickelt.
Diese werden im 7. Kapitel anhand von Testreihen praktisch verglichen, um die theoretischen Ergebnisse auf ihre Übertragbarkeit in die Praxis zu überprüfen. Die den Tests zugrundeliegenden Implementierungen finden sich im Anhang.
Den Abschluß bildet Kapitel 8 mit einer Zusammenfassung der Arbeit und einem Ausblick.

Kapitel 2

Grundlagen

Dieser Abschnitt dient dazu, die Begriffe zu klären, die im Verlaufe dieser Arbeit verwendet werden. Wir beschreiben zuerst eine Beispielsprache, die wir im Laufe der Arbeit benutzen wollen. Im folgenden werden die wichtigsten Begriffe der Relationentheorie [27] und der Graphentheorie [13, 18] sowie ein wichtiges Durchlaufverfahren auf Graphen, die Tiefensuche [23, 31, 6, 8], erklärt. Dabei wird auch speziell auf die Verbindung von Graphen und Relationen eingegangen.
Außerdem setzen wir die Grundbegriffe der Mengentheorie voraus, die an dieser Stelle nicht näher beschrieben werden.

2.1 Beispielsprache

Wir wollen in dieser Arbeit Algorithmen entwickeln. Um eine einheitliche Grundlage zu haben, führen wir hier eine beispielhafte Sprache ein. Dabei sind auch umgangssprachliche Beschreibungen zugelassen.
Für die formale Programmentwicklung stellen wir die folgenden Konstrukte zur Verfügung, die PASCAL-ähnlich notiert werden und die natürlichen Bedeutungen haben. Wir nehmen mit b eine boolesche Bedingung, mit p_1 und p_2 zwei Programme, mit $j, l, k \in \mathbb{Z}$ ganze Zahlen und mit M eine endliche Menge an.

- **if** b **then** p_1 **else** p_2 **end**;
- **while** b **do** p_1 **end**;
- **repeat** p_1 **until** b;
- **for** $i = j$ **to** l **by** k **do** p_1 **end**;
- **forall** $x \in M$ **do** p_1 **end**;

Außerdem nehmen wir die Gleichheitsoperation „=“ und die (kollaterale) Zuweisung „:=“ an. Dabei unterscheiden wir zwischen Prozeduren und Funktionen (**procedure**, **function**). Als Beispiel betrachte z.B. die in Abschnitt 2.5 vorgestellte Tiefensuche in den Tabellen 2.1 und 2.2.

2.2 Relationentheorie

Für gegebene Mengen $X, Y \neq \emptyset$ wird die Menge aller Relationen mit Definitionsbereich X und Wertebereich Y mit $[X \leftrightarrow Y]$ bezeichnet. Dabei sind die üblichen Operationen auf Relationen in folgender Notation gegebenen, wobei $R, S \in [X \leftrightarrow Y]$ und $T \in [Y \leftrightarrow Z]$:
die Transposition R^T, die Negation $\overline{R}$, die Vereinigung $R \cup S$, der Schnitt $R \cap S$, die Komposition (das Produkt) RT und die Inklusion $R \subseteq S$.
Außerdem seien die konstanten Relationen L, O und I gegeben.
Falls der Definitionsbereich und Wertebereich identisch sind, nennen wir eine Relation homogen, sonst heterogen. Eine homogene Relation R heißt reflexiv, falls $\mathsf{I} \subseteq R$ gilt, transitiv, falls $RR \subseteq R$ gilt. Die transitive Hülle R^+ wird durch $\bigcup_{i \geq 1} R^i$ definiert, die reflexiv-transitive Hülle von R durch $R^* := \bigcup_{i \geq 0} R^i$.
Die Beschreibung einer Teilmenge $U \subseteq X$ erfolgt mit Hilfe des Vektors $v \in [X \leftrightarrow X]$ für den $v = v\mathsf{L}$ gilt. Ein Vektor kann auch immer als heterogene Relation mit einem einelementigen Wertebereich aufgefaßt werden. Eine einelementige Teilmenge eines nichtleeren Vektors wird Punkt genannt. Ein Punkt p kann durch

$$p = p\,\mathsf{L} \wedge pp^T \subseteq \mathsf{I} \wedge p \neq \mathsf{O}$$

charakterisiert werden. Allgemein werden einelementige Teilmengen Atome genannt. Für ausführlichere Definitionen der Relationentheorie sei auf Schmidt und Ströhlein [27] verwiesen.

2.3 Graphentheorie

Ein ungerichteter Graph ist ein Paar $G = (V, E)$, wobei V eine nichtleere Menge und E eine Teilmenge von $\{\{x, y\} \mid x, y \in V, x \neq y\}$ ist. Die Elemente von V heißen Knoten, die von E Kanten von G.
In einem ungerichteten Graph $G = (V, E)$ definieren wir:

- Eine Folge $p = (v_0, v_1, \ldots, v_n)$ mit $\{v_{i-1}, v_i\} \in E$ für alle $i \in \{1, \ldots, n\}$ bezeichnen wir als Pfad von v_0 nach v_n in G der Länge n.
- G heißt zusammenhängend, falls es für alle $u, v \in V$ einen Pfad von u nach v gibt.
- G nennen wir Baum, falls G zusammenhängend ist, und $|V| = |E| + 1$ gilt.

Sei die Relation $\sim$ auf V wie folgt definiert:

$$u \sim v \iff \exists \text{ Pfad } p = (u, \ldots, v) \text{ in } G.$$

$\sim$ ist eine Äquivalenzrelation auf V. Seien $V_1, \ldots, V_m$ die Äquivalenzklassen von $\sim$ und $E_i := \{\{x, y\} \mid \{x, y\} \in E \wedge x, y \in V_i\}$ für alle $i \in \{1, \ldots, m\}$. Als Folge daraus bilden die E_i eine Partition von E. Die Graphen $G_i := (V_i, E_i)$ für $i \in \{1, \ldots, m\}$ heißen die Zusammenhangskomponenten von G.

Ein gerichteter Graph ist ein Paar $G = (V, E)$, wobei V eine nichtleere Menge und E eine Teilmenge von $\{(x, y) \mid x, y \in V\} = V \times V$ ist. Die Elemente von E heißen Pfeile, und für $e = (u, v) \in E$ nennen wir u Anfangspunkt und v Endpunkt des Pfeiles e. Wir nennen u Vorgänger von v bzw. v Nachfolger von u.
Ein Graph ist endlich, wenn $|V|$ endlich ist.
Im folgenden werden gerichtete, endliche Graphen abkürzend als Graphen bezeichnet.

Analog zu ungerichteten Graphen definieren wir in einem Graphen $G = (V, E)$ folgendes:

- Der G zugrundeliegende ungerichtete Graph ist definiert als (V, E') mit
$$E' := \{\{x, y\} \mid (x, y) \in E \vee (y, x) \in E) \wedge x \neq y\}.$$
- Der transponierte Graph von G wird durch $G^T := (V, \{(x, y) \mid (y, x) \in E\})$ definiert.
- Eine Folge $W = (v_0, v_1, \ldots, v_n)$ mit $(v_{i-1}, v_i) \in E$ für alle $i \in \{1, \ldots, n\}$ heißt Weg von v_0 nach v_n in G der Länge n.
- Die Existenz eines Weges von x nach y wird mit $x \xrightarrow[G]{*} y$ bezeichnet. Falls dieser mindestens die Länge 1 hat, schreiben wir $x \xrightarrow[G]{+} y$, für einen Pfeil (x, y) notieren wir $x \xrightarrow[G]{} y$.
- Ein Weg $W = (w_0, \ldots, w_n)$ wird als einfach bezeichnet, falls seine Knoten paarweise verschieden sind, wobei aber $w_0 = w_n$ gelten darf.
- Einen einfachen Weg $W = (u, \ldots, u)$ mit mindestens der Länge 1 nennen wir Kreis.
- G heißt kreisfrei, falls es in G keinen Kreis gibt.
- Einen Pfeil $e = (x, x)$ bezeichnen wir als Schlinge, einen Graphen ohne Schlingen nennen wir schlingenfrei.
- G heißt zusammenhängend, falls der zugrundeliegende ungerichtete Graph zusammenhängend ist.
- Sind $V_1, \ldots, V_s$ die Knotenmengen der Zusammenhangskomponenten des zugrundeliegenden ungerichteten Graphen und definieren wir für $i \in \{1, \ldots, s\}$ die Graphen $E_i := E \cap (V_i \times V_i)$, so sind $G_i := (V_i, E_i)$ die Zusammenhangskomponenten von G für $i \in \{1, \ldots, s\}$.

- G nennen wir Baum, wenn es einen ausgezeichneten Knoten $v \in V$ gibt, so daß für alle $w \in V$ ein Weg von v nach w existiert, und $|V| = |E| + 1$ gilt.
- G heißt Wurzelbaum, falls es einen ausgezeichneten Knoten $v \in V$ gibt, so daß für alle $w \in V$ ein Weg von w nach v existiert, und $|V| = |E| + 1$ gilt.
- G bezeichnen wir als Wald, falls seine Zusammenhangskomponenten Bäume sind.
- G heißt stark zusammenhängend, falls es für alle $u, v \in V$ einen Weg von u nach v gibt.
- Für einen Knoten $v \in V$ definieren wir die Menge der Nachfolger bzw. Vorgänger durch $succ(v, G) := \{x \in V \mid (v, x) \in E\}$ und $pred(v, G) := \{x \in V \mid (x, v) \in E\}$. Die Kardinalitäten dieser Mengen nennen wir Ausgangs- bzw. Eingangsgrad von v.
- Der ausgezeichnete Knoten $v \in V$ in einem Baum heißt Wurzel.

  Für ihn gilt $pred(v, G) = \emptyset$. Er wird allgemein mit *root* bezeichnet. Ein Knoten $v \in V$ bezeichnen wir als Blatt, falls $succ(v, G) = \emptyset$ gilt, und inneren Knoten, falls v kein Blatt ist.
- Der transponierte Wurzelbaum ist ein Baum im transponierten Graphen. Daher werden Wurzel, Blatt und innerer Knoten analog definiert.
- Ein Graph $H = (V, F)$ mit $F \subseteq E$ wird Untergraph von G genannt.
- Die Größe des Graphen G ist die Anzahl seiner Knoten und wird mit $|V|$ bezeichnet.
- Ein Kreis der Länge $|V|$ durch alle Knoten von V heißt Hamiltonkreis.
- Mit $V(G)$ bzw. mit $E(G)$ bezeichnen wir die Knoten- bzw. Pfeilmenge von G.
- Die Dichte eines Graphen ist durch $\frac{|E|}{|V|^2} \in [0, 1]$ definiert.
- Der transitive Graph von G wird durch $G^+ := (V, \{(x, y) \mid x \xrightarrow[G]{+} y\})$, der reflexiv-transitive Graph durch $G^* := (V, \{(x, y) \mid x \xrightarrow[G]{*} y\})$ definiert.

Definition 2.1. *Seien $G = (V, E)$ ein Graph und $V_1, \ldots, V_k$ die starken Zusammenhangskomponenten. Dann wird der Reduktionsgraph $G' := (V', E')$ von G durch*

- $V' := \{V_1, \ldots, V_k\}$
- $E' := \{(V_i, V_j) \mid \exists\, (x, y) \in E : x \in V_i \wedge y \in V_j\}$

definiert.

Bemerkung 2.1. *Ein Reduktionsgraph ist ein homomorphes, kreisfreies Bild von G, d.h. es gibt homomorphe Abbildungen $\phi_1 : V \to V'$ und $\phi_2 : E \to E'$. Die Umkehrrelationen sind i.a. nicht eindeutig. Es sind allerdings zwei Abbildungen $\psi_1 : V' \to V$ und $\psi_2 : E' \to E$ ableitbar, für die gilt:*

- $\forall\ v \in V' : \psi_1(\phi_1(v)) = v$
- $\forall\ e \in E' : \psi_2(\phi_2(e)) = e.$

Diese Funktionen bilden jede Komponente bzw. jeden Pfeil auf einen Repräsentanten ab.

Abkürzend werden an dieser Stelle einige einfache Sprechweisen eingeführt. So sagen wir, daß v ein Knoten von G oder e ein Pfeil von G ist. Ein Knoten w heißt vom Knoten v aus erreichbar, falls es einen Weg von v nach w gibt.
Alle Operationen auf Graphen beziehen sich, wenn nicht ausdrücklich anders erwähnt, auf die Pfeilmenge, d.h. Ausdrücke wie $G \subseteq H$ für zwei Graphen G und H besagen, daß die Pfeilmenge von G Teilmenge der Pfeilmenge von H ist $(E(G) \subseteq E(H))$.
Pfeile werden im allgemeinen mit e, f und Knoten mit u, v, w, x, y, z bezeichnet.

2.4 Gemeinsamkeiten von Relationen und Graphen

Der Einfachheit halber wird die Knotenmenge eines Graphen $G = (V, E)$ oft mit der Menge $\{1, \ldots, |V|\}$, und der Graph selbst deshalb mit der Pfeilmenge identifiziert. Diese Menge E wird auch als homogene Relation $E \in [V \leftrightarrow V]$ aufgefaßt.
Wir machen in dieser Arbeit keinen Unterschied zwischen einem Graphen und der zugehörigen Relation. Deshalb werden auf E sowohl mengen- als auch relationentheoretische Operationen angewendet.
Nachfolgend zeigen wir für einen Graphen $G = (V, R)$ und die Relation R noch die Parallelen auf, die wir im folgenden synonym verwenden werden:

$$\begin{aligned}
R^* = \mathsf{L} &\iff G \text{ ist stark zusammenhängend} \\
R^+ \subseteq \bar{\mathsf{I}} &\iff G \text{ ist kreisfrei} \\
R^*_{xy} &\iff x \xrightarrow[G]{*} y \\
R^+_{xy} &\iff x \xrightarrow[G]{+} y \\
R_{xy} &\iff x \xrightarrow[G]{} y.
\end{aligned}$$

Außerdem ist $t = (V, T)$ genau dann ein Baum von G, wenn es einen Punkt p aus der Menge V gibt, für den gilt:

$$T \subseteq R \wedge p\,\mathsf{L} \subseteq T^* \wedge TT^T \subseteq \mathsf{I} \wedge T^+ \subseteq \bar{\mathsf{I}}.$$

2.5 Tiefensuche

Die Tiefensuche (engl. depth first search, DFS) ist das wichtigste Graphdurchlaufverfahren. Die Suchmethode ist dadurch charakterisiert, daß ausgehend von einem Knoten x zunächst ein Knoten $y \in succ(x, G)$ ausgewählt wird, um dort den Tiefensuchdurchlauf zu starten, bevor die restlichen Nachfolger behandelt werden.
Der Algorithmus zur Tiefensuche $G = (V, R)$ wird in den Tabellen 2.1 und 2.2 dargestellt.

```
procedure dfs(u : V, G : graph)
  colour[u] := grau;
  time := time + 1;
  d[u] := time;
  forall x ∈ succ(u, G) do
    if colour[x] = weiß
      then p[x] := u;
           dfs(x, G);
    end;
  end;
  colour[x] := schwarz;
  time := time + 1;
  f[u] := time;
```

Tabelle 2.1: Pseudocode `dfs`

```
procedure search(G = (V, R) : graph)
  forall x ∈ V do
    colour[x] := weiß;
    d[x] := 0;
    f[x] := 0;
    p[x] := nil;
  end;
  time := 0;
  forall x ∈ V do
    if colour[x] = weiß
      then dfs(x, G);
    end;
  end;
```

Tabelle 2.2: Pseudocode `search`

Wir sprechen vom Besuchen von Knoten bzw. Pfeilen, wenn diese abgearbeitet werden.
Im folgenden werden die wichtigsten Eigenschaften und die Bedeutung der Felder d, f, p und *colour* zusammengefaßt.
Das Feld *colour* stellt für jeden Knoten den Besuchszustand dar. Zu Beginn ist jeder Knoten `weiß` gefärbt, wird er das erste Mal besucht, erfolgt eine Färbung nach `grau`. Diese Farbe zeigt den Zustand *besucht, aber noch unbesuchte Nachfolger* an. Der Zustand `schwarz` bedeutet, daß der Knoten besucht ist und nur besuchte Nachfolger besitzt.
Das Feld p definiert eine Vater-Kind-Relation, die als Abbildung $p : V \to V \cup \{\texttt{nil}\}$ aufgefaßt wird. Die Felder d und f sind Zeitmarkierungen und stellen die Zeit des ersten bzw. des letzten Besuches dar.
Die Menge $T := \{(p[v], v) \mid v \in V \wedge p[v] \neq \texttt{nil}\}$ definiert einen Graphen, den DFS-Wald. Falls alle Knoten von der gewählten Wurzel aus erreichbar sind, ist der Graph ein Baum und wird DFS-Baum genannt. In diesem Fall ist die umgebende Prozedur `search` nur zur Initialisierung notwendig.
Die Komplexität des Algorithmus `search` beträgt $\mathcal{O}(|V| + |E|)$.

Durch einen DFS-Durchlauf können alle Pfeile in vier verschiedene Klassen eingeteilt werden.

Definition 2.2. *Seien $G = (V, E)$ ein Graph und $t = (V, T)$ ein DFS-Wald, erzeugt durch einen Aufruf von* `search`(G). *Dann ist E disjunkte Vereinigung von vier Mengen von Pfeilen.*

Baumpfeile	$(x, y) \in T :\iff (x, y) \in \{(p[v], v) \mid v \in V \wedge p[v] \neq \texttt{nil}\}$
Vorwärtspfeile	$(x, y) \in F :\iff (x, y) \in E \backslash T \wedge (x, y) \in T^* \wedge x \neq y$
Rückwärtspfeile	$(x, y) \in B :\iff (x, y) \in E \backslash T \wedge (y, x) \in T^*$
Querpfeile	$(x, y) \in C :\iff (x, y) \in E \backslash (T \cup B \cup F)$

Die Klassifikation der Pfeile kann auch anhand der d- und f-Felder erfolgen, was sich im folgenden Lemma ausdrückt:

Lemma 2.1. *Seien $G = (V, E)$ ein Graph und T, B, C, F die Klassifikation der Pfeile nach einem DFS-Durchlauf. Dann kann ein Pfeil $(x, y) \in E \backslash T$ wie folgt durch die Felder d und f charakterisiert werden:*

- $(x, y) \in F \iff d[x] < d[y] < f[y] < f[x]$
- $(x, y) \in B \iff d[y] \leq d[x] < f[x] \leq f[y]$
- $(x, y) \in C \iff d[y] < f[y] < d[x] < f[y]$

Der Beweis findet sich bei Tarjan [31].
Die Abbildung 2.1 zeigt ein Beispiel für einen Tiefensuchdurchlauf eines Graphen sowie die Klassifikation der Pfeile. Dabei werden außerdem die Zeitmarkierungen d (links) und f (rechts)

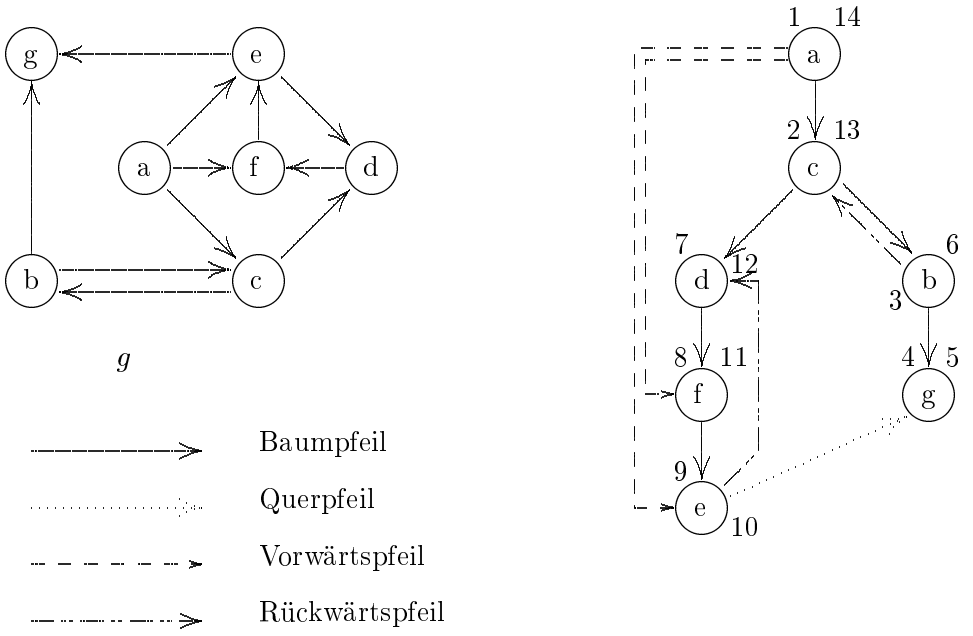


Abbildung 2.1: DFS-Durchlauf und Klassifikation

angegeben. Aus dem rechten Bild ist auch graphisch anschaulich die Bedeutungen der Pfeilklassen zu erkennen.
Mit Hilfe eines DFS-Durchlaufes wird durch den DFS-Baum T eine Ordnung auf den Knoten durch $x <_T y :\iff d[x] < d[y]$ definiert. Umgekehrt ist es möglich, durch eine gegebene Ordnung die Auswahl des nächsten Nachfolgers eindeutig zu bestimmen.

In einem Graphen $G = (V, E)$ mit DFS-Baum T wird die Abbildung $lowpoint : V \to V$ wie folgt für jeden Knoten $v \in V$ definiert:

$$lowpoint(v) := \min_{<_T}(\{v\} \cup \{y \in V \mid \exists\, z \in V : v \xrightarrow[T]{*} z \wedge (z, y) \in B\}).$$

Für einen gegebenen Knoten v betrachten wir also alle Knoten u, die von v aus auf einem (möglicherweise leeren) Weg erreicht werden können, der aus einem Weg in T gefolgt von höchstens einem Rückwärtspfeil besteht. Dann wird $lowpoint(v)$ als das Minimum der erreichbaren Knoten u gesetzt.
Im Beispiel der Abbildung 2.1 ist der *lowpoint* des Knotens f der Knoten d.

Für zwei Knoten x und y definieren wir den nächsten gemeinsamen Vorfahren (engl. lowest common ancestor, lca) bezüglich des DFS-Baumes T wie folgt:

$$lca(x, y) := \max_{<_T} \{u \mid u \xrightarrow[T]{*} y \wedge u \xrightarrow[T]{*} x\}.$$

In der Abbildung 2.1 ist der nächste gemeinsame Vorfahr der Knoten b und f der Knoten c.

Kapitel 3

Problemstellung

In diesem Kapitel wollen wir mit Hilfe der im vorherigen Abschnitt eingeführten Grundlagen, die in der Einleitung schon angesprochenen Begriffe formal einführen. Danach wollen wir wichtige Eigenschaften erarbeiten und die gestellten Aufgaben auf speziellere Probleme reduzieren. Abschließend stellen wir die wichtigsten Arbeiten zu diesem Themengebiet in ihrer zeitlichen Entwicklung dar.

3.1 Definitionen

Zunächst definieren wir die minimalen Äquivalenzgraphen und transitiven Reduktionen:

Definition 3.1. *Sei $G = (V, E)$ ein Graph. Ein Graph $H = (V, F)$ heißt anzahlminimaler Äquivalenzgraph bzw.* MEG *(engl.: minimum equivalent digraph), falls gilt:*

(i) $F \subseteq E$

(ii) $G^* = H^*$

(iii) $\forall\, I = (V, R) : R \subseteq E \wedge G^* = I^* \implies |F| \leq |R|$.

F ist also in der Menge $\{R \mid R \subseteq E \wedge E^* = R^*\}$ ein minimales Element bezüglich der Anzahl der Pfeile. Dabei bezeichne $\mathcal{OPT}(G) := |F|$ die minimale Pfeilanzahl.

Definition 3.2. *Sei $G = (V, E)$ ein Graph. Ein Graph $H = (V, F)$ heißt inklusionsminimaler Äquivalenzgraph bzw. transitive Reduktion, falls gilt:*

(i) $F \subseteq E$

(ii) $G^* = H^*$

(iii) $\forall\, I = (V, R) : R \subseteq F \wedge G^* = I^* \implies F \subseteq R$.

F ist also in der Menge $\{R \mid R \subseteq E \wedge E^* = R^*\}$ ein minimales Element bezüglich der Mengeninklusion auf der Pfeilmenge.
Eine transitive Reduktion von G wird häufig mit G_* bezeichnet.

Wir werden in dieser Arbeit immer von einem MEG (Mehrzahl: MEG's) und einer transitiven Reduktion eines Graphen sprechen.

Ein Approximationsalgorithmus zu einem Optimierungsproblem definieren wir wie folgt:

Definition 3.3. *Es sei ein Optimierungsproblem mit Universum M, zulässigem Bereich Z, linearer Ordnung $(O, \leq)$ und Zielfunktion $zf : M \to O$ gegeben. Ein Approximationsalgorithmus mit Approximationsfaktor (Güte) α liefert ein Element $n \in Z$, so daß gilt:*
$\alpha \cdot \min\{zf(m) \mid m \in Z\} \leq zf(n)$.

Bemerkung 3.1. *Wir werden im folgenden immer nur approximative Algorithmen zum Problem* MEG *betrachten, d.h. wir suchen zu einem gegebenen Graphen $G = (V, E)$ auf dem Universum $M := \{(V, F) \mid F \subseteq V \times V\}$, dem zulässigem Bereich $Z := \{H \mid H \in M \wedge G^* = H^*\}$, der linearen Ordnung $(\mathbb{N}, \leq)$ und der Zielfunktion $zf : M \to \mathbb{N} \quad (V, F) \mapsto |F|$ eine Approximation des* MEG *von G.*

Ob ein Pfeil in einem Graphen notwendig ist, läßt sich durch die Definition der Reduzierbarkeit ausdrücken:

Definition 3.4. *Sei $G = (V, E)$ ein Graph. Ein Pfeil $e = (v, w) \in E$ heißt reduzierbar (in G), falls es einen einfachen Weg W von v nach w in $(V, E\backslash\{(v, w)\})$ gibt. Dabei wird W der reduzierende Weg genannt.*

Bemerkung 3.2. *Für die Definition der Reduzierbarkeit eines Pfeiles benötigen wir nicht unbedingt einen einfachen Weg, es reicht auch ein „normaler" Weg. Die Definition ist äquivalent zu der Tatsache, daß (V, E) und $(V, E\backslash\{(v, w)\})$ dieselbe reflexiv-transitive Hülle haben.*

Allgemeines Ziel ist es nun, einen Algorithmus zu entwickeln, der für einen Graphen einen MEG berechnet.

3.2 Eigenschaften und Reduktion

Zuerst beschäftigen wir uns mit einigen grundlegenden Eigenschaften von transitiven Reduktionen und MEG's, danach werden wir die Probleme komplexitätstheoretisch einordnen und auf einfachere Probleme reduzieren.

Zunächst halten wir in folgenden Bemerkungen die Eigenschaften zur Existenz und Eindeutigkeit von MEG's und transitiven Reduktionen fest:

Bemerkung 3.3. *Eine transitive Reduktion und ein* MEG *eines Graphen existieren immer, da sie minimale Elemente von nichtleeren, endlichen Mengen sind.*

Bemerkung 3.4. *Transitive Reduktionen und* MEG*'s eines Graphen sind nicht eindeutig.*

Zu dem Graphen G (links) in Abbildung 3.1 sind sowohl G^1_* (Mitte) als auch G^2_* (rechts) transitive Reduktionen und MEG's.
Da die in der Definition 3.1.(iii) auftretende Ordnung „$\leq$" stärker ist als „$\subseteq$" in 3.2.(iii) folgt:

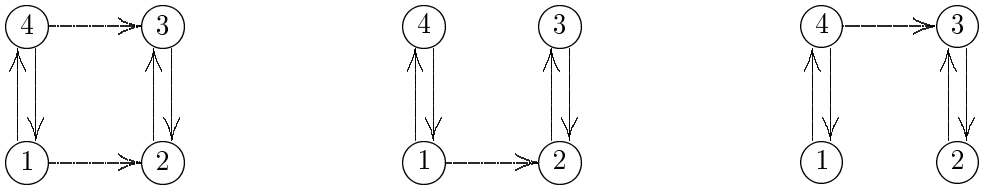


Abbildung 3.1: Verschiedene transitive Reduktionen und MEG's eines Graphen

Bemerkung 3.5. *Jeder* MEG *ist auch eine transitive Reduktion. Die Umkehrung gilt i.a. nicht.*

Bemerkung 3.6. *Zwei transitive Reduktionen können von unterschiedlicher Kardinalität sein. Zwei* MEG*'s sind laut Definition gleichmächtig.*

Abbildung 3.2 zeigt den Graphen H (links) mit den transitiven Reduktionen H^1_* (Mitte) und H^2_* (rechts). Dabei enthält H^1_* weniger Pfeile als H^2_*.

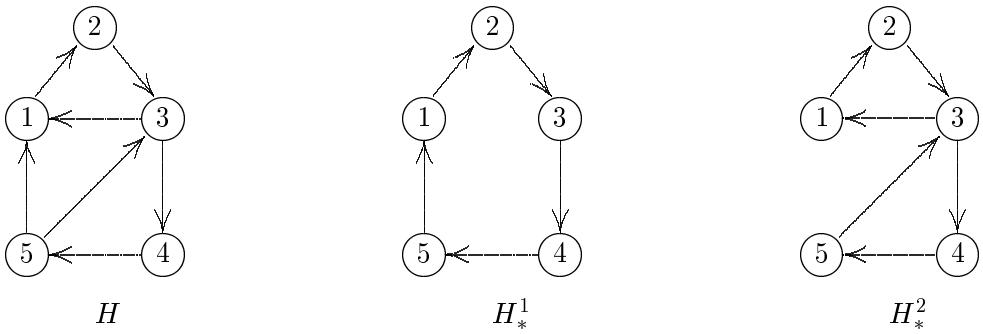


Abbildung 3.2: Verschiedene Kardinalitäten von transitiven Reduktionen

Nachdem wir nun einige grundlegende Eigenschaften erarbeitet haben, wollen wir Algorithmen zur Berechnung von MEG's und transitiven Reduktionen entwickeln. Dazu betrachten wir zunächst die Einordnung der Probleme in die großen Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ [10, 26].

Theorem 3.1. *Die Berechnung eines* MEG *eines Graphen G ist $\mathcal{NP}$-hart.*

Beweis: Zu dem Optimierungsproblem MEG betrachten wird das entsprechende „ja/nein"-Problem MEG(k), das wie folgt lautet: Gibt es zu gegebenem Graphen G und $k \in \mathbb{N}$ einen MEG von G mit weniger als k Pfeilen?
Sei $G = (V, E)$ ein stark zusammenhängender Graph und $k = |V|$. Dann ist die Lösung des Problems MEG(k) für G gerade eine Lösung für das Problem $HAMILTON$, das in G einen Hamiltonkreis sucht.
Da dieses Problem aber $\mathcal{NP}$-vollständig [10] ist, folgt, daß die Berechnung eines MEG $\mathcal{NP}$-hart ist. □

Die Berechnung eines MEG in allgemeinen Graphen ist also in polynomieller Zeit voraussichtlich nicht möglich, d.h. sie ist, falls $\mathcal{P} \neq \mathcal{NP}$ gilt, nicht effizient durchführbar.
Da wir aber an polynomiellen Algorithmen interessiert sind, müssen wir nach anderen Möglichkeiten suchen. Dabei werden wir uns in geeigneter Weise einschränken müssen.
Allgemeine Vorgehensweisen sind dabei, sich auf spezielle Graphen zu beschränken oder nicht das anzahlminimale Problem, sondern daß inklusionsminimale zu lösen. Eine weitere Herangehensweise ist es, eine Approximation zu berechnen.
Wir wollen nun auf diese Möglichkeiten eingehen und uns zuerst auf kreisfreie Graphen beschränken.

Lemma 3.1. *Sei $G = (V, E)$ ein kreisfreier Graph. Dann ist der Graph $(V, E \cap \overline{E\ E^+})$ die einzige transitive Reduktion und damit ein* MEG *von G.*

Beweis: Seien $E_* := E \cap \overline{E\ E^+}$ und $T := \{X \subseteq E \mid X^* = E^*\}$. Wir zeigen, daß E_* ein minimales Element von T ist, daß also E_* eine untere Schranke von T ist, und $E_* \in T$ gilt.

(i) Sei $X \subseteq E$ mit $X^* = E^*$ gegeben. Dann gilt:

- $E_* \cap \overline{X} = E \cap \overline{E\ E^+} \cap \overline{X} \subseteq E$.
- Da E kreisfrei ist ($E^+ \subseteq \bar{\mathsf{I}}$), trifft dies auch für X zu ($X^+ \subseteq \bar{\mathsf{I}}$).
  Aus $\mathsf{I} \cup X^+ = X^* = E^* = \mathsf{I} \cup E^+$ folgt, daß $X^+ = E^+$ und
  $E_* \cap \overline{X} = E \cap \overline{E\ E^+} \cap \overline{X} \subseteq \overline{E\ E^+} \cap \overline{X} = \overline{E^+E^+} \cap \overline{X} = \overline{X^+X^+} \cap \overline{X}$
  $= \overline{X^+X^+ \cup X} = \overline{X^+} = \overline{E^+} \subseteq \overline{E}$
  gilt.

Aus $E_* \cap \overline{X} \subseteq E$ und $E_* \cap \overline{X} \subseteq \overline{E}$, folgt $E_* \cap \overline{X} = \mathsf{O}$. Also gilt $E_* \subseteq X$. Daraus folgt, daß E_* eine untere Schranke von T ist.

(ii) Zeige nun, daß $E_* \in T$ gilt, d.h. ${E_*}^* = E^*$.

Aus $E_* = E \cap \overline{E\ E^+} \subseteq E$ und der Monotonie von * folgt ${E_*}^* = \left(E \cap \overline{E\ E^+}\right)^* \subseteq E^*$.

Es bleibt die umgekehrte Inklusion zu zeigen.

Diese weisen wir komponentenweise nach. Seien dazu $x, y \in V$ mit E^*_{xy}. Da G endlich und kreisfrei ist, gibt es einen längsten Weg $W := (x = x_0, \ldots, x_n = y)$ von x nach y, also gilt $E_{x_{i-1}\ x_i}$ für alle $i \in \{1, \ldots, n\}$.

Ann.: $\exists\ i \in \{1, \ldots, n\}$ mit $(E\ E^+)_{x_{i-1}x_i}$. Dann ist $W' = (x_0, \ldots, x_{i-1}, \ldots, x_i, \ldots, x_n)$ ein echt längerer Weg von x nach y. Widerspruch.

Es folgt, daß $\left(\overline{E\ E^+}\right)_{x_{i-1}x_i}$ für alle $i \in \{1, \ldots, n\}$ gilt. Also folgt für alle $i \in \{1, \ldots, n\}$: $\left(E \cap \overline{E\ E^+}\right)_{x_{i-1}x_i}$. Insgesamt ergibt sich $\left(E \cap \overline{E\ E^+}\right)^*_{xy}$.

Also ist E_* das einzige minimale Element der Menge T und $(V, E \cap \overline{E\ E^+})$ die einzige transitive Reduktion von G. □

Das Lemma 3.1. liefert ein Verfahren zur Bestimmung eines MEG in einem kreisfreien Graphen. Graphentheoretisch gesprochen sind also genau die Pfeile im MEG, die im Originalgraphen enthalten sind und für die es keinen anderen Weg als den Pfeil selbst gibt.
Es folgt, daß die Berechnung eines MEG in einem kreisfreien Graphen in polynomieller Zeit möglich ist. Dazu benötigen wir im wesentlichen „nur“ einen Algorithmus zur Bestimmung der reflexiv-transitiven Hülle. Für dieses Problem wurden schon eine Vielzahl von Algorithmen entwickelt, die hier nicht diskutiert werden. Wir verweisen aber auf die Arbeit von Strassen [30], in der zur Berechnung der reflexiv-transitiven Hülle die Matrizenmultiplikation verwendet wird. Der dort entwickelte Algorithmus benötigt die Laufzeit $\mathcal{O}(|V|^{2,8})$.

Eine Beschränkung auf stark zusammenhängende Graphen ist wegen des Beweises zum Theorem 3.1. nicht möglich. Die Berechnung eines MEG ist also nur in kreisfreien Graphen in polynomieller Zeit möglich. Daher werden wir im folgenden unser allgemeines Ziel einschränken und uns auf die leichteren Probleme der transitiven Reduktionen und der Approximationen beschränken.

Dazu zeigen wir in Lemma 3.2., daß man sich bei der Entwicklung von Algorithmen für transitive Reduktionen und Approximationen auf stark zusammenhängende Graphen beschränken kann.

Lemma 3.2. *Seien $G = (V, E)$ ein Graph und T ein polynomieller Algorithmus, der zu einem stark zusammenhängenden Graphen eine transitive Reduktion berechnet. Dann ist eine transitive Reduktion von G in polynomieller Zeit berechenbar.*

Beweis: Sei G' der zugehörige Reduktionsgraph von G mit den starken Zusammenhangskomponenten $G_1, \ldots, G_k$. Sei G'_* eine transitive Reduktion von G', die nach Lemma 3.1. in polynomieller Zeit berechenbar ist. Außerdem sei G_* ein Graph, der für jeden Pfeil aus G'_* einen Repräsentanten aus E enthält. Dieser kann nach der Bemerkung 2.1. in polynomieller Zeit gefunden werden.
Definiere nun $F := \bigcup_{1 \leq i \leq k} E(T(G_i)) \cup E(G_*)$ und den Graphen $H := (V, F)$. Dann ist H in polynomieller Zeit berechenbar.
Wir zeigen nun, daß H eine transitive Reduktion von G ist.
Seien dazu $x, y \in V$ mit $x \xrightarrow[G]{*} y$. Wir zeigen $x \xrightarrow[H]{*} y$.

(i) x und y liegen in einer Zusammenhangskomponente G_i für ein $i \in \{1, \ldots, k\}$. Da T einen stark zusammenhängenden Graphen berechnet, ist in $T(G_i)$ ein Weg von x nach y enthalten.

(ii) x liegt in G_i, y in G_j mit $i, j \in \{1, \ldots, k\}$. Sei P der Weg in G_* von G_i nach G_j und $Q = (q_0, \ldots, q_l)$ der entsprechende Weg in G'_*. Dabei sind $q_0 \in G_i$ und $q_l \in G_j$. Dieser Weg existiert, da x und y in G verbunden sind. Also ist auch er in der transitiven Reduktion enthalten. Da $T(G_i)$ und $T(G_j)$ stark zusammenhängend sind, gibt es Wege $P_1 = (x, \ldots, q_0)$ und $P_2 = (q_l, \ldots, y)$. Daraus folgt, daß $(x, \ldots, q_0, \ldots, q_l, \ldots, y)$ ein Weg von x nach y in G ist.

Außerdem ist H ein Untergraph von G, und die Inklusionsminimalität folgt aus der von ${G'}_*$ sowie aus der entsprechenden Eigenschaft der transitiven Reduktionen $T(G_i)$. □

Lemma 3.3. liefert ein Verfahren, um eine transitive Reduktion in einem Graphen zu berechnen. Dieses ist in Tabelle 3.1 dargestellt.

Bestimme den Reduktionsgraphen G' von G
Bestimme für G' eine transitive Reduktion gemäß Lemma 3.1.
Bestimme transitive Reduktionen für jede starke Zusammenhangskomponente

Tabelle 3.1: Algorithmus zur Bestimmung einer transitiven Reduktion in allgemeinen Graphen

Bemerkung 3.7. *Lemma 3.2. kann auch mit einem Algorithmus T bewiesen werden, der in polynomieller Zeit einen α-approximativen Graphen berechnet. Dann ist das Ergebnis H α-approximativ.*

Der oben beschriebene Ansatz legt nahe, sich auf stark zusammenhängende Graphen zu beschränken. Deshalb werden wir nun einige Eigenschaften speziell für stark zusammenhängende Graphen festhalten:

Lemma 3.3. *Sei $G = (V, E)$ ein stark zusammenhängender Graph. Dann gilt:*

(i) $\mathcal{OPT}(G) \geq |V|$.

(ii) *Sei $(v_0, \ldots, v_n)$ ein Hamiltonkreis in G. Dann ist $M := (V, \{(v_{i-1}, v_i) \mid 1 \leq i \leq n\})$ ein* MEG *von G.*

(iii) *Jede transitive Reduktion von G beinhaltet höchstens $2|V| - 2$ Pfeile.*

(iv) *In jeder transitiven Reduktion und jedem* MEG *von G ist keine Schlinge enthalten.*

Beweis:

(i) Da jeder Graph mit weniger als $|V|$ Pfeilen nicht zusammenhängend ist, folgt die Behauptung.

(ii) Da $|E(M)| = |V|$ gilt, folgt die Aussage mit (i).

(iii) Seien $root \in V$ ein ausgezeichneter Knoten, S ein Baum und T ein Wurzelbaum jeweils zur Wurzel $root$ und $H := S \cup T$. Dann ist $H = S \cup T \subseteq G$. Zum Beweis, daß H stark zusammenhängend ist, seien $x, y \in V$. Dann ist $x \xrightarrow[T]{*} root \xrightarrow[S]{*} y$ ein Weg von x nach y.

Für einen relationalen Beweis seien $root$ der entsprechende Punkt der Menge V und U der zu T entsprechende Baum im transponierten Graphen. Beachte außerdem, daß wir mit S und U sowohl die Graphen als auch die Relationen bezeichnen. Dann gilt:

$$\mathsf{L} = \mathsf{L}\, root^T root\, \mathsf{L} = (root\, \mathsf{L})^T root\, \mathsf{L} \subseteq (U^*)^T S^* = (U^T)^* S^* \subseteq (S \cup U^T)^* = H^*$$

Außerdem gilt: $|E(H)| = |E(S \cup T)| \leq |E(S)| + |E(T)| = 2|V| - 2$.

Also besitzt jeder Graph mit mehr als $2|V| - 2$ Pfeilen einen echten Untergraphen, der stark zusammenhängend ist. Also ist er selbst nicht inklusionsminimal.

(iv) Die Behauptung folgt aus $G^* = (V, E \backslash \{(x,x)\})^*$ für alle $x \in V$ und der Minimalität einer transitiven Reduktion bzw. eines MEG. □

Das Ziel besteht nun darin, in den nächsten Kapiteln Algorithmen für stark zusammenhängende Graphen zu entwickeln, die transitive Reduktionen oder Approximationen eines MEG berechnen. Dabei sind Approximationen nur dann sinnvoll, wenn die Güte besser als 2 ist, da jede transitive Reduktion nach Lemma 3.3. diese Güte besitzt.

- Im vierten Kapitel wird ein einfaches Durchlaufverfahren zur Berechnung einer transitiven Reduktionen entwickelt, das keinem Autor zugeordnet werden kann. Wir nennen es in dieser Arbeit Minimierungsverfahren.
- Im nächsten Kapitel wird ebenfalls ein Algorithmus zur Berechnung einer transitiven Reduktion erarbeitet, dessen Ideen aus einer Arbeit von Simon [28] stammt.
- Im sechsten Kapitel entwickeln wir einen approximativen Algorithmus, dessen Grundgedanken aus einer Arbeit von Khuller et al. [20] stammen.

3.3 Geschichtliche Entwicklung

Zum Abschluß dieses Abschnittes wird eine zeitliche Übersicht der Arbeiten zu diesem Thema gegeben.
Untersuchungen zu diesem Themengebiet gibt es seit Ende der 60er Jahre. Moyles und Thomson [21] veröffentlichten 1969 eine leicht fehlerhafte Arbeit zur Berechnung eines MEG.
In der Arbeit von Aho, Garey und Ullmann [1] im Jahr 1972 wurden kreisfreie Graphen untersucht. Desweiteren betrachteten sie ein abgewandeltes Problem, in dem das Ergebnis nicht Untergraph des Originalgraphen sein mußte. In kreisfreien Graphen macht dies keinen Unterschied, in stark zusammenhängenden Graphen kann aber ein polynomieller Algorithmus für das Problem MEG angegeben werden.
1975 korrigierte Hsu [17] die Arbeit von Moyles und Thomson und untersuchte MEG's.
In der Arbeit von Noltemeier [22], ebenfalls aus dem Jahre 1975, wurden kreisfreie und allgemeine Graphen untersucht. Noltemeier bezeichnete seine Ergebnisse als transitiv irreduzible Kerne.
Im Jahr 1989 stellte Simon [28] einen linearen Algorithmus zur Berechnung von transitiven Reduktionen vor. Diese Arbeit wird im Kapitel 5 vorgestellt, sie ist aber fehlerhaft.
Gibbons et al. [11] stellten 1991 einen parallelen Algorithmus vor, die Arbeit enthält als Ergänzung auch sequentielle Ansätze.
Ein approximativer Ansatz wurde im Jahre 1995 von Khuller et al. [20] vorgestellt. Diese Arbeit ist Grundlage des Kapitels 6.

Kapitel 4

Minimierungsalgorithmus

Dieser Abschnitt beinhaltet einen einfachen Ansatz zur Berechnung einer transitiven Reduktion für einen stark zusammenhängenden Graphen.
Der vorgestellte Algorithmus ist stark an ein Durchlaufverfahren angelehnt und wird in diesem Kapitel formal entwickelt. Deswegen wird zuerst ein generisches Programm zur Berechnung allgemeiner inklusionsminimaler Probleme vorgestellt [5], mit dessen Hilfe dann im zweiten Teil eine transitive Reduktion berechnet werden kann.
Der Vorteil dieses Ansatzes liegt in seiner Kürze und Einfachheit. Er ist formal entwickelbar und für praktische Zwecke schnell und korrekt zu implementieren.
Als Voraussetzung werden noch einige Grundlagen für diesen Abschnitt geklärt, abschließend die theoretischen Eigenschaften zusammengefaßt und der Algorithmus an einigen Beispielen verdeutlicht.

4.1 Grundlagen

Als Grundlagen benötigen wir den Hoare-Kalkül [12] und vor allem die Relationentheorie, die in Abschnitt 2.2 vorgestellt wurde.
Um in diesem Kapitel ein **while**-Programm `p` der Form

$$\mathit{init};\ \textbf{while}\ b\ \textbf{do}\ \mathit{body}\ \textbf{end};$$

entwickeln zu können, wollen wir aus dem Themenbereich der Programmentwicklung die wichtigsten Dinge klarstellen:
Eine Problemspezifikation besteht aus einer Vorbedingung *pre* (Beschreibung der Eingabe) und der Nachbedingung *post* (Beschreibung der Ausgabe).
Der formale Beweis bzw. die formale Entwicklung beruht meist auf dem Finden einer Schleifeninvariante *inv*. Für das oben stehende Programm `p` ergeben sich dann folgende drei Beweisverpflichtungen:

(a) Die Formel $\mathit{inv} \wedge \mathit{NOT}(b) \rightarrow \mathit{post}$ ist gültig.

(b) Die Hoare-Formel $\{pre\}\ init\ \{inv\}$ ist herleitbar.

(c) Die Hoare-Formel $\{inv\ \wedge\ b\}\ body\ \{inv\}$ ist herleitbar.

Die Zusicherungen, wie z.B. die Invariante, werden dabei in den Programmtext als $\{inv\}$ geschrieben.
Durch die Beweisverpflichtungen ist `p` partiell korrekt. Um die totale Korrektheit zu zeigen, müssen wir zusichern, daß `p` terminiert.

Grundlage der Beweise sind zudem einige mengentheoretische Grundlagen, die wir hier nicht vorstellen wollen. Zur Übersichtlichkeit werden wir für eine Menge X und ein Element $x \in X$ anstelle von $X\ \cup\ \{x\}$ $X\ \cup\ x$ bzw. $X - x$ für $X \backslash \{x\}$ schreiben.

4.2 Entwicklung des Algorithmus

Wir wollen nun einen Algorithmus entwickeln, der für einen stark zusammenhängenden Graphen eine transitive Reduktion berechnet.
Dazu wird im ersten Teil ein generisches Programm für inklusionsminimale Probleme entwickelt und anschließend verfeinert, das wir im zweiten Abschnitt verwenden, um eine transitive Reduktion zu berechnen.

4.2.1 Generisches Programm für inklusionsminimale Probleme

Zur Entwicklung des generischen Programms nehmen wir uns eine endliche Menge M, ein Prädikat $\mathcal{P}$ auf der Potenzmenge $Pot(M)$ und ein Element $R \in Pot(M)$ her. Dabei sei das Prädikat $\mathcal{P}$ nach oben vererbend, d.h.

$$\forall\ X, Y \in Pot(M)\ :\ X\ \subseteq\ Y\ \wedge\ \mathcal{P}(X) \Longrightarrow \mathcal{P}(Y).$$

Diese Annahme macht Sinn, da wir sonst nicht von minimal bezüglich der Inklusion sprechen können. Es folgt, daß das negierte Prädikat $\neg\mathcal{P}$ nach unten vererbend ist, d.h. es gilt:

$$\forall\ X, Y \in Pot(M)\ :\ Y\ \subseteq\ X\ \wedge\ \neg\mathcal{P}(X) \Longrightarrow \neg\mathcal{P}(Y).$$

Wir nehmen an, daß wir für die Eingabe R eine inklusionsminimale Teilmenge berechnen wollen, die das Prädikat $\mathcal{P}$ erfüllt. Dabei wollen wir das Ergebnis in der Variablen A speichern und formulieren diese Aussage als Nachbedingung des generischen Programms:

$$post(R, A)\ \stackrel{\wedge}{=}\ \mathcal{P}(A)\ \wedge\ A\ \subseteq\ R\ \wedge\ \forall\ X \in Pot(A) : \mathcal{P}(X) \rightarrow X\ =\ A.$$

Zum Bestimmen der Invariante wird nun die Nachbedingung abgeschwächt. Dies führt mit einer Variablen B auch zur Abbruchbedingung der **while**-Schleife und entspricht der Beweisverpflichtung (a).

$$
\begin{aligned}
post(R, A) &\overset{(i)}{\Longleftrightarrow} \mathcal{P}(A) \wedge A \subseteq R \wedge \forall X \in Pot(A) : \mathcal{P}(X) \rightarrow X = A \\
&\overset{(ii)}{\Longleftrightarrow} \mathcal{P}(A) \wedge A \subseteq R \wedge \forall X \in Pot(A) : X \neq A \rightarrow \neg\mathcal{P}(X) \\
&\overset{(iii)}{\Longleftrightarrow} \mathcal{P}(A) \wedge A \subseteq R \wedge \forall x \in A : \neg\mathcal{P}(A - x) \\
&\overset{(iv)}{\Longleftarrow} \mathcal{P}(A) \wedge A \subseteq R \wedge B = \emptyset \wedge B \subseteq A \wedge \forall x \in A \backslash B : \neg\mathcal{P}(A - x)
\end{aligned}
$$

Der Schritt (i) ist die Definition von *post*, (ii) verwendet die Kontraposition und (iv) eine Verstärkung mit der Variablen B. Zu untersuchen ist noch der Schritt (iii). Die Richtung „$\Longrightarrow$“ folgt, da für alle $x \in A$ auch $A - x \in Pot(M)$ ist. Die umgekehrte Richtung „$\Longleftarrow$“ begründen wir wie folgt: Sei dazu $X \subset A$. Dann existiert ein Element $x \in A$ mit $X \subseteq A - x$. Aus $\neg\mathcal{P}(A - x)$ und der nach unten vererbenden Eigenschaften von $\neg\mathcal{P}$ folgt, daß $\neg\mathcal{P}(X)$ gilt.

Aus der Ableitung müssen wir nun die Abbruchbedingung und die Invariante extrahieren. Dabei wird $B \neq \emptyset$ zur Abbruchbedingung, der Rest zur Invariante der **while**-Schleife, die dann wie folgt aussieht:

$$
inv(R, A, B) \triangleq \mathcal{P}(A) \wedge A \subseteq R \wedge B \subseteq A \wedge \forall x \in A \backslash B : \neg\mathcal{P}(A - x)
$$

Die so erarbeiteten Zusicherungen führen zum in Tabelle 4.1 dargestellten Schema `Min`.

```
function Min(R)
  ...
  {inv(R, A, B)}
  while B ≠ ∅ do
    ...
  end;
  {post(R, A)}
  return A
```

Tabelle 4.1: Pseudocode Schema `Min`

Nach Beweisverpflichtung (b) suchen wir nun eine passende Initialisierung, um die Invariante zu etablieren. Da wir $\mathcal{P}$ nach oben vererbend angenommen haben, ist es sinnvoll, als Vorbedingung $pre(R) \triangleq \mathcal{P}(R)$ zu wählen. Falls R selbst das Prädikat nicht erfüllt, existiert auch keine inklusionsminimale Teilmenge. Es gilt:

$$
\begin{aligned}
pre(R) &\overset{(i)}{\Longleftrightarrow} \mathcal{P}(R) \\
&\overset{(ii)}{\Longleftrightarrow} \mathcal{P}(R) \wedge R \subseteq R \wedge R \subseteq R \wedge \forall x \in \emptyset : \neg\mathcal{P}(R - x) \\
&\overset{(iii)}{\Longleftrightarrow} inv(R, R, R)
\end{aligned}
$$

Die Schritte (i) und (iii) entsprechen dabei den Definitionen von pre und inv, (ii) ist eine Auffüllung der Formel mit trivial wahren Werten.
Diese Ableitung legt nahe, die Variablen A und B jeweils mit R zu initialisieren.

Wir müssen jetzt lediglich noch den Schleifenrumpf entwickeln. Nach Beweisverpflichtung (c) muß dieser die Invariante aufrechterhalten. Wir können die Abbruchbedingung $B \neq \emptyset$ sowie $inv(R, A, B)$ annehmen. Außerdem sei $b \in B$ gegeben. Wir entwickeln nun in 2 Fällen den Schleifenrumpf.
Im ersten Fall nehmen wir an, daß $\mathcal{P}(A - b)$ gilt:

$$
\begin{aligned}
& inv(R, A, B) \ \wedge \ \mathcal{P}(A - b) \\
& \overset{(i)}{\Longleftrightarrow} \quad \mathcal{P}(A) \ \wedge \ \mathcal{P}(A - b) \ \wedge \ A \ \subseteq \ R \ \wedge \ B \ \subseteq \ A \ \wedge \ \forall \ x \in A \backslash B : \neg \mathcal{P}(A - x) \\
& \overset{(ii)}{\Longrightarrow} \quad \mathcal{P}(A) \ \wedge \ \mathcal{P}(A - b) \ \wedge \ A \ \subseteq \ R \ \wedge \ B - b \ \subseteq \ A - b \ \wedge \ \forall \ x \in A \backslash B : \neg \mathcal{P}(A - x) \\
& \overset{(iii)}{\Longrightarrow} \quad \mathcal{P}(A - b) \ \wedge \ A - b \ \subseteq \ R \ \wedge \ B - b \ \subseteq \ A - b \ \wedge \ \forall \ x \in A \backslash B : \neg \mathcal{P}((A - b) - x) \\
& \overset{(iv)}{\Longrightarrow} \quad \mathcal{P}(A - b) \ \wedge \ A - b \ \subseteq \ R \ \wedge \ B - b \ \subseteq \ A - b \ \wedge \ \forall \ x \in (A \backslash B) - b : \neg \mathcal{P}((A - b) - x) \\
& \overset{(v)}{\Longleftrightarrow} \quad \mathcal{P}(A - b) \ \wedge \ A - b \ \subseteq \ R \ \wedge \ B - b \ \subseteq \ A - b \\
& \qquad \wedge \forall \ x \in (A - b) \backslash (B - b) : \neg \mathcal{P}((A - b) - x) \\
& \overset{(vi)}{\Longleftrightarrow} \quad inv(R, A - b, B - b)
\end{aligned}
$$

Schritt (i) und (vi) entsprechen dabei den Definitionen von inv, die Schritte (ii), (iv) und (v) sind einfache Mengentheorie und Logik. Der Schritt (iii) folgt aus $\mathcal{P}(A - b)$ und der nach unten vererbenden Eigenschaft von $\neg\mathcal{P}$. Also gilt auch $\neg\mathcal{P}((A - b) - x)$ für alle $x \in A \backslash B$.

Im zweiten Fall gilt $\mathcal{P}(A - b)$ nicht:

$$
\begin{aligned}
& inv(R, A, B) \ \wedge \ \neg\mathcal{P}(A - b) \\
& \overset{(i)}{\Longleftrightarrow} \quad \mathcal{P}(A) \ \wedge \ \neg\mathcal{P}(A - b) \ \wedge \ A \ \subseteq \ R \ \wedge \ B \ \subseteq \ A \ \wedge \ \forall \ x \in A \backslash B : \neg \mathcal{P}(A - x) \\
& \overset{(ii)}{\Longrightarrow} \quad \mathcal{P}(A) \ \wedge \ \neg\mathcal{P}(A - b) \ \wedge \ A \ \subseteq \ R \ \wedge \ B - b \ \subseteq \ A \ \wedge \ \forall \ x \in A \backslash B : \neg \mathcal{P}(A - x) \\
& \overset{(iii)}{\Longleftrightarrow} \quad \mathcal{P}(A) \ \wedge \ A \ \subseteq \ R \ \wedge \ B - b \ \subseteq \ A \ \wedge \ \forall \ x \in (A \backslash B) \ \cup \ b : \neg \mathcal{P}(A - x) \\
& \overset{(iv)}{\Longleftrightarrow} \quad \mathcal{P}(A) \ \wedge \ A \ \subseteq \ R \ \wedge \ B - b \ \subseteq \ A \ \wedge \ \forall \ x \in A \backslash (B - b) : \neg \mathcal{P}(A - x) \\
& \overset{(v)}{\Longleftrightarrow} \quad inv(R, A, B - b)
\end{aligned}
$$

Auch hier sind die Schritte (i) und (v) die Definition von inv. (ii) und (iv) sind einfache Mengentheorie und Logik. Schritt (iii) benutzt wieder die nach unten vererbende Eigenschaft von $\neg\mathcal{P}$.

Wir stellen fest, daß B jeweils durch $B - b$ ersetzt, und A nur im ersten Fall $A - b$ zugewiesen werden muß.

```
function Min(R)
  {pre(R)}
  A, B := R, R;
  {inv(R, A, B)}
  while B ≠ ∅ do
    b := elem(B);
    if P(A − b)
      then A, B := A − b, B − b;
      else  B := B − b;
    end;
  end;
  {post(R, A)}
  return A
```

Tabelle 4.2: Pseudocode `Min`

Aus der erarbeiteten Invariante, der Abbruchbedingung, der Initialisierung und dem Schleifenrumpf ergibt sich der in Tabelle 4.2 dargestellte Pseudocode für das generische Programm `Min`. Dabei nehmen wir an, daß `elem`(X) ein Element der (nichtleeren) Menge X liefert.
Mit der Herleitung unter Berücksichtigung der drei Beweisverpflichtungen ist die partielle Korrektheit von `Min` gezeigt. Da in jedem Durchlauf der **while**-Schleife die endliche Menge B echt verkleinert wird, folgt die Terminierung und damit die totale Korrektheit.

Verfeinerungen

Die Laufzeit des generischen Programms hängt vor allem von der Anzahl der Durchläufe der **while**-Schleife sowie von den Kosten zur Auswertung des Prädikates $\mathcal{P}$ ab. Wir wollen nun durch zwei Verfeinerungen die Gesamtkosten senken.

Um den Aufwand für das Auswerten des Prädikates $\mathcal{P}$ zu verringern, ist es manchmal möglich, ein weiteres Prädikat $\mathcal{Q}$ auf $Pot(M) \times M$ einzuführen, das folgender Eigenschaft genügt:

$$\mathcal{P}(X - b) \iff \mathcal{P}(X) \wedge \mathcal{Q}(X, b) \quad f.a.\ X \in Pot(M), b \in X$$

Da $\mathcal{P}(X)$ als Schleifeninvariante gilt und durch $b :=$ `elem`(B) diese Eigenschaft nicht verändert wird, braucht nur $\mathcal{Q}(X, b)$ ausgewertet werden.
Dieses ist vor allem dann sinnvoll, wenn die Auswertung von $Q(X, b)$ schneller erfolgen kann als die von $\mathcal{P}(X - b)$.

Eine weitere Verbesserungsmöglichkeit ist es, die Initialisierung zu einer Vorberechnung zu erweitern. Dabei wird eine Menge S gesucht, die $\mathcal{P}(S)$ und $S \subseteq R$ erfüllt, durch einen anderen Ansatz effizient berechenbar und deren Kardinalität viel kleiner als die von R ist. Diese muß bei der jeweiligen speziellen Instantiierung entsprechend gewählt werden.

Mit der vorgeschalteten Vorberechnung sowie einem Prädikat $\mathcal{Q}$ sieht das verfeinerte generische Programm `Min'` wie in Tabelle 4.3 aus.

```
function Min'(R)
  {pre(R)}
  ≫ Vorberechnung von S ≪
  {P(S) ∧ S ⊆ R}
  A, B := S, S;
  {inv(R, A, B)}
  while B ≠ ∅ do
    b := elem(B);
    if Q(A, b)
      then A, B := A − b, B − b;
      else  B := B − b;
    end;
  end;
  {post(R, A)}
  return A
```

Tabelle 4.3: Pseudocode `Min'`

4.2.2 Transitive Reduktion als Spezialfall des generischen Programms

Gegeben sei nun ein Graph $G = (V, R)$. Nach der Problemstellung können wir G als stark zusammenhängend annehmen. Dabei wählen wir die Menge M und das Prädikat $\mathcal{P}$, so daß die Nachbedingung $post(R, A)$ der Forderung einer transitiven Reduktion entspricht. Dazu sei $M := V \times V$ und $\mathcal{P}(X) :\Longleftrightarrow X^* = \mathsf{L}$. Dann ist $\mathcal{P}$ nach oben vererbend. Wir ersetzen außerdem die allgemeine mengentheoretische Operation `elem` durch die relationale `atom`, die aus einer (nichtleeren) Pfeilmenge einen Pfeil liefert, und die mengentheoretischen Operationen durch die entsprechenden relationentheoretischen.
Dann lassen sich die Zusicherungen innerhalb von `Min` wie folgt schreiben:

$$\begin{array}{lll} \text{Vorbedingung } (pre) & \stackrel{\wedge}{=} & \{R^* = \mathsf{L}\} \\ \text{Invariante } (inv) & \stackrel{\wedge}{=} & \{A^* = \mathsf{L} \wedge A \subseteq R \wedge B \subseteq A \wedge \forall\, x \in A \backslash B : (A \cap \overline{x})^* \neq \mathsf{L}\} \\ \text{Nachbedingung } (post) & \stackrel{\wedge}{=} & \{A^* = \mathsf{L} \wedge A \subseteq R \wedge \forall\, x \in A : (A \cap \overline{x})^* \neq \mathsf{L}\} \end{array}$$

Die Vorbedingung entspricht der Tatsache, daß G stark zusammenhängend ist.
Die Invariante kann wie folgt gedeutet werden: Das Zwischenergebnis, der Graph (V, A), ist ein stark zusammenhängender Untergraph von G, in dem alle Pfeile in $A \backslash B$ nicht reduzierbar in A sind.

Dabei kann mit Hilfe der Abbruchbedingung $B \neq \mathsf{O}$ die Nachbedingung gefolgert werden, die der Tatsache entspricht, daß der Graph (V, A) eine transitive Reduktion ist.
Wir erhalten durch Einsetzen den in Tabelle 4.4 dargestellten Algorithmus `TransRed`.

function `TransRed`(R)
  $\{R^* = \mathsf{L}\}$
  $A, B := R, R;$
  $\{A^* = \mathsf{L} \wedge A \subseteq R \wedge B \subseteq A \wedge \forall\, x \in A \backslash B : (A \cap \overline{x})^* \neq \mathsf{L}\}$
  while $B \neq \mathsf{O}$ **do**
    $b :=$ `atom`$(B);$
    if $(A \cap \overline{b})^* = \mathsf{L}$
      then $A, B := A \cap \overline{b}, B \cap \overline{b};$
      else $B := B \cap \overline{b};$
    end;
  end;
  $\{A^* = \mathsf{L} \wedge A \subseteq R \wedge \forall\, x \in A : (A \cap \overline{x})^* \neq \mathsf{L}\}$
  return A

Tabelle 4.4: Pseudocode `TransRed`

Verfeinerungen

Entsprechend der Entwicklung des generischen Programms wollen wir die im vorherigen Abschnitt vorgestellten allgemeinen Verfeinerungen speziell für dieses Problem umsetzen.

Dazu entwickeln wir zuerst eine Vorberechnung, die graphentheoretisch gesprochen einen stark zusammenhängenden Untergraphen liefert. Als weitere Forderung soll die Anzahl der Pfeile möglichst klein und die gesamte Vorberechnung schnell durchführbar sein.
Wir lassen uns durch das Lemma 3.3. leiten und vereinigen einen Baum und einen Wurzelbaum mit gemeinsamer Wurzel des eingegebenen Graphen.
Daß dabei ein stark zusammenhängender Untergraph berechnet wird, folgt sofort aus dem Beweis zum Lemma.
Dabei nehmen wir nun einen Algorithmus `Tree` an, der zu einem Graphen Q und einem Punkt p einen Baum berechnet. Formal heißt dies, daß der Baumalgorithmus der Vorbedingung

$$Q^* = \mathsf{L} \wedge p = p\,\mathsf{L} \wedge pp^T \subseteq \mathsf{I} \wedge p^T = \mathsf{L}$$

und der Nachbedingung

$$T \subseteq Q \wedge p\,\mathsf{L} \subseteq T^* \wedge TT^T \subseteq \mathsf{I} \wedge T^+ \subseteq \overline{\mathsf{I}}$$

genügt.

Typische Beispiele für solche Algorithmen sind die Tiefensuche [31, 6] und die Breitensuche (engl.: breadth first search, BFS) [18, 23]. Das erste Verfahren wurde schon in Abschnitt 2.5 vorgestellt, das Ergebnis ist der DFS-Baum. Der zweite Ansatz ist dadurch charakterisiert, daß zuerst alle Nachfolger vor den anderen Knoten besucht werden. Auch hier kann leicht ein Baum zurückgegeben werden.
Für einen eingegebenen Graphen $G = (V, R)$ kann mit Hilfe dieser Vorberechnung die Anzahl der Durchläufe von $|R|$ auf mindestens $2|V| - 2$ gesenkt werden.

Um die zweite Verfeinerung umzusetzen, wollen wir nun ein Prädikat $\mathcal{Q}$ entwickeln.
Seien dazu $G = (V, E)$ ein stark zusammenhängenden Graph und $e = (x, y) \in E$ ein Pfeil. Das Prädikat $\mathcal{P}$ entsprach bei unserer Instantiierung dem starken Zusammenhang. Um das Prädikat $\mathcal{P}$ für den Graphen $G' := (V, E \backslash \{e\})$ zu prüfen, muß die Hülle von G' berechnet und anschließend überprüft werden, ob sie dem vollen Graphen entspricht. Diese Auswertungsmethode benötigt nach [30] $\mathcal{O}(|V|^{2,8})$ Zeit.
Da wir aber die Invariante, G ist stark zusammenhängend, voraussetzen können, genügt es zu überprüfen, ob es einen Weg von x nach y in G' gibt. Existiert kein solcher Weg, ist G' nicht stark zusammenhängend. Andernfalls nehmen wir uns zwei Knoten $u, v \in V$ und einen Weg $W := u \xrightarrow[G]{*} v$ in G her. Falls der Pfeil (x, y) in W vorhanden ist, ersetzen wir diesen durch den gefundenen Weg in G'. Wir erhalten so einen Weg in G' von u nach v, womit der starke Zusammenhang von G' gezeigt ist.
Geleitet von dieser Argumentation gilt nun:

$$\mathcal{P}(X \cap \overline{e}) \iff (X \cap \overline{e})^* = \mathsf{L} \iff X^* = \mathsf{L} \wedge (X \cap \overline{e})^*_{xy}.$$

Also können wir nun Q für alle $X \in Pot(M), e = (x, y) \in M$ wie folgt definieren:

$$\mathcal{Q}(X, e) : \iff (X \cap \overline{e})^*_{xy}.$$

Das Prädikat $\mathcal{Q}$ kann nun schneller ausgewertet werden als $\mathcal{P}$. $\mathcal{Q}$ ist genau dann wahr, wenn der Knoten y im Graphen G' von x aus erreichbar ist. Dies kann z.B. mit Hilfe eines Tiefensuchdurchlaufes vom Knoten x aus überprüft werden. Damit sind die Kosten zur Auswertung auf $\mathcal{O}(|V| + |E|)$ gesenkt.

Diese Verfeinerungen führen zu dem neuen Algorithmus `TransRed'`, dargestellt in Tabelle 4.5, wobei wir die relationentheoretische Operation `point` voraussetzen, die aus einem (nichtleeren) Vektor einen Punkt liefert.

4.2.3 Theoretische Eigenschaften

In diesem Abschnitt werden die theoretischen Eigenschaften des Algorithmus `TransRed'` dargestellt.
Zuerst betrachten wir die Güte der entstehenden transitiven Reduktion, sie folgt auch direkt aus dem Lemma 3.3..

function TransRed'(R)
  $\{R^* = \mathsf{L}\}$
  $p := \mathtt{point}(\mathsf{L})$;
  $S := \mathtt{Tree}(R, p) \cup \mathtt{Tree}(R^T, p)^T$;
  $\{S^* = \mathsf{L} \wedge S \subseteq R\}$
  $A, B := S, S$;
  $\{A^* = \mathsf{L} \wedge A \subseteq R \wedge B \subseteq A \wedge \forall\, x \in A \backslash B : (A \cap \overline{x})^* \neq \mathsf{L}\}$
  while $B \neq \mathsf{O}$ **do**
    $b := (x, y) := \mathtt{atom}(B)$;
    if $(A \cap \overline{b})^*_{xy}$
      then $A, B := A \cap \overline{b}, B \cap \overline{b}$;
      else $B := B \cap \overline{b}$;
    end;
  end;
  $\{A^* = \mathsf{L} \wedge A \subseteq R \wedge \forall\, x \in A : (A \cap \overline{x})^* \neq \mathsf{L}\}$
  return A

Tabelle 4.5: Pseudocode TransRed'

Lemma 4.1. *Seien $G = (V, E)$ ein stark zusammenhängender Graph und $A :=$ TransRed'(E). Der resultierende Graph (V, A) besitzt höchstens $2n - 2$ Pfeile, also $|A| < 2 \cdot \mathcal{OPT}(G)$.*

Beweis: Da jeder Baum von G genau $|V - 1|$ Pfeile besitzt und jeder stark zusammenhängende Graph mindestens $|V|$ Pfeile enthält, gilt:

$$2 \cdot |V| = 2 \cdot |\mathtt{Tree}(G)| - 2 \geq |A| \geq \mathcal{OPT}(G) \geq |V|.$$

Es folgt $|A| \leq 2 \cdot \mathcal{OPT}(G)$. □

Im folgenden wird die Laufzeit untersucht. Dabei nehmen wir an, daß die Berechnung eines Baumes in Zeit $\mathcal{O}(|V| + |E|)$ möglich ist. Dies kann, wie oben erwähnt, mit Hilfe eines DFS-Durchlaufes oder eines BFS-Durchlaufes geschehen. Außerdem kann das Prädikat $\mathcal{Q}$ z.B. mit Hilfe eines DFS-Durchlaufes getestet werden.
Berücksichtigen wir, daß der Graph nach der Vorberechnung höchstens $2|V| - 2$ Pfeile enthält, ergibt sich folgendes Lemma:

Lemma 4.2. *Sei $G = (V, E)$ ein stark zusammenhängender Graph. Bezeichne mit $n = |V|$ die Anzahl der Knoten. Das Programm* TransRed' *berechnet eine transitive Reduktion von G in Zeit $\mathcal{O}(n^2)$, falls der Algorithmus* Tree *zur Bestimmung eines Baumes höchstens die Zeit $\mathcal{O}(n^2)$ benötigt.*

Beweis: Die Korrektheit wurde schon in der Herleitung gezeigt. Nach dem Lemma zur Güte ist klar, daß die **while**-Schleife höchstens $2 \cdot n$-mal durchlaufen wird. Die Schleife selbst besteht

bis auf die Prüfung $(A \cap \overline{b})^*_{xy}$ aus elementaren Operationen. Die Prüfung des Prädikates $\mathcal{Q}$ kann in $\mathcal{O}(n + 2 \cdot n)$ Zeit erfolgen. Die Vorberechnung der Bäume ist nach Voraussetzung ebenfalls in $\mathcal{O}(n^2)$ möglich, die Operationen `point` und ^{T} sind ebenfalls in dieser Zeit möglich. Als Gesamtaufwand ergibt sich $2 \cdot \mathcal{O}(n^2) + 2 \cdot n \cdot \mathcal{O}(n + 2 \cdot n) = \mathcal{O}(n^2)$. □

4.3 Beispiele

In diesem Abschnitt wird das hergeleitete Verfahren an einigen Beispielen veranschaulicht. Dazu betrachten wir zwei Beispiele.

Beispieldurchlauf

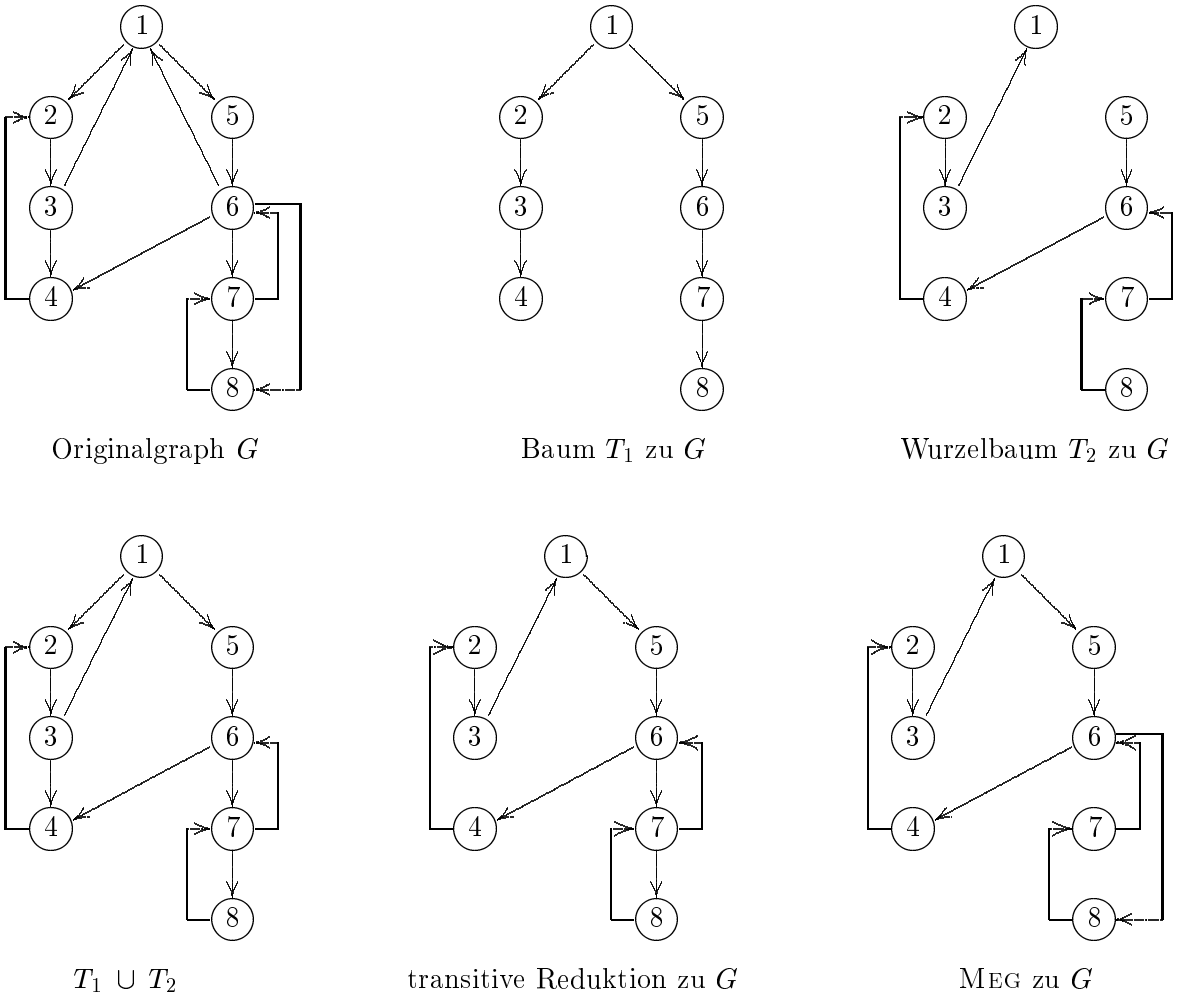


Abbildung 4.1: Beispieldurchlauf für das Minimierungsverfahren

In Abbildung 4.1 wird zuerst der Originalgraph G mit 8 Knoten gezeigt. Im ersten Schritt wird zur Wurzel 1 ein Baum T_1 sowie ein Wurzelbaum T_2 berechnet. Die beiden Bäume werden vereinigt und in diesem Untergraph jeder Pfeil nacheinander geprüft, ob der Graph ohne diesen Pfeil noch stark zusammenhängend ist. Daher können, abhängig von der Abarbeitungsreihenfolge, z.B. zuerst $(1,2)$ und dann $(3,4)$ gelöscht werden. Das Ergebnis ist eine transitive Reduktion. Das letzte Bild zeigt außerdem einen MEG von G.

Worst-case Beispiel

Das zweite Beispiel zeigt, daß die obere Schranke des Lemmas 4.2. auch erreichbar ist.
Dazu sei der Beispielgraph G auf n Knoten, wie in Abbildung 4.2 ersichtlich, gegeben. Es erfolgt wieder die Berechnung eines Baumes T_1 und eines Wurzelbaumes T_2 zur Wurzel 1. Der entstehende Graph ist schon eine transitive Reduktion, d.h. kein Pfeil ist mehr reduzierbar. Das letzte Bild zeigt außerdem einen MEG zu G, der aus einem Hamiltonkreis besteht. Somit gilt $\mathcal{OPT}(G) = n$. Einfaches Abzählen ergibt, daß die transitive Reduktion aus $2n-2$ Pfeilen besteht.

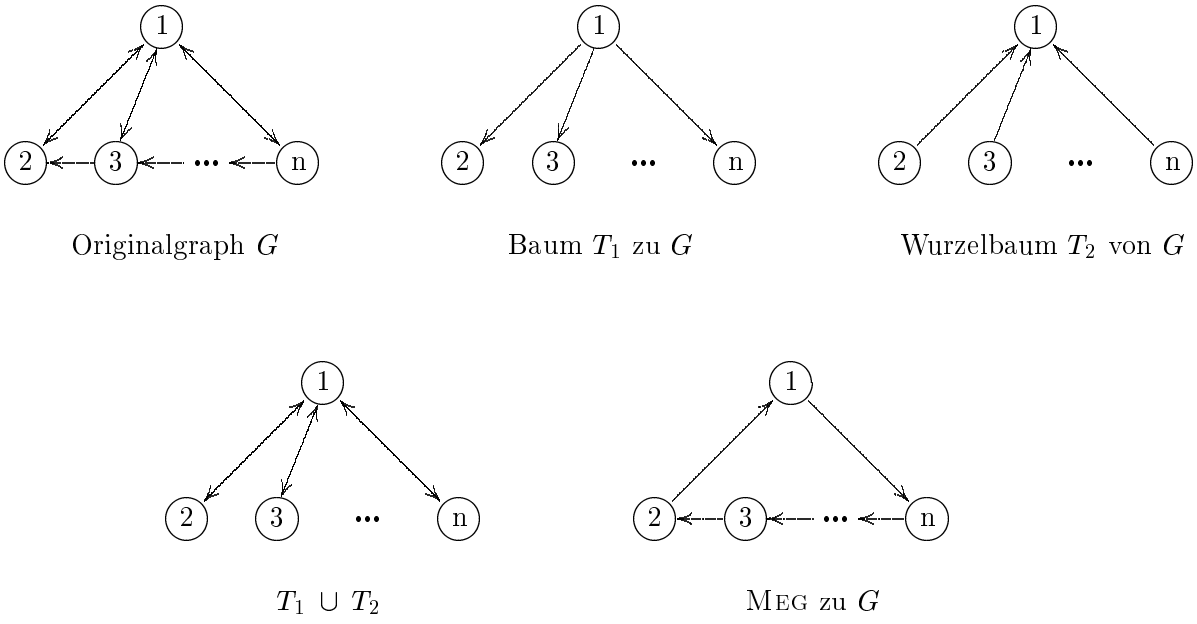


Abbildung 4.2: Worst-case Beispiel zum Minimierungsverfahren

Kapitel 5

Algorithmus von Simon

Grundlage dieses Kapitels ist eine Arbeit von Simon [28], die zum Ziel hatte, die quadratische Laufzeit anderer Verfahren zu senken.
Da diese Arbeit, wie in der Untersuchung festgestellt, aber fehlerhaft ist, wird hier eine abgewandelte Version vorgestellt. Auf den fehlerhaften Teil sowie die Gründe für das Scheitern dieses Ansatzes wird im separaten Abschnitt 5.3 eingegangen.
Die Idee des Ansatzes von Simon ist es, Pfeile aufgrund ihrer Eigenschaften aus einem Tiefensuchdurchlauf auf ihre Reduzierbarkeit zu überprüfen. Dabei werden in einem ersten Schritt die Rückwärts-, Vorwärts- und Querpfeile einer Prüfung unterzogen, der zweite Teil benutzt den ersten, um auch die Baumpfeile zu testen.
Dazu benötigen wir einige Lemmata, die im ersten Teil vorgestellt werden. Nach dem Algorithmus zur Berechnung einer transitiven Reduktion in einem stark zusammenhängenden Graphen folgen der Abschnitt über den fehlerhaften Teil sowie einige erläuternde Beispiele.

5.1 Grundlagen

Wir benötigen für die Entwicklung des Algorithmus zwei Lemmata.
Die Reduzierbarkeit von Pfeilen in stark zusammenhängenden Graphen entspricht dem Prädikat $\mathcal{P}$ aus Kapitel 4. Das nachfolgende Lemma beschreibt graphentheoretisch die nach oben vererbende Eigenschaft von $\mathcal{P}$ (i) bzw. die nach unten vererbende von $\neg\mathcal{P}$ (ii).

Lemma 5.1. *Seien $G = (V, E)$ und $H = (V, F)$ Graphen mit $E \subseteq F$ und $e = (x, y) \in E$ und $f \in F$. Dann gilt:*

(i) *e reduzierbar in $G \Longrightarrow e$ reduzierbar in H*

(ii) *f nicht reduzierbar in $H \Longrightarrow f$ nicht reduzierbar in G*

(iii) *(x, y) reduzierbar in $G \iff (y, x)$ reduzierbar in G^T.*

Beweis:

(i) Sei $e = (x, y)$ reduzierbar in G. Dann gibt es einen Weg $W = (x, \ldots, y)$ in $(V, E\backslash\{e\})$. W ist auch ein Weg in $(V, F\backslash\{e\})$, also ist e reduzierbar in H.

(ii) Folgt sofort aus (i) mit Hilfe der Kontraposition.

(iii) Sei (x, y) reduzierbar in G. Dann gibt es einen Weg $(x = x_0, \ldots, x_n = y)$ in $(V, E\backslash\{e\})$. $(x_n, \ldots, x_0)$ ist ein Weg in $(V, E^T\backslash\{(y, x)\})$, also ist (y, x) reduzierbar in G^T. □

Außerdem benötigen wir einige Eigenschaften der DFS-Partitionen in einem Graphen.

Lemma 5.2. *Seien $G = (V, E)$ ein stark zusammenhängender Graph mit DFS-Partitionen T, B, C, F und $e \in C \cup B \cup F$ ein Pfeil. Sei $G' := (V, E\backslash\{e\})$ ein Graph mit DFS-Partitionen $T = T'$, B', C', F'. Dann gilt:*

(i) $e \in B \Longrightarrow B' = B\backslash\{e\} \ \wedge \ C' = C \ \wedge \ F' = F$

(ii) $e \in F \Longrightarrow F' = F\backslash\{e\} \ \wedge \ C' = C \ \wedge \ B' = B$

(iii) $e \in C \Longrightarrow C' = C\backslash\{e\} \ \wedge \ B' = B \ \wedge \ F' = F.$

D.h. die DFS-Partitionen bleiben bis auf den gelöschten Pfeil erhalten.
Seien $f \notin E$ ein Pfeil und $G'' := (V, E \cup \{f\})$ ein Graph mit DFS-Partitionen $T = T''$, B'', C'', F''. Dann gilt:

$$f \in B' \Rightarrow B'' = B \cup \{f\} \ \wedge \ C'' = C \ \wedge \ F'' = F.$$

Beweis: Der Beweis folgt mit Lemma 2.1. und der Tatsache, daß die Reihenfolge in der die Knoten besucht werden, durch das Löschen bzw. Einfügen nicht verändert wird. □

Bemerkung 5.1. *Die Bedingungen $T = T'$ und $T = T''$ können auch wegfallen bzw. lassen sich folgern, wenn wir annehmen, daß der DFS-Durchlauf von derselben Wurzel aus und mit derselben zugrundeliegenden Ordnung ausgeführt wird.*

In diesem Abschnitt wird jeder Knoten v mit der DFS-Zahl $d[v]$ identifiziert. Dies bedeutet, daß die zugrundeliegende Ordnung auf den Knoten i.a. die Ordnung $<_T$ auf dem d-Feld ist.

5.2 Entwicklung des Algorithmus

Der Algorithmus ist grundsätzlich in zwei große Abschnitte aufgeteilt. Im ersten Teil werden mit Hilfe von Eigenschaften des DFS-Durchlaufes alle reduzierbaren Quer-, Vorwärts- und Rückwärtspfeile gelöscht. Der zweite Teil verwendet den ersten, um auch die Baumpfeile zu überprüfen.

5.2.1 Reduktion von Vorwärts-, Rückwärts- und Querpfeilen

Sei $G = (V, E)$ ein stark zusammenhängender Graph mit DFS-Partitionen T, B, C, F mit Wurzel *root*.
Wir wollen nun einen Algorithmus `A` entwickeln, der zu G einen Untergraphen berechnet, der neben allen Baumpfeilen nur nichtreduzierbare Pfeile enthält.
Wir unterscheiden im folgenden zwischen drei verschiedenen Zuständen eines Pfeiles. Entweder ist ein Pfeil reduzierbar, nicht reduzierbar oder es ist noch keine Aussage über seine Reduzierbarkeit möglich. Dazu markieren wir zuerst alle Baumpfeile $e \in T$ als nicht reduzierbar, die sonstigen Pfeile werden dem letzten Zustand zugeordnet. Dabei ist das Prädikat reduzierbar damit gleichsetzbar, daß der Pfeil gelöscht werden kann.
Wir werden jetzt versuchen, Pfeile gemäß ihrer Eigenschaft aus einem Tiefensuchdurchlauf durch verschiedene Lemmata den Zuständen reduzierbar und nicht reduzierbar zuzuordnen. Nach jedem Lemma werden die Ergebnisse in einen Pseudocode umgesetzt, um die Übersichtlichkeit zu erhalten. Am Ende dieses Abschnittes werden wir diese Teile dann zu dem Gesamtalgorithmus zusammensetzen.

Vorwärtspfeile

Begonnen wird mit den Vorwärtspfeilen. Das erste Lemma zeigt, daß in einem stark zusammenhängenden Graphen alle Vorwärtspfeile reduzierbar sind.

Lemma 5.3. *Seien $G = (V, E)$ ein stark zusammenhängender Graph mit DFS-Partitionen T, B, C, F und $e = (v, w) \in F$ ein Vorwärtspfeil. Dann gibt es einen reduzierenden Weg $W = v \xrightarrow[T]{+} w$ für (v, w) in G.*

Beweis: Die Definition eines Vorwärtspfeiles liefert einen Weg $v \xrightarrow[T]{+} w$. Dieser ist ein reduzierender Weg für (v, w). □

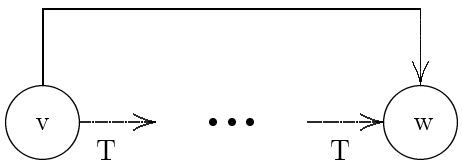


Abbildung 5.1: Lemma 5.3

Die Reduzierbarkeit zweier Vorwärtspfeile hängt nicht voneinander ab, da der reduzierende Weg nur aus Baumpfeilen besteht. Es ist also möglich, alle Vorwärtspfeile als reduzierbar zu markieren, was der Pseudocode in Tabelle 5.1 zeigt.

```
forall e ∈ F do
    delete e from G;
end;
```

Tabelle 5.1: Pseudocode Lemma 5.3.

Rückwärtspfeile 1

Als nächstes untersuchen wir Rückwärtspfeile auf ihre Reduzierbarkeit. So sind nach dem folgenden Lemma einige Rückwärtspfeile reduzierbar.

Lemma 5.4. *Seien $G = (V, E)$ ein stark zusammenhängender Graph mit DFS-Partitionen T, B, C, F und $\{(v, w_1), \ldots, (v, w_s)\}$ eine Menge von Rückwärtspfeilen vom Knoten v aus, so daß $w_1 <_T w_2 <_T \ldots <_T w_s$ gilt. Dann sind die Pfeile $(v, w_2), \ldots, (v, w_s)$ reduzierbar in G.*

Beweis: Die Knoten $w_1, \ldots, w_s$ sind alle im Weg $root \xrightarrow[T]{+} v$ enthalten, da die Pfeile (v, w_i) sonst Querpfeile wären. Aus diesem Grund gibt es reduzierende Wege $W_i := v \to w_1 \xrightarrow[T]{+} w_i$ für jeden Rückwärtspfeil (v, w_i), $i \in \{2, \ldots, s\}$. □

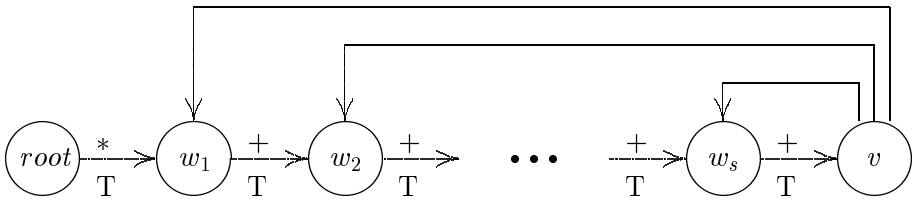


Abbildung 5.2: Lemma 5.4

Nach dem Lemma 5.4. können wir nun die genannten Rückwärtspfeile als reduzierbar markieren. Über den Rückwärtspfeil (v, w_1) wird noch keine Aussage gemacht, dieser wird erst mit Hilfe des Lemmas 5.6. auf seine Reduzierbarkeit getestet.
Um für jeden Knoten v jeweils den kleinsten Knoten $w_1(v)$ zu berechnen, brauchen wir ein Feld *backpoint*. Dieses wird mit $backpoint(v) := v$ initialisiert und für jeden Rückwärtspfeil (v, w) durch $backpoint(v) := \min(backpoint(v), w)$ aktualisiert. Nach so einem Durchlauf bezeichnet dann $backpoint(v)$ den Knoten $w_1(v)$. Anschließend können dann die Pfeile $\{(v, w_2), \ldots, (v, w_s)\}$ für jeden Knoten $v \in V$ gelöscht werden, da ihre Reduzierbarkeit nicht voneinander abhängt; ihre reduzierbaren Wege bestehen nur aus Baumpfeilen und aus dem in diesem Schritt nicht reduzierten Pfeil $(v, v(w_1))$.
Der in Tabelle 5.2 vorgestellte Pseudocode entspricht dem erarbeiteten Verfahren.

Führen wir nun die ersten beiden erarbeiteten Schritte aus, so erhalten wir einen Graphen, der keinen Vorwärtspfeil und für jeden Knoten nur höchstens einen Rückwärtspfeil enthält.

```
forall v ∈ V do
  backpoint(v) := v;
end;
forall e = (v, w) ∈ B do
  backpoint(v) := min (backpoint(v), w);
end;
forall e = (v, w) ∈ B do
  if w ≠ backpoint(v)
    then delete (v, w) from G;
  end;
end;
```

Tabelle 5.2: Pseudocode Lemma 5.4.

Querpfeile

Der nächste Schritt ist die Überprüfung der Reduzierbarkeit aller Querpfeile. Dabei stellen wir zunächst eine Bedingung dafür auf, wann ein Querpfeil auf jeden Fall reduzierbar ist. Falls ein Querpfeil diese Bedingung nicht erfüllt, wird dieser Querpfeil durch einen Rückwärtspfeil ersetzt. Dabei muß gewährleistet sein, daß die Reduzierbarkeit beider Pfeile äquivalent ist, d.h. daß der Querpfeil genau dann reduzierbar ist, wenn der Rückwärtspfeil reduzierbar ist. Außerdem muß sichergestellt sein, daß die Ersetzung keine Auswirkungen auf die Reduzierbarkeit anderer Pfeile hat.
Den neu eingefügten Rückwärtspfeil prüfen wir später mit Hilfe des Lemmas 5.6 auf seine Reduzierbarkeit und ersetzen ihn zum Schluß des Algorithmus, falls er nicht reduziert wurde, durch den ursprünglichen Querpfeil.

Lemma 5.5. *Seien $G = (V, E)$ ein stark zusammenhängender Graph mit DFS-Partitionen T, B, C, F, $e = (v, z) \in C$ ein Querpfeil und w der nächste gemeinsame Vorfahr von v und z im DFS-Baum T. Dann gilt:*

(i) *Wenn es einen Knoten $t \in V$ mit $t \xrightarrow[T]{*} w$ und $(v, t) \in E$ gibt, so ist (v, z) reduzierbar in G.*

(ii) *Falls es so einen Knoten t nicht gibt, gilt:*

  (a) *Der Pfeil (v, z) ist reduzierbar in G genau dann, wenn der Pfeil $e' = (v, w)$ reduzierbar in $G_1 := (V, E \backslash \{(v, z)\} \cup \{(v, w)\})$ ist.*

  (b) $G_1^* = G^*$

  (c) *Sei $e'' \in C \cup B \cup F$ ein Pfeil mit $e \neq e'' \neq e'$. Dann ist e'' reduzierbar in G genau dann, wenn e'' reduzierbar in G_1 ist.*

Beweis:

(i) Sei t so ein Knoten. Dann ist der reduzierende Weg für (v, z) durch $v \to t \xrightarrow[T]{*} w \xrightarrow[T]{+} z$ gegeben ist.

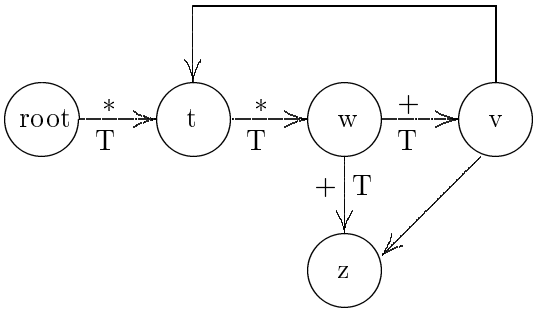


Abbildung 5.3: Lemma 5.5 (i): Graph G

(ii) Da es keinen Knoten t mit den obigen Eigenschaften gibt, gilt $(v, w) \notin E$.

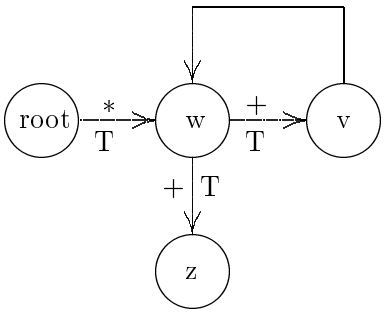


Abbildung 5.4: Lemma 5.5 (ii): Graph G'

(a) Setze $G_2 := (V, E \backslash \{(v, z)\})$.

„$\Longrightarrow$“: Sei (v, z) reduzierbar in G. Dann gibt es einen reduzierenden Weg W_1 von v nach z. Da G stark zusammenhängend ist, gibt es einen einfachen Weg W_2 von z nach w. Da W_2 einfach ist, enthält er den Pfeil (v, z) nicht. Die Komposition der Wege W_1 und W_2 ergibt einen neuen Weg W_3; dieser ist ein Weg in G_2 und damit ein reduzierender Weg in G_1 für (v, w).

„$\Longleftarrow$“: Sei nun W_1 ein reduzierender Weg für (v, w) in G_1. Wird dieser erweitert durch den Weg $w \xrightarrow[T]{+} z$ zu einem Weg von v nach z, so ergibt sich ein reduzierender Weg für (v, z) in G.

(b) Da G stark zusammenhängend ist und damit $G^* = \mathsf{L}$ gilt, bleibt nur ${G_1}^* = \mathsf{L}$ zu zeigen. Seien dazu $x, y \in V$. Wir wollen nun beweisen, daß es in G_1 einen Weg von

x nach y gibt. Sei W ein Weg von x nach y in G. Falls $(v,z) \notin W$, ist W auch ein Weg in G_1. Sonst ersetze (v,z) durch $v \to w \xrightarrow[T]{+} z$ in W und erhalte einen Weg in G_1.

(c) Sei $e'' = (x,y) \in E$ ein Pfeil von G mit $e'' \notin T$, $(v,w) \neq e'' \neq (v,z)$.

„$\Longrightarrow$“: Sei e'' reduzierbar in G und W_1 der reduzierende Weg. Falls (v,z) kein Pfeil von W_1 ist, ist nichts zu zeigen. Also nehmen wir an, daß $(v,z) \in W_1$ ist. Ersetzen wir (v,z) in W_1 durch $v \to w \xrightarrow[T]{+} z$, so ergibt sich ein Weg W_2 in G_1. Dabei ist e'' kein Pfeil von W_2, weil er kein Baumpfeil ist. Also ist W_2 ein reduzierender Weg für e'' in G_1.

„$\Longleftarrow$“: Sei $e'' = (x,y)$ reduzierbar in G_1. Falls es einen reduzierenden Weg W_1 in G_1 für e'' ohne den Pfeil (v,w) gibt, ist W_1 auch ein Weg in G und damit ein reduzierender Weg. Andernfalls ist der Pfeil (v,w) in jedem reduzierenden Weg enthalten. Sei u der erste Knoten auf dem Weg $w \xrightarrow[T]{+} v$, also $w \xrightarrow[T]{} u \xrightarrow[T]{*} v$.

Bezeichne die Knoten auf dem Weg $root \xrightarrow[T]{*} w$ mit $w_1, \ldots, w_s$.

Dann existiert kein $a <_T w_s, a \neq w_i, 1 \leq i < s$ mit $(a, w_s) \in E$, sonst wäre w im DFS-Durchlauf schon vorher besucht worden. Die Knoten $w_1, \ldots, w_s$ bilden Separatoren für alle Knoten a, b mit $a < w_s \leq b$, $a \neq w_i$, d.h. jeder Weg von a nach b geht mindestens durch ein $w_i, 1 \leq i < s$.

Ann.: $x <_T u$. Dann geht jeder Weg W_1 von x nach y mit Pfeil (v,w) durch ein w_i (mit $a = x$ und $b = y$), also $W_1 = x \xrightarrow{+} w_i \xrightarrow{+} v \to w$. Aber dann ist der Weg $W_2 = x \xrightarrow{+} w_i \xrightarrow[T]{+} w_s = w$ ein Weg ohne den Pfeil (v,w). Widerspruch.

Also gilt $x \geq u$.

Mit demselben Argument und $a = z, b = w$ gibt es einen Weg von z nach w_j. Also gibt es einen Weg W von z nach w, der keinen Knoten $h >_T u$ enthält. Da der Pfeil $e'' = (x,y)$ nicht in W enthalten ist (sonst Widerspruch mit $h = x >_T u$), kann in W_1 der Pfeil (v,w) gestrichen und durch $v \to z \xrightarrow[W]{+} w$ ersetzt werden. Dies ist dann ein reduzierender Weg für (x,y) in G. □

Das Lemma 5.5. liefert den in Tabelle 5.3 dargestellten Pseudocode, wobei wir annehmen, daß das Feld *backpoint* schon gemäß Lemma 5.4. berechnet wurde. Falls ein Rückwärtspfeil eingefügt wird, erfolgt nach dem Argument aus Lemma 5.4. eine Reduzierung des eventuell vorhandenen Rückwärtspfeiles. Außerdem benötigen wir ein Feld *standfor*, das eine Abbildung $E \to E$ modelliert. Sie wird als identische Abbildung initialisiert und gemäß der Ersetzung von Querpfeilen aktualisiert. Außerdem sei die Funktion `lca` gegeben, die den nächsten gemeinsamen Vorfahren von zwei Knoten bezüglich eines Baumes berechnet.

Rückwärtspfeile 2

Setzen wir voraus, daß Pfeile nach den ersten drei Lemmata reduziert sind, so haben wir einen Graphen erhalten, der nur Baumpfeile und für jeden Knoten höchstens einen Rückwärtspfeil

```
forall e ∈ E do
  standfor(e) := e;
end;
forall e = (v, z) ∈ C do
  w := lca(v, z, T);
  if w < backpoint(v)
    then add (v, w) to G;
         delete (v, backpoint(v)) from G;
         standfor((v, w)) := (v, z);
         backpoint(v) := w;
  end;
  delete (v, z) from G;
end;
```

Tabelle 5.3: Pseudocode Lemma 5.5.

enthält.
Nun entwickeln wir ein bottom-up-Verfahren, um für jeden Rückwärtspfeil zu prüfen, ob er reduzierbar ist oder nicht.
Sei dazu v ein Knoten mit zugehörigem Rückwärtspfeil (v, w).
Ähnlich dem Lemma 5.5. werden zuerst Kriterien aufgestellt, die zeigen, ob (v, w) reduzierbar ist oder nicht.
Falls keine der beiden Situationen gegeben ist, wird (v, w) durch einen anderen Rückwärtspfeil ersetzt. Dies macht nur Sinn, wenn die wesentlichen Eigenschaften des Graphen dabei unberührt bleiben. Diese eingefügten Pfeile müssen dann später, wie in Lemma 5.5., durch ihre Originalpfeile ersetzt werden. Außerdem ist im letzten Fall noch nichts über die Reduzierbarkeit ausgesagt. Entscheidend dabei ist, daß der eingefügte Rückwärtspfeil von einem bezüglich des DFS-Baumes T kleineren Knoten ausgeht. So können mit Hilfe des anschließend vorgestellten Verfahrens die Rückwärtspfeile von den Blättern ausgehend überprüft werden.

Lemma 5.6. *Seien $G = (V, E)$ ein stark zusammenhängender Graph mit DFS-Partitionen T, B, C, F, $C = F = \emptyset$, $(v, w) \in B$ ein Rückwärtspfeil und $z := lowpoint(v)$. Dann gilt:*

(i) *Falls $z \geq v$ gilt, so ist (v, w) nicht reduzierbar.*

(ii) *Falls $z < v$ gilt, so setze $s := \min(\{z\} \cup \{t \in V : (z, t) \in E\})$, und sei $x \in V$, so daß $v \xrightarrow[T]{+} x$ und $(x, z) \in B$ ist. Dann gilt:*

  (a) *Falls $w \geq s$ ist, so ist (v, w) reduzierbar in G.*

  (b) *Falls $w < s$ ist, so gilt:*

    (1) *Der Pfeil (v, w) ist reduzierbar in G genau dann, wenn der Pfeil (z, w) in dem Graphen $G_1 := (V, E_1) := (V, E \backslash \{(v, w)\}) \cup \{(z, w)\}$ reduzierbar ist.*

(2) $G_1^* = G^*$

(3) *Sei* $e = (a, b) \in B$ *ein Pfeil mit* $(v, w) \neq e \neq (z, w)$, *und es existiere kein Weg* $v \xrightarrow[T]{*} a$. *Dann ist* e *reduzierbar in* G *genau dann, wenn* e *reduzierbar in* G_1 *ist.*

Beweis:

(i) Aus $z \geq v$ folgt sofort $z = v$. Da die Partitionen C und F leer sind, ist (v, w) damit der einzige Pfeil im Subbaum von v aus (bzgl. T), der aus diesem herausführt. Dann ist $(V, E \backslash \{(v, w)\})$ nicht stark zusammenhängend und (v, w) nicht reduzierbar.

(ii) Sei nun $z < v$.

(a) Sei $w \geq s$. Dann ist $v \xrightarrow[T]{+} x \to z \to s \xrightarrow[T]{*} w$ ein reduzierender Weg für (v, w) in G.

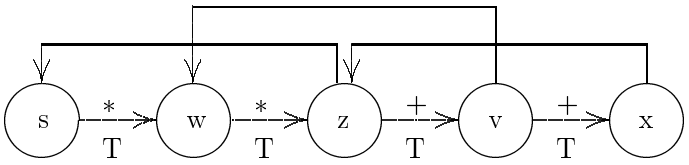


Abbildung 5.5: Lemma 5.6 (ii) (a)

(b) Sei $w < s$. Es folgt, daß $(z, w) \notin E$ ist.

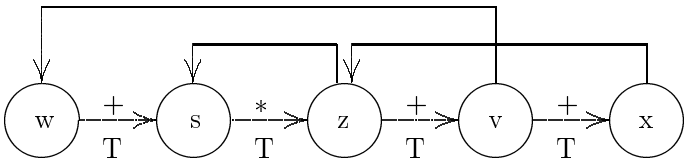


Abbildung 5.6: Lemma 5.6 (ii) (b)

(1) „$\Longrightarrow$“: Sei (v, w) reduzierbar in G. Dann existiert ein reduzierender Weg W_1 von v nach w in G. W_1 ist auch ein Weg in G_1. Dieser kann durch $z \xrightarrow[T]{+} v$ zu einem Weg $W_2 = z \xrightarrow[T]{+} w$ erweitert werden. Da (z, w) ein Rückwärtspfeil in G_1 ist und der Weg W_2 einfach ist, gilt $(z, w) \notin W_2$. W_2 ist dann ein reduzierender Weg für (z, w) in G_1.
„$\Longleftarrow$“: Sei W_1 ein reduzierender Weg für (z, w) in G_1. Nach Voraussetzung existiert ein Weg $v \xrightarrow[T]{+} x \to z$ in G. Wird dieser durch W_1 zu $W_2 = v \xrightarrow[T]{+} x \to z \to w$ erweitert, so erhalten wir einen reduzierenden Weg für (v, w) in

G. Dabei ist (v, w) weder in $v \xrightarrow[T]{+} x$ enthalten (alles Baumpfeile) noch gleich (x, z), da $x \neq v$.

(2) Nach Voraussetzung gilt $G^* = \mathsf{L}$. Es bleibt also nur $G^* \subseteq G_1^*$ zu zeigen. Sei dazu $x, y \in V$, so daß es einen Weg $W = (x, \ldots, y)$ in G gibt. Falls der Pfeil (v, w) aus W ist, ersetzen wir ihn durch $v \xrightarrow[T]{+} x \to z \to w$, und erhalten so einen Weg von x nach y in G_1.

(3) Sei e nach Voraussetzung gewählt.
„$\Longrightarrow$“: Sei $e = (a, b)$ reduzierbar in G und W der reduzierende Weg mit dem Pfeil (v, w). (Der Fall $(v, w) \notin W_1$ ist trivial.) Um a und b in G_1 miteinander zu verbinden, ersetzten wir in W den Pfeil (v, w) durch den Weg $v \xrightarrow[T]{*} x \to z \to w$ und erhalten so einen Weg W_1. Wir zeigen nun, daß W_1 ein Weg in G_1 ist, d.h. daß der Pfeil (v, w) nicht im Weg $v \xrightarrow[T]{+} x \to z \to w$ enthalten ist.
Nach Voraussetzung ist $e \in B$, also $e \notin v \xrightarrow[T]{+} x$. Falls $(a, b) = (x, z)$ gilt, folgt sofort $a = x$. Damit würde es einen Weg $v \xrightarrow[T]{+} a = x$ geben, was im Widerspruch zur Voraussetzung steht. Außerdem ist $e = (a, b) \neq (z, w)$, da $e \in E$, aber $(z, w) \notin E$ ist.
„$\Longleftarrow$“: Sei nun W_1 ein reduzierender Weg für (a, b) in G_1, wobei $(z, w) \in W_1$ sei. (Der Fall $(v, w) \notin W_1$ ist trivial.) Durch die Ersetzung von (z, w) durch den Weg $z \xrightarrow[T]{+} v \to w$ erhalten wir einen Weg W von a nach b. Dieser ist nach denselben Argumenten wie in „$\Longrightarrow$“ ein reduzierender Weg in G. □

Wir wollen das Lemma 5.6. nun nacheinander auf alle Rückwärtspfeile anwenden. Dabei starten wir bei den Blättern des DFS-Baumes und arbeiten die Rückwärtspfeile gemäß ihrer Entfernung von der Wurzel ab.
Seien dazu $C = \emptyset$, $F = \emptyset$ und $|\{(v, w) \in B\}| \leq 1$ für jedes $v \in V$.
Wir wollen zeigen, daß nach der induktiven Anwendung für alle Knoten $v \in V$ folgendes gilt: Falls es einen Rückwärtspfeil (v, x) gibt, so ist dieser nicht reduzierbar.
Wir führen eine strukturelle Induktion über den Aufbau eines Baumes durch.
Für den Induktionsanfang sei v ein Blatt und (v, w) der zugehörige Rückwärtspfeil. Dann ist (v, w) nicht reduzierbar, da er der einzige Pfeil ist, der von v ausgeht.
Für den Induktionsschritt sei nun v ein innerer Knoten. Wir nehmen an, daß für alle Knoten $w \in \{x \mid v \xrightarrow[T]{+} x\}$ die Induktionsbehauptung gelte. Wir zeigen nun die Behauptung für die Menge $M := \{x \mid v \xrightarrow[T]{+} x\} \cup \{v\}$.
Falls es keinen Rückwärtspfeil von v aus gibt, so gilt die Behauptung trivialerweise.
Andernfalls sei (v, w) der Rückwärtspfeil und $z := \mathit{lowpoint}(v)$.
Falls $z \geq v$ gilt, so ist nach Lemma 5.4.(i) (v, w) nicht reduzierbar.
Ansonsten löschen wir (v, w) und ersetzten ihn, falls $w < s$ gilt, durch den Rückwärtspfeil (z, w). Da z in diesem Fall kleiner als v ist, gilt die Induktionsbehauptung nun für die Menge M. Falls der Rückwärtspfeil (z, s) existiert, kann dieser nach Lemma 5.4. gelöscht werden.

Wenden wir dieses Verfahren auf die Wurzel an, erhalten wir einen Graphen, der keine reduzierbaren Rückwärtspfeile enthält. Dies liefert für einen Knoten den in Tabelle 5.4 dargestellten Pseudocode, der die in einem Induktionsschritt angeführten Schritte ausführt. Dabei wird durch die induktive Anwendung das Feld *backpoint* dazu verwendet, den *lowpoint* zu berechnen. Diese wird in Tabelle 5.5 dargestellt. Sie sichert, daß zuerst alle Rückwärtspfeile aller Nachfolger im Baum überprüft werden, bevor der „eigene" Rückwärtspfeil getestet wird. Wir gehen außerdem davon aus, daß das Feld *backpoint* gemäß den Lemmata 5.3.–5.5. schon berechnet wurde.

```
procedure ReduceBackward(v : V)
  z := min {backpoint(w) | (v, w) ∈ T};
  if v ≠ z
    then if backpoint(v) < backpoint(z)
           then delete (z, backpoint(z)) from G;
                add (z, backpoint(v)) to G;
                standfor((z, backpoint(v))) := standfor((v, backpoint(v)));
                backpoint(z) := backpoint(v);
         end;
         delete (v, backpoint(v)) from G;
         backpoint(v) := z;
  end;
```

Tabelle 5.4: Pseudocode `ReduceBackward` (Lemma 5.6.)

```
procedure R-test(v : V)
  forall w with (v, w) ∈ T do
    R-test(w);
  end;
  if v is not a leaf
    then ReduceBackward(v);
  end;
```

Tabelle 5.5: Pseudocode `R-test` (Lemma 5.6.)

Zusammenfassung

Die Hintereinanderausführung aller Pseudocodes sowie das nach Lemma 5.5. und 5.6. nötige Zurückersetzen der Pfeile ergibt den Algorithmus `A`, dargestellt in Tabelle 5.6.

Dabei gehen wir von einem Algorithmus `DFS` aus, der eine Klassifikation der Pfeilmenge liefert.

```
function A((V, E) : graph, root : V)
        (T, B, C, F) := DFS(G, root);
        G := (V, E);
  (i)   forall e ∈ G do
          standfor(e) := e;
        end;
  (ii)  forall v ∈ V do
          backpoint(v) := v;
        end;
  (iii) forall e ∈ F do
          delete e from E_1;
  (iv)  forall e = (v, w) ∈ B do
          backpoint(v) := min (backpoint(v), w);
        end;
  (v)   forall e = (v, w) ∈ B do
          if w ≠ backpoint(v)
            then delete (v, w) from E_1;
          end;
        end;
  (vi)  forall e = (v, z) ∈ C do
          w := lca(v, z, T);
          if w < backpoint(v)
            then add (v, w) to G;
                 delete (v, backpoint(v)) from G;
                 standfor((v, w)) := (v, z);
                 backpoint(v) := w;
          end;
          delete (v, z) from G;
        end;
  (vii) R-test(root);
  (viii) forall e ∈ G do
          delete e from G;
          add standfor(e) to G;
        end;
        return (G, (V, T))
```

Tabelle 5.6: Pseudocode Algorithmus `A`

Formal gesehen ergibt sich durch das Anwenden der verschiedenen Lemmata eine Folge von Graphen. Sei $G_0 = G$ der Ausgangsgraph, und G_{i+1} der Graph, der durch das Löschen oder das Ersetzen eines Pfeiles aus G_i entsteht, sowie G_s der letzte Graph dieser Folge. Nach jedem Lemma bleibt die Hülle der Graphen erhalten, $G_i^* = G_{i+1}^*$; wir erhalten induktiv $G^* = G_0^* = \ldots = G_s^*$.
Wir merken uns, wie schon nach Lemma 5.5. und 5.6. beschrieben, jede Ersetzung im Feld *standfor*. Die Reduzierbarkeit aller anderen Pfeile wird dabei nicht verändert.
Es ist zum einem zu beachten, daß die Klassifikation der Pfeile durch das Löschen und Ersetzen nicht verändert wird (Lemma 5.2.), zum anderen, daß die Nichtreduzierbarkeit eines Pfeiles auch für jeden Untergraphen gilt (Lemma 5.1.).
Durch die (Zurück-)Ersetzung aller Pfeile durch ihre Originale erhalten wir einen Untergraphen G' von G, der keine reduzierbaren Quer-, Vorwärts- und Rückwärtspfeile enthält und stark zusammenhängend ist.
Neben dem Feld *standfor* benötigen wir außerdem noch das Feld *backpoint*. Dieses wird gemäß den Bemerkungen zu Lemma 5.4. initialisiert. Für die anschließende Anwendung des Lemmas 5.5. werden die Werte für neu eingefügte Rückwärtspfeile aktualisiert. Für das Lemma 5.6. bezeichnet $backpoint(v)$ den *lowpoint* von v, nachdem die Aufrufe `R-test` für alle Nachfolger beendet sind.
Zusammenfassend kann festgestellt werden:

Theorem 5.1. *Der Algorithmus* `A` *berechnet einen stark zusammenhängenden Untergraphen* $G' = (V, E')$ *von* $G = (V, E)$*, der keine reduzierbaren Nichtbaumpfeile bezüglich des DFS-Baumes* T *enthält, d.h.*

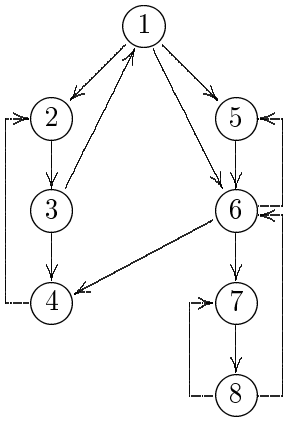
- $G'^* = \mathsf{L}$
- $\forall\, e \in E' \backslash T \;:\; (G' \backslash \{e\})^* \neq \mathsf{L}$.

Der Algorithmus hat einen Zeit- und Platzbedarf von $\mathcal{O}(|V|+|E|)$*. Außerdem gilt* $|E'| \leq 2 \cdot |V|$.

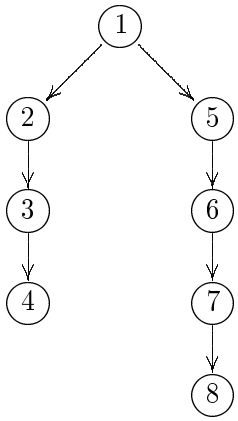
Beweis: Die Korrektheit ist nach den Lemmata 5.3.–5.6. sowie den Bemerkungen zur Entwicklung und der Zusammenfassung klar. Daher sind nur noch die Komplexitäten zu überprüfen. Zuerst verwenden wir `DFS`, um die Partitionen zu erstellen. Dies kann in linearem Zeit- und Platzbedarf ausgeführt werden [31]. Die Schritte (i), (ii), (iii), (iv), (v) und (viii) können in $\mathcal{O}(1)$ pro Pfeil ausgeführt werden. Außerdem kann die Berechnung des nächsten gemeinsamen Vorfahren in Schritt (vi) mit einer speziellen Datenstruktur (z.B. von Harel und Tarjan [15]) in $\mathcal{O}(1)$ Zeit pro Pfeil und insgesamt $\mathcal{O}(|V|)$ Platz erfolgen. `R-test` in Schritt (vii) kann für jeden festen Knoten v linear zur Anzahl der ausgehenden Pfeile berechnet werden. Daher ist der gesamte Zeitaufwand linear zur Anzahl der Pfeile im gesamten Graphen zu Beginn des Aufrufes von `R-test`. Diese Anzahl beträgt höchstens $2|V|$.
Also ist der Algorithmus `A` linear zeit- und platzbeschränkt. □

Beispiel

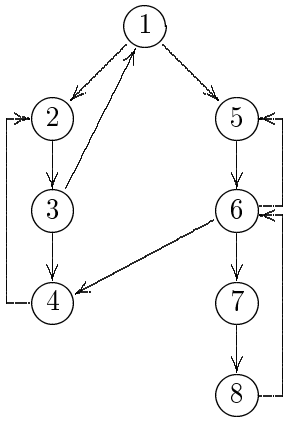
Da der Algorithmus `A` in seiner Gesamtheit sehr lang und in Teilen auch kompliziert ist, wird hier mit Hilfe eines Beispiels noch einmal ein Durchgang durch den Algorithmus erfolgen. Wir betrachteten dazu die Graphen in Abbildung 5.7.



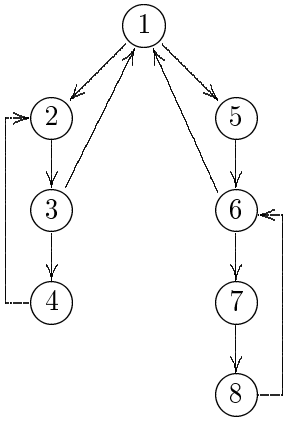
Originalgraph G



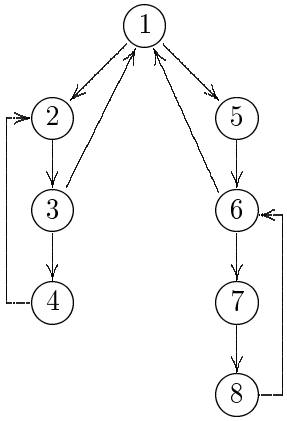
DFS Baum T zu G



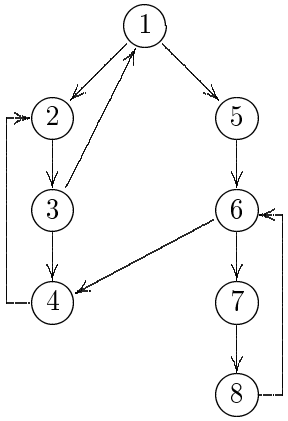
Rückwärtspfeile 1



Querpfeil $(6, 4)$



nach `R-test`



Ergebnis `A`(G)

Abbildung 5.7: Beispiel zum Algorithmus `A`

Der stark zusammenhängende Graph G sei gegeben, dazu der DFS-Baum T. Es ergibt sich für die restlichen Pfeile folgende Klassifikation. Der Pfeil $(1, 6)$ ist der einzige Vorwärtspfeil, $(6, 4)$ ist ein Querpfeil, die restlichen sind Rückwärtspfeile. Die zugrundeliegende Ordnung ist die natürliche $(\mathbb{N}, \leq)$.
Zuerst werden die Felder *standfor* und *backpoint* initialisiert ((i) und (ii)). Danach wird der

einzige Vorwärtspfeil in Schritt (iii) gelöscht.
In Schritt (iv) wird das Feld *backpoint* wie folgt berechnet:

	1	2	3	4	5	6	7	8
backpoint	1	2	1	2	5	5	7	6

Mit Hilfe des Lemmas 5.4. (Schritt (v)) kann anschließend der Rückwärtspfeil $(8, 7)$ eliminiert werden, da $backpoint(8) = 6$ gilt.
Dies ergibt das Zwischenergebnis, dargestellt im dritten Graphen.
Das Lemma 5.5. (Schritt (vi)) ersetzt den Querpfeil $(6, 4)$ durch den Pfeil vom Knoten 6 nach $\texttt{lca}(6, 4) = 1$. Außerdem kann der Rückwärtspfeil $(6, 5)$ gelöscht werden, dies ergibt die Anwendung von Lemma 5.4. auf den Knoten 6. Die Felder werden zu $standfor((6, 1)) = (6, 4)$ und

	1	2	3	4	5	6	7	8
backpoint	1	2	1	2	5	1	7	6

aktualisiert.
Im letzten Schritt werden die restlichen Rückwärtspfeile mittels `R-Test` überprüft.
Durch die rekursive Aufrufstruktur werden die Knoten in der Reihenfolge 4, 3, 2, 8, 7, 6, 5, 1 abgearbeitet.
Da der Knoten 4 ein Blatt ist, wird der Pfeil $(4, 2)$ nicht gelöscht. Dies gilt auch für den Pfeil $(8, 6)$, weil der Knoten 8 ein Blatt ist. Die Pfeile $(6, 1)$ und $(3, 1)$ werden ebenfalls nicht gelöscht. Der jeweilige *lowpoint* ist der Knoten 1, also trifft die Bedingung (i) in Lemma 5.6. zu. Abschließend erfolgt die Rückersetzung der Pfeile, und es ergibt sich der Graph im letzten Bild.
Das Ergebnis ist nicht inklusionsminimal; die Baumpfeile $(1, 2)$ und $(3, 4)$ sind reduzierbar.

5.2.2 Reduktion von Baumpfeilen

Wir wollen nun mit Hilfe des im vorherigen Abschnitt entwickelten Algorithmus `A` einen Algorithmus `TransRed-Simon` entwickeln, der eine transitive Reduktion berechnet. Als Zwischenschritt werden wir dazu einen nichtreduzierbaren Wurzelbaum berechnen, der also nur nicht reduzierbare Pfeile und einen Wurzelbaum enthält.

Sei $G = (V, E)$ ein stark zusammenhängender Graph. Wir wenden nun den Algorithmus `A` auf den transponierten Graphen G^T an und erhalten so den Graphen G_1 mit zugehörigem DFS-Baum T_1. Dann ist G_1^T Untergraph von G und T_1 ein Wurzelbaum von G_1. Eine erneute Anwendung von `A` auf den Graphen G_1^T ergibt einen Graphen G_2 mit DFS-Baum T_2. Nach den Ergebnissen aus dem letzten Abschnitt sind alle Pfeile in $G_2 \setminus (T_1 \cap T_2)$ nicht reduzierbar. Um einen nicht reduzierbaren Wurzelbaum zu erhalten, müssen nur noch alle Pfeile aus $T_1 \cap T_2$ überprüft werden, ob sie für die Wurzelbaumeigenschaft notwendig sind.
In der Originalfassung der Arbeit von Simon wurde ein linearer Ansatz zur Überprüfung dieser Pfeile präsentiert. Dieser Teil enthält einen Fehler.

function `ReduceTreeEdges`$(G = (V, E) : graph, T_1 : graph, T_2 : graph, root : V)$
  $H_1 := G$;
  $H_2 := G$;
  forall $e = (x, y) \in (T_1 \cap T_2)$
    if $x \xrightarrow[H_1 \backslash \{e\}]{*} y$
      then $H_1 := H_1 \backslash \{e\}$;
    end;
    if $x \xrightarrow[H_2 \backslash \{e\}]{*} root$
      then $H_2 := H_2 \backslash \{e\}$;
    end;
  end;
  return (H_1, H_2)

Tabelle 5.7: Pseudocode `ReduceTreeEdges`

Um den allgemeinen Aufbau der Arbeit von Simon hier zu erhalten, entwickeln wir einen Ansatz, der dieses Problem mit einem quadratischen Verfahren löst, das aber eigentlich zu mächtig ist. Wir kommen darauf später zurück.
Angelehnt an das Verfahren aus Kapitel 4 werden alle Pfeile aus $T_1 \cap T_2$ sowohl auf ihre Reduzierbarkeit als auch auf ihre Notwendigkeit bezüglich des Wurzelbaumes überprüft. In dem Pseudocode aus Tabelle 5.7 werden zwei Graphen berechnet, einer jeweils für jede Eigenschaft.

Lemma 5.7. *Seien $G = (V, E)$ ein stark zusammenhängender Graph, $|E| \leq 2\,|V|$, T_1 ein Baum und $T_2 \subseteq G$ ein Wurzelbaum jeweils zur Wurzel $root \in V$. Dann berechnet* `ReduceTreeEdges`$(G, T_1, T_2, root)$ *zwei Graphen in Laufzeit $\mathcal{O}(|V|^2)$ mit folgenden Eigenschaften:*

(i) $H_2 \subseteq H_1 \subseteq G$

(ii) $H_1^* = \mathsf{L}$

(iii) $\forall\, v \in V \;:\; v \xrightarrow[H_2]{*} root$

(iv) $\forall\, e \in H_2 \cap T_1 \cap T_2 \;:\; (H_1 \backslash \{e\})^* \neq \mathsf{L}$.

Beweis: Um die Ähnlichkeit zwischen dem Algorithmus `ReduceTreeEdges` und dem Ansatz aus Kapitel 4 zu verdeutlichen, zeigen wir die Parallelen auf. Die graphentheoretische Bedingung $x \xrightarrow[H_1 \backslash \{e\}]{*} y$ läßt sich relational als $(H_1 \cap \overline{e})^*_{xy}$ schreiben.
Es folgt, daß die erste **if**-Bedingung stärker ist als die zweite.
Daraus folgt sofort die Eigenschaft (i). Die Eigenschaften (ii) und (iv) folgen sofort aus den Betrachtungen in Kapitel 4. Dabei kann natürlich die Inklusionsminimalität nur bezüglich der überprüften Mengen gewährleistet werden.

Auch die Eigenschaft (iii) folgt, weil der Graph zu Beginn einen Wurzelbaum enthält und Pfeile nur dann gelöscht werden, wenn diese Eigenschaft erhalten bleibt.
Die Laufzeit $\mathcal{O}(n^2)$ ergibt sich aus der Tatsache, daß die **forall**-Schleife höchstens $|V|$-mal durchlaufen wird ($|T_1 \cap T_2| \leq |V|$) und der Test, ob es einen Weg gibt, z.B. mit Hilfe eines DFS-Durchlaufes, in Zeit $\mathcal{O}(|V| + |E|) = \mathcal{O}(|V|)$ durchgeführt werden kann. □

Wir benutzen die Funktion `ReduceTreeEdges`, um den oben beschriebenen Ansatz umzusetzen. Der so entwickelte Pseudocode `Theo2` in Tabelle 5.8 entspricht dabei dem Aufbau der zugrundeliegenden Arbeit. Wir halten aber auch fest, daß der dort berechnete Graph G_3 schon eine transitive Reduktion ist. Deshalb ist der hier gewählte Ansatz zu mächtig. Die Suche nach linearen Ansätzen für `ReduceTreeEdges` ist bisher fehlgeschlagen. Wir verwenden deshalb diesen quadratischen Ansatz.

```
function Theo2(G = (V, E) : graph, root : V)
  (G_1, T_1) := (A(G^T, root))^T
  (G_2, T_2) := A(G_1, root);
  (G_3, G_4) := ReduceTreeEdges(G_2, T_1, T_2, root);
  return (G_3, G_4)
```

Tabelle 5.8: Pseudocode `Theo2` zu Theorem 5.2.

Theorem 5.2. *Seien $G = (V, E)$ stark zusammenhängend und $root \in V$. Dann berechnet* `Theo2`$(G, root)$ *Graphen in Laufzeit $\mathcal{O}(|V|^2)$ mit folgenden Eigenschaften:*

(i) $G_3{}^* = \mathsf{L}$

(ii) $\forall\, v \in V \;:\; v \xrightarrow[G_4]{*} root$

(iii) $\forall\, e \in G_4 \;:\; (G_3 \backslash \{e\})^* \neq \mathsf{L}$

(iv) $G_4 \subseteq G_3 \subseteq G$.

Beweis:

(I) Die erste Anwendung des Algorithmus `A` bringt nach Theorem 5.1. folgende Eigenschaften:

(a) $G_1{}^* = \mathsf{L}$

(b) $\forall\, e \in G_1 \backslash T_1 \;:\; (G_1 \backslash \{e\})^* \neq \mathsf{L}$

(c) $T_1 \subseteq G_1 \subseteq G$

(d) T_1 Wurzelbaum zur Wurzel $root$.

Dabei werden die Eigenschaften durch die Transposition erhalten (Lemma 5.1.).

(II) Nach (I)(a) ist G_1 stark zusammenhängend, also ist das Theorem 5.1 anwendbar, es ergeben sich folgende Eigenschaften:

(a) ${G_2}^* = \mathsf{L}$

(b) $\forall\, e \in G_2 \backslash T_2 \;:\; (G_2 \backslash \{e\})^* \neq \mathsf{L}$

(c) $T_2 \subseteq G_2 \subseteq G_1$

(d) T_2 Baum zur Wurzel *root*.

(III) Aus den Eigenschaften (I)(c) und (d), (II)(c) und (d) sowie (II)(a) folgt mit dem Algorithmus `ReduceTreeEdges` und Lemma 5.7.:

(a) $\forall\, x \in V : x \xrightarrow[G_4]{*} root$

(b) ${G_3}^* = \mathsf{L}$

(c) $G_4 \subseteq G_3 \subseteq G_2$

(d) $\forall\, e \in G_4 \cap T_1 \cap T_2 \;:\; (G_3 \backslash \{e\})^* \neq \mathsf{L}$.

Es folgt zuerst (i) aus (III)(b), (ii) aus (III)(a), sowie (iv) aus (I)(c),(II)(c) und (III)(c).
Zum Beweis von (iii) sei $e \in G_4$.

- Falls $e \in G_4 \backslash T_1$, folgt die Behauptung aus (I)(b) mit Hilfe von Lemma 5.1. und $G_3 \subseteq G_1$.
- Falls $e \in G_4 \backslash T_2$, folgt sie aus (II)(b) mit Hilfe von Lemma 5.1. und $G_3 \subseteq G_2$.
- Falls $e \in G_4 \cap T_1 \cap T_2$, folgt die Behauptung aus (III)(b).

Also ist e nicht reduzierbar in G_3. □

Nach Theorem 5.2. haben wir nun einen nichtreduzierbaren Wurzelbaum G_4 erhalten, d.h. jeder Pfeil ist sowohl im Graphen G_3 nicht reduzierbar als auch notwendig für die Wurzelbaumeigenschaft in G_4. Durch zweimaliges Anwenden des Theorems erhalten wir eine transitive Reduktion.

```
function TransRed-Simon(G : graph)
  (G1, T1) := Theo2(G, root);
  (G2, T2) := (Theo2(G1^T, root))^T;
  return T1 ∪ T2
```

Tabelle 5.9: Pseudocode `TransRed-Simon` zu Theorem 5.3.

Wir erhalten also einen nichtreduzierbaren Wurzelbaum T_1 sowie einen nicht reduzierbaren Baum T_2, die Vereinigung ist eine transitive Reduktion. Die Korrektheit wird formal durch folgendes Theorem gesichert:

Theorem 5.3. *Sei $G = (V, E)$ ein stark zusammenhängender Graph. Dann berechnet $T := \texttt{TransRed} - \texttt{Simon}(G)$ in Laufzeit $\mathcal{O}(n^2)$ eine transitive Reduktion von G mit Güte 2, d.h. es gilt:*

(i) $T^* = \mathsf{L}$

(ii) $\forall\, e \in E(T) \;:\; (T \backslash \{e\})^* \neq \mathsf{L}$

(iii) $T \subseteq G$

(iv) $\dfrac{|E(T)|}{\mathcal{OPT}(G)} < 2.$

Beweis: Nach Theorem 5.2. gilt folgendes:

(I) (a) ${G_1}^* = \mathsf{L}$

(b) $\forall\, v \in V \;:\; v \xrightarrow[T_1]{*} root$

(c) $\forall\, e \in T_1 \;:\; (G_1 \backslash \{e\})^* \neq \mathsf{L}$

(d) $T_1 \subseteq G_1 \subseteq G$

(II) (a) ${G_2}^* = \mathsf{L}$

(b) $\forall\, v \in V \;:\; root \xrightarrow[T_2]{*} v$

(c) $\forall\, e \in T_2 \;:\; (G_2 \backslash \{e\})^* \neq \mathsf{L}$

(d) $T_2 \subseteq G_2 \subseteq G_1.$

Wir halten fest, daß $T = T_1 \cup T_2$ ist.
Zuerst zeigen wir einige Hilfsbehauptungen:

- $T = T_1 \cup T_2 \subseteq G_1 \cup G_2 = G_1.$
- $T_1 \subseteq G_2.$

  Dazu sei $e \in T_1$. Dann ist nach (I)(a) und (c) e nicht reduzierbar in T_1. Mit Hilfe von Lemma 5.1. und $G_2 \subseteq G_1$ folgt, daß e nicht reduzierbar in G_2 ist.

  Ann.: $e \notin G_2$. Nach (II)(a) ist G_2 stark zusammenhängend, also ist e reduzierbar in G_2. Widerspruch

  Also gilt $e \in G_2$, es folgt die Behauptung $T_1 \subseteq G_2$.
- $T = T_1 \cup T_2 \subseteq G_2 \cup G_2 = G_2.$

Wir wollen nun zeigen, daß T eine transitive Reduktion von G ist, also (i) – (iii).

(i) Dies weisen wir komponentenweise nach: Seien dazu $x, y \in V$ zwei Knoten. Dann ist durch $x \xrightarrow[T_1]{*} root \xrightarrow[T_2]{*} y$ ein Weg von x nach y in T gegeben.

(ii) Sei $e = (x, y) \in E(T)$. Zum Beweis, daß e nicht reduzierbar in T ist, unterscheiden wir folgende Fälle:

- $e \in T_1$: Nach (I)(a) und (c) ist e nicht reduzierbar in G_1, es folgt mit Hilfe von 5.1. und $T \subseteq G_1$, daß e nicht reduzierbar in T ist.
- $e \in T_2$: Nach (II)(a) und (c) ist e nicht reduzierbar in G_2, es folgt mit Hilfe von 5.1. und $T \subseteq G_2$, daß e nicht reduzierbar in T ist.

(iii) Es gilt $T = T_1 \cup T_2 \subseteq G_1 \cup G_2 \subseteq G$.

(iv) Die Güte folgt sofort aus Lemma 3.3.

Die quadratische Laufzeit sichert Theorem 5.2.. □

Bemerkung 5.2. *In den Theoremen 5.2. und 5.3. hängt die quadratische Laufzeit nur vom Algorithmus* `ReduceTreeEdges` *ab.*

5.2.3 Theoretische Eigenschaften

Zusammenfassend halten wir fest, daß wir einen Algorithmus `TransRed-Simon` mit quadratischer Laufzeit entwickelt haben, der eine transitive Reduktion berechnet. Die Entwicklung sichert außerdem zu, daß der Algorithmus die Güte 2 besitzt. Dies zeigt Theorem 5.1. sowie das Lemma 3.3..

5.3 Originalalgorithmus

Der in Abschnitt 5.2. vorgestellte Algorithmus stammt zum großen Teil aus der Arbeit von Simon [28]. Allerdings ist ein fehlerhafter Teil[1] [29] durch einen theoretisch langsameren, aber korrekten ersetzt worden. Die ursprüngliche Absicht der Arbeit von Simon war es, einen linearen Ansatz zu entwickeln. Daher gab es an Stelle der hier vorgestellten Prozedur

[1] Sehr geehrter Herr Kasper,

ich erinnere mich nur noch dunkel an diese Geschichte. Nach diesem Workshop in Aachen, hatte ich diese Arbeit Bob Tarjan gegeben, er war damals auch gerade an diesem Thema interessiert. Er hat ein Gegenbeispiel fuer die Reduktion der Baumkanten gefunden. Wir haben dann noch ein paar Monate Briefe geschrieben, wie man den Fehler beheben kann. Ich war der Meinung, dass ich die falsche Induktion gewaehlt hatte, die richtige koennte die Rueckkehrreihenfolge der DFS-Prozedur sein. Tarjan hat dann mit Karp ein Paper ueber diese Dinge fuer PRAMs geschrieben, das auch einen kleinen sequenziellen Teil zur trans. Reduktion hatte (dasselbe wie bei mir, aber in O(n log n), ueber optimal branchings, ich konnte dieses Resultat aber nicht verfizieren und optimal branchings waren bis dahin bereits fuenf oder sechsmal korrigiert worden). PRAMs waren aber nicht mein Fall. Die Diskussion ist dann irgendwann eingeschlafen. Andere Dinge waren wichtiger. Der Zusammenhang zu Dominators war der damaligen Disskusionsgruppe bekannt (wem auch sonst?). Aber mehr weiss ich nicht. Es war mein letzter Beitrag zu dieser Community.

Gruss

Klaus Simon

`ReduceTreeEdges` einen linearen Ansatz, der mit Hilfe von Dominatoren versucht, Baumpfeile zu reduzieren. Diesen wollen wir in seiner Idee hier vorstellen und zeigen, warum er nicht korrekt ist.
Wir betrachten dazu eine spezielle Form von dominierenden Knoten.

Definition 5.1. *Seien $G = (V, E)$ ein stark zusammenhängender Graph und $root \in V$ ein ausgezeichneter Knoten. Ein Knoten v dominiert einen Knoten w bezüglich root, falls jeder Weg von w nach root, den Knoten v enthält. v heißt direkter Dominator von w bezüglich root, falls v w bezüglich root dominiert und jeder andere Dominator von w v dominiert.*

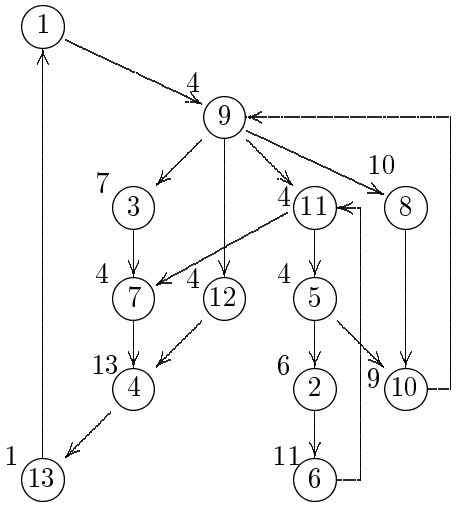
Aufgabe der Funktion `ReduceTreeEdges` ist es, zu einem Wurzelbaum T_1 und einem Baum T_2 eines Graphen G mit gemeinsamer Wurzel *root* Pfeile zu reduzieren, um einen Graphen zu erhalten, der einen nichtreduzierbaren Wurzelbaum enthält.
Wir benötigen einen linearen Algorithmus `ImmDom`$(G, root)$ [15, 14], der zu einem Graphen G für jeden Knoten den direkten Dominator bezüglich *root* berechnet.
Dazu wird in der Arbeit von Simon der in Tabelle 5.10 dargestellte Algorithmus `ReduceTree-Edges-Simon` vorgeschlagen.

```
function ReduceTreeEdges-Simon(G : graph, T1 : graph, T2 : graph, root : V)
  H2 := G;
  dom := ImmDom(G, root);
  forall e = (x, y) ∈ (T1 ∩ T2)
    if dom[x] ≠ y
      then H2 := H2\{e};
    end;
  end;
  return (G, H2)
```

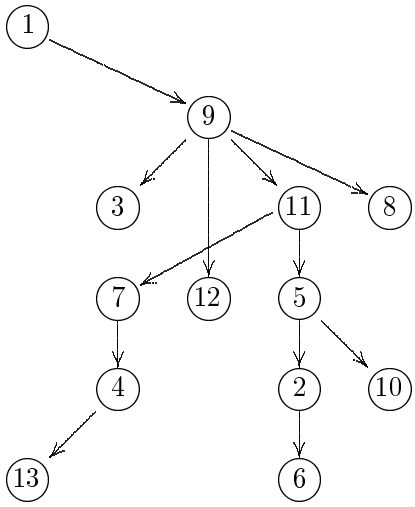
Tabelle 5.10: Pseudocode `ReduceTreeEdges-Simon`

Wir wollen nun die Argumentation aus der Arbeit von Simon [28, Seite 257] vorstellen.
Dazu müssen wir zwei Richtungen betrachten. Sei dazu $e = (x, y) \in (T_1 \cap T_2)$ ein Pfeil. Im ersten Fall sei der direkte Dominator von x der Knoten y. Das heißt, jeder Weg vom Knoten x zur Wurzel *root* führt durch den Knoten y. Es folgt, daß der Pfeil (x, y) notwendig für die Wurzelbaumeigenschaft ist. Im anderen Fall gibt es einen Weg zur Wurzel ohne den Pfeil (x, y). Simon folgert daraus, daß der Pfeil für die Wurzelbaumeigenschaft nicht benötigt wird. Das ist aber falsch! Dies wird mit Hilfe des folgenden Beispiels verdeutlicht werden.
Wir betrachten in Abbildung 5.8 den Beispielgraphen G im Bild 1 mit den direkten Dominatoren sowie den Wurzelbaum T_1 und Baum T_2 im zweiten und dritten Bild.
Die folgenden Pfeile liegen sowohl in T_1 als auch in T_2 und werden im Algorithmus überprüft: $(4, 13), (11, 7), (5, 10), (2, 6), (9, 11), (12, 4), (7, 4)$.

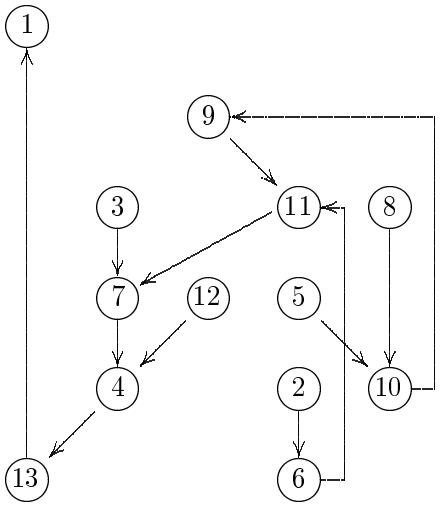
Die Pfeile $(4, 13), (2, 6), (12, 4)$ und $(7, 4)$ werden nicht gelöscht, da die direkten Dominatoren der Anfangspunkte die jeweiligen Endpunkte der Pfeile sind.
Die restlichen Pfeile $(11, 7), (5, 10)$ und $(9, 11)$ werden gelöscht.
Es ergibt sich der Graph im Bild 4. Dieser ist aber kein Wurzelbaum.
Der Grund hierfür ist in den beiden Pfeilen $e_1 = (5, 10)$ und $e_2 = (11, 7)$ zu suchen. Der Knoten 4 ist jeweils der direkte Dominator der Knoten 5 und 10. Für den Knoten 5 gibt es zwei verschiedene Wege zum Knoten 4, einer benutzt e_1, der andere e_2. Dies gilt ebenfalls für den Knoten 10.
Löschen wir nun den Pfeil e_1, so verändert sich der direkte Dominator des Knoten 10 zu 7.



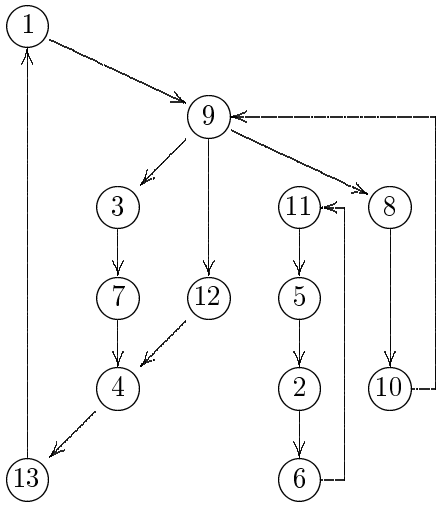
G mit direkten Dominatoren



Baum T_1 zu G



Wurzelbaum T_2 zu G



Ergebnis

Abbildung 5.8: Beispiel zu `ReduceTreeEdges-Simon`

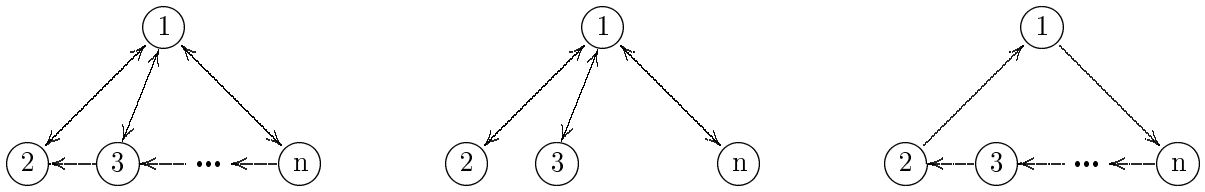
Die Berechnung der Dominatoren liefert nur solche im Originalgraphen, aber kann nicht berücksichtigen, wie sich die Dominatoren verändern, falls ein Pfeil gelöscht wird. So müßten nach jedem Löschen eines Pfeiles die direkten Dominatoren erneut berechnet werden. Dies erhöht die Laufzeit in quadratische Laufzeit.
Der Fehler des Ansatzes ist aber nicht nur in der Laufzeit zu suchen. Nehmen wir nun einen iterativen Dominatoralgorithmus an und betrachten die oben gegebene Situation. Der Pfeil e_1 wird gelöscht, e_2 aber nicht. Es entsteht ein Wurzelbaum. Wir benötigen aber noch die Eigenschaft der Nichtreduzierbarkeit für jeden Pfeil. Der Pfeil e_2 ist aber reduzierbar in G, denn es gibt einen Weg $11 \to 5 \to 10 \to 9 \to 3 \to 7$ in G.
Es muß also außerdem noch eine Anpassung des Graphen G erfolgen. Dieses wurde bei der Entwicklung des Algorithmus `ReduceTreeEdges` durch das Löschen von reduzierbaren Pfeilen berücksichtigt und führte zum Graphen H_1.

5.4 Beispiel

Den Algorithmus `A` haben wir schon an einem Beispiel erläutert. Wir wollen nun zu dem entwickelten Algorithmus `TransRed-Simon` ein worst-case Beispiel betrachten.

Worst-case Beispiel

Dazu betrachten wir die Abbildung 5.9 und den Graphen G auf n Knoten.
Das Ergebnis ist im zweiten Bild dargestellt. Es besteht aus Pfeilen, die von der Wurzel 1 wegführen, dem nichtreduzierbaren Baum, und solchen die zur Wurzel hinführen, dem nichtreduzierbaren Wurzelbaum. Das Ergebnis ist eine transitive Reduktion. Dieser Graph enthält $2n - 2$ Pfeile.
Mit Hilfe des MEG im dritten Bild, der n Pfeile enthält, folgt die Güte $2 - \frac{2}{n}$.



Originalgraph G Ergebnis MEG zu G

Abbildung 5.9: Worst-case zu `TransRed-Simon`

Kapitel 6

Approximativer Algorithmus von Khuller et al.

Wir wollen nun einen approximativen Algorithmus entwickeln, der aus einer Arbeit von Khuller et al. [20] stammt.
Die Idee dieses Algorithmus ist es, in dem Graphen Kreise größtmöglicher Länge zu finden. Diese bilden zwar i.a. keine starke Zusammenhangskomponente, sind aber stark zusammenhängend. Durch eine Reduktion wird der Kreis im Graphen durch einen neuen Knoten ersetzt. In diesem neuen Graphen wird nun wiederum nach langen Kreisen gesucht.
Dazu wählen wir vor der Ausführung des Algorithmus eine feste Zahl $k \in \mathbb{N}$ und suchen Kreise der Länge größer gleich k. Sind keine mehr vorhanden, so dekrementieren wir k.
Dabei werden für die unterschiedliche Wahl von k auch verschieden gute Approximationen erreicht. Für große k erreichen wir Güten bis zu $\pi^2/6 \approx 1,645$, für $k = 3$ 'nur' solche von 1,75. Dabei sind alle Ansätze in polynomieller Zeit implementierbar. Der Ansatz für $k = 3$ ist fast-linear.
In diesem Kapitel werden zuerst einige Definitionen eingeführt sowie zugehörige Eigenschaften bewiesen. Danach wird der allgemeine Ansatz genauer erläutert und die Güte für den allgemeinen Fall hergeleitet. Der fast-lineare Ansatz für $k = 3$ wird anschließend vorgestellt. Der letzte Abschnitt dient dazu, die gefundenen Lösungen an einigen Beispielen zu verdeutlichen.

6.1 Grundlagen

Wir führen nun die Kontraktion formal ein und leiten wichtige Eigenschaften her, um diese später verwenden zu können.
Der kontrahierte Graph ersetzt die Knoten durch einen neuen und streicht auftretende Schleifen und doppelte Pfeile. Formal ist er durch folgende Definition gegeben:

Definition 6.1. *Sei $G = (V, E)$ ein Graph und $e = (v, w) \in E$ ein Pfeil aus G. Dann ist der reduzierte (kontrahierte) Graph $G/\{e\} := (V', E')$ durch*

- $V' := V \backslash \{v, w\} \cup (v; w)$

- $E' := (E \cap (V' \times V')) \cup \{((v;w),x) \mid x \in V \wedge ((v,x) \in E \vee (w,x) \in E)\}$
  $\cup \{(x,(v;w)) \mid x \in V \wedge ((x,v) \in E \vee (x,w) \in E)\}$

definiert.

Diese Definition wird nun durch das in Abbildung 6.1 dargestellte Beispiel veranschaulicht. Durch die Kontraktion eines Pfeiles (v,w) werden die Knoten v und w zusammengelegt, und dadurch auftretende doppelte Pfeile sowie Schlingen gelöscht.

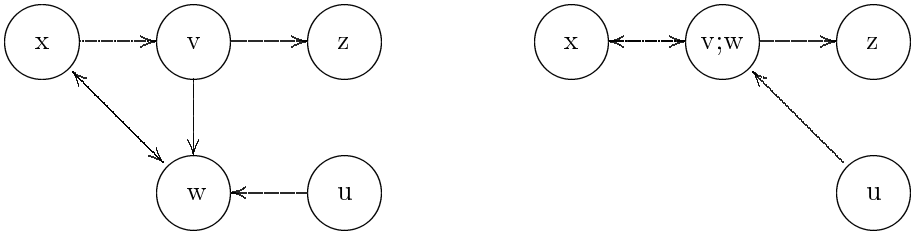


Abbildung 6.1: Beispiel zur Definition 6.1.

Die Kontraktion ähnelt dem Reduktionsgraphen, bei dem die starken Zusammenhangskomponenten zu Knoten reduziert werden. Auch hier kann die Kontraktion durch zwei Abbildungen auf der Knotenmenge und der Pfeilmenge aufgefaßt werden. Analog können auch die Repräsentantenabbildungen definiert werden. Da wir aber keine Schlingen zulassen, müssen wir die zur Kontraktion passende Abbildung auf der Pfeilmenge mit einem neuen Element $\perp$ zu $\phi_2 : E \to E' \cup \{\perp\}$ erweitern, um deutlich zu machen, daß ein Pfeil aus dem Originalgraphen im kontrahierten Graphen nicht abgebildet wird. Es stört nicht, daß Schlingen nicht abgebildet werden, da diese nach Lemma 3.3. in Approximationen nicht benötigt werden.

Um eine Menge von Pfeilen zu kontrahieren, findet eine Kontraktion der Pfeile nacheinander statt. Formal ist dies durch die folgende Definition gegeben:

Definition 6.2. *Seien $G = (V,E)$ ein Graph und $S = \{e_1,\ldots,e_n\} \subseteq E$ eine Menge von Pfeilen aus G. Wir definieren G/S rekursiv durch*

$$G'/S := \begin{cases} G/\{e_1\} & : |S| = 1 \\ (G/\{e_1\})/\{\phi(e_2),\ldots,\phi(e_n)\} & : |S| > 1 \end{cases},$$

wobei ϕ die Abbildung ist, die durch die Kontraktion $G/\{e_1\}$ entsteht. Auftretende Kontraktionen mit dem Element $\perp$ werden durch $G/\{\perp\} := G$ definiert.

Bemerkung 6.1. *Dabei ist es wichtig zu beachten, daß die Reihenfolge in der die Pfeile kontrahiert werden, nicht relevant ist. Formal ergeben sich zwar verschiedene Graphen, die aber isomorph sind. Die Abbildung ϕ ist notwendig, da die Pfeile $e_2,\ldots,e_n$ durch die Kontraktion von e_1 eventuell nicht mehr vorhanden sind.*

Auch die Kontraktionen mehrerer Pfeile können als Abbildungen aufgefaßt werden. Formal sind diese Funktionen durch die Hintereinanderausführung der einzelnen durch die Kontraktionen entstandenen Abbildungen gegeben.

Wir wollen nun einige wichtige Eigenschaften der Kontraktionsoperation erarbeiten.
Seien G ein stark zusammenhängender Graph und K ein Kreis in G. Wir wollen zeigen, daß auch der kontrahierte Graph G/K stark zusammenhängend ist. Ersetzen wir die neu entstandene Komponente wieder zurück, d.h. den neuen Knoten durch die Knoten auf dem Kreis, und die neu entstandenen Pfeile jeweils durch einen Repräsentanten, so ist auch dieser Graph stark zusammenhängend.
Um später nicht formal argumentieren zu müssen, beweisen wir folgendes Lemma mit den dazu eingeführten Abbildungen:

Lemma 6.1. *Seien $G = (V, E)$ ein stark zusammenhängender Graph, $U = \{u_0, \ldots, u_n\}$ und $K = (u_0, \ldots, u_n, u_0)$ ein Kreis in G. Seien $(W, F) := G/K$ der kontrahierte Graph und $\phi_V : V \to W$, $\phi_E : E \to F \cup \{\perp\}$ die zur Kontraktion gehörenden Abbildungen sowie $\psi_W : W \to V$ und $\psi_F : F \to E$ die Repräsentantenabbildungen. Außerdem sei $\hat{U} = \phi_V(u_0)$ der neu hinzugefügte Knoten in G/K.*
Dann sind $G' := (V, (F \cap E) \cup K \cup \psi_F(F \backslash E))$ und G/K stark zusammenhängend.

Beweis: Für den Beweis des starken Zusammenhanges von G/K seien $x, y \in W$. Falls $x = y$ oder $x, y \neq \hat{U}$ gilt, ist ein Weg von x nach y offensichtlich. Sei nun $x = \hat{U} \neq y$. Da G stark zusammenhängend ist, gibt es einen einfachen Weg $P = (u_0, \ldots, y)$ in G. Sei u_i der letzte Knoten aus U im Weg P und (u_i, z) der zugehörige Pfeil. Dann ist $\phi_E(u_i, z) = (\hat{U}, z) \in F$ und damit $\hat{U} \to z \to \cdots \to y$ ein Weg in G/K. Analog folgt die Behauptung für $x \neq \hat{U} = y$.
Für den zweiten Beweis wollen wir zuerst die Pfeilmenge von G' beschreiben. Sie besteht aus Pfeilen des Graphen G/K, die auch in G enthalten sind. Damit sind die beteiligten Knoten nicht in dem Kreis K enthalten. Außerdem sind die Pfeile auf dem Kreis K sowie je ein Repräsentant für jeden neuen Pfeil, der durch die Kontraktion entstanden ist, in G' enthalten.
Seien $x, y \in V$. Wir wollen nun zeigen, daß es in G' einen Weg von x nach y gibt. Falls $x, y \in U$ gilt, ist ein Weg durch den Kreis K gegeben. Falls $x, y \notin U$ gilt, so ist ein Weg durch den starken Zusammenhang von G gegeben.
Sei nun $x \notin U$ und $y = u_j \in U$. Da G/K stark zusammenhängend ist, gibt es einen einfachen Weg $P = (x = x_0, \ldots, x_n = y)$ von x nach y in G/K. Sei $(x_{n-1}, u_i) := \psi_F(x_n - 1, x_n)$ der Repräsentant des Pfeiles (x_{n-1}, x_n). Dann ist $x_0 \to \cdots \to x_{n-1} \to u_i \xrightarrow[K]{*} u_j = y$ ein Weg von x nach y in G'. Der umgekehrte Fall (mit $y \notin U$ und $x \in U$) folgt analog. □

Im folgenden Lemma wird die Beziehung zwischen der optimalen Lösung und den Eigenschaften der enthaltenen Kreise charakterisiert.

Lemma 6.2. *Seien $G = (V, E)$ ein stark zusammenhängender Graph und c die Länge eines längsten Kreises in G. Dann gilt:*

$$\mathcal{OPT}(G) \geq \frac{c}{c-1}(|V| - 1).$$

Beweis: Sei $M = (V, E')$ ein MEG von G und $S = \emptyset$. Wir bemerken zuerst, daß es in einem stark zusammenhängenden Graphen mit mindestens zwei Knoten immer einen Kreis gibt.
Finde einen Kreis $K = (v, \ldots, v)$ in M. Kontrahiere nun K, füge die Pfeile zu S hinzu und erhalte $M := M/K$. Wiederhole diesen Schritt solange, bis M nur noch aus einem Knoten besteht.
Dabei bleibt M nach Lemma 6.1. immer stark zusammenhängend. Wir können die Schritte also wiederholt anwenden, und durch $|V(M)| < |V(M/K)|$ wird abgesichert, daß es nur endlich viele Schritte gibt. Außerdem wird durch die Kontraktion die Länge eines längsten Kreises nicht größer, und in S werden alle Pfeile von M aufgesammelt. Es sind nicht weniger Pfeile, denn sonst würden wir einen stark zusammenhängenden Graphen erhalten, der weniger Pfeile als der MEG enthält.
Wir wollen nun die Anzahl der Pfeile gegen die Anzahl der Knoten abschätzen:
In jedem Schritt werden die Pfeile auf einem Kreis der Länge $b \leq c$ kontrahiert und entsprechend die b Knoten auf dem Kreis zu einem neuen zusammengefaßt. Außerdem werden eventuell noch weitere l Pfeile gelöscht. Der neue Graph M/K besitzt $b-1$ Knoten und $b+l$ Pfeile weniger als M.
Das Verhältnis der Anzahl der kontrahierten Pfeile gegenüber der Abnahme der Knoten in M entspricht also: $\frac{b+l}{b-1} \geq \frac{b}{b-1} \geq \frac{c}{c-1}$. Also ist die Güte in jedem Schritt mindestens $\frac{c}{c-1}$.
Da es höchstens $|V|-1$ solcher Schritte geben kann, ist die Gesamtgüte $\frac{c}{c-1}(|V|-1)$.
Es folgt die Behauptung: $\mathcal{OPT}(G) \geq \frac{c}{c-1}(|V|-1)$. □

Dieses Lemma zeigt einen einfachen Beweis für das Lemma 3.3. (i). Es folgt außerdem, daß in einem stark zusammenhängenden Graphen, der nur Kreise der Länge zwei enthält, ein MEG (mindestens) $2|V|-2$ Pfeile besitzt.

Bemerkung 6.2. *Seien $G = (V, E)$ ein stark zusammenhängender Graph und $S \subseteq E$ eine Menge von Pfeilen. Dann gilt:*

$$\mathcal{OPT}(G) \geq \mathcal{OPT}(G/S).$$

Beweis: Seien $M = (V, R)$ ein MEG von G und $G' := (V', E') := G/S$.
Damit gilt $|R| = \mathcal{OPT}(G)$ und durch die Kontraktion entsteht eine Repräsentantenabbildung $\varphi : E \to E' \cup \{\perp\}$, die jedem Pfeil in G höchstens einen Repräsentanten in G' zuordnet. Dann ist $(V', \varphi(R) \cap V' \times V')$ ein stark zusammenhängender Untergraph von G' mit höchstens $|R|$ Pfeilen.
Es folgt die Behauptung durch $\mathcal{OPT}(G) = |R| \geq |\varphi(R) \cap V' \times V'| \geq |E'| = \mathcal{OPT}(G/S)$. □

Im weiteren Verlauf dieses Kapitels benötigen wir einige mathematische Gleichungen, die hier kurz zusammengestellt sind und sich bei Bronstein [7] finden.

Bemerkung 6.3. *Es gilt:*

(i) $\forall\, k \in \mathbb{N} : \sum_{i=k}^{\infty} \frac{1}{i(i+1)} = \frac{1}{k}$

(ii) $\sum_{i=3}^{\infty} \frac{1}{(i-1)(i-2)} = 1$

(iii) $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$.

6.2 Entwicklung des Algorithmus

Ziel ist es nun, einen approximativen Algorithmus zu entwickeln. Dazu wird zuerst ein allgemeiner Ansatz vorgestellt, der im folgenden Abschnitt in einem Spezialfall in einen programmnahen Pseudocode umgesetzt wird.

6.2.1 Allgemeiner Algorithmus

Wie in der Einleitung beschrieben, besteht die Idee dieses Ansatzes darin, Kreise möglichst großer Länge zu finden, um diese dann zu kontrahieren. Dabei ist es nicht möglich nacheinander die längsten Kreise suchen, da die Bestimmung des längsten Kreises in einem Graphen $\mathcal{NP}$-hart ist [10].
Um einen polynomiellen Algorithmus zu entwickeln, müssen wir uns auf geeignete Weise einschränken und suchen aus diesem Grund nur Kreise einer bestimmten (konstanten) Länge. Dazu wählen wir uns vor der Ausführung ein festes $k \in \mathbb{N}$.
Diese Ideen liefern den in Tabelle 6.1 dargestellten Algorithmus.

```
function CONTRACT-CYCLES_k(G : graph)
  H := G;
  Res := ∅;
  for i = k to 2 by −1 do
    while H enthält Kreis der Länge ≥ i
      Sei L so ein Kreis;
      Res := Res ∪ L;
      H := H/L;
    end;
  end;
  return Res
```

Tabelle 6.1: Pseudocode CONTRACT-CYCLES$_k$

Wir werden später zeigen, daß der Algorithmus in polynomieller Zeit implementiert werden kann. Der Spezialfall $k = 3$ und seine fast-lineare Implementierung sind Thema des Abschnittes 6.2.2.
Wir kümmern uns in diesem Abschnitt nicht um formale Details einer Implementierung, bei der z.B. zu den Pfeilen auf dem Kreis jeweils ein Repräsentant ausgewählt werden muß. Abgesehen

davon ist durch eine wiederholte Anwendung des Lemmas 6.1. klar, daß die aufgesammelten Pfeile einen stark zusammenhängenden Untergraphen der Eingabe bilden.
Im folgenden Lemma betrachten wir die Güte des Algorithmus CONTRACT-CYCLES$_k$.

Theorem 6.1. *Seien $G = (V, E)$ ein stark zusammenhängender Graph und $k \in \mathbb{N}$. Dann gibt* CONTRACT-CYCLES$_k(G)$ *höchstens $c_k \cdot \mathcal{OPT}(G)$ Pfeile zurück, wobei*

$$\frac{\pi^2}{6} \leq c_k \leq \frac{\pi^2}{6} + \frac{1}{(k-1)k}$$

gilt.

Beweis: Wir zählen zunächst die Pfeile ab, die im Ergebnis *Res* sind. Bezeichne dazu mit n_i die Anzahl der Knoten, die nach der Kontraktion der Kreise der Länge i im Graphen vorhanden sind, und mit $n = n_{k+1} = |V|$ die Anzahl der Knoten in G.
Wir betrachten die Kontraktion der Kreise der Länge $i < k$. Seien in diesem Schritt j Kreise gefunden worden. Dann gilt zunächst, daß ji Pfeile zum Ergebnis hinzugefügt worden sind und daß sich die Anzahl der Knoten um $j(i-1)$ verringert hat. Zudem läßt sich diese Anzahl auch durch $n_{i+1} - n_i$ beschreiben. Dann gilt:

$$ji = j(i-1)\frac{i}{i-1} = (n_{i+1} - n_i)\frac{i}{i-1}.$$

Im ersten Schritt, der Kontraktion der Kreise der Länge $\geq k$, schätzen wir die Anzahl dementsprechend mit $\frac{k}{k-1}(n - n_k)$ ab. Für $i \in \{2, \ldots, k-1\}$ benutzen wir im folgenden die Abschätzung: $\frac{i}{i-1}(n_{i+1} - n_i)$.
Nun schätzen wird die Gesamtzahl der Pfeile wie folgt ab:

$$\begin{aligned}
&\frac{k}{k-1}(n-n_k) + \sum_{i=2}^{k-1} \frac{i}{i-1}(n_{i+1}-n_i) \\
&= \frac{k}{k-1}(n-n_k) + \frac{2}{1}(n_3-n_2) + \frac{3}{2}(n_4-n_3) + \cdots + \frac{k-1}{k-2}(n_k - n_{k-1}) \\
&= \frac{k}{k-1}(n-n_k) - 2n_2 + \left(\frac{2}{1} - \frac{3}{2}\right) n_3 + \cdots + \left(\frac{k-2}{k-3} - \frac{k-1}{k-2}\right) n_{k-1} + \frac{k-1}{k-2} n_k \\
&= \frac{kn}{k-1} - \frac{k}{k-1} n_k - 2n_2 + \sum_{i=3}^{k-1} \left(\frac{i-1}{i-2} - \frac{i}{i-1}\right) n_i + \frac{k-1}{k-2} n_k \\
&= \frac{kn}{k-1} - 2n_2 + \sum_{i=3}^{k} \left(\frac{i-1}{i-2} - \frac{i}{i-1}\right) n_i \\
&= \frac{kn}{k-1} - 2n_2 + \sum_{i=3}^{k} \left(\frac{(i-1)(i-1)}{(i-2)(i-1)} - \frac{i(i-2)}{(i-1)(i-2)}\right) n_i \\
&= \frac{kn}{k-1} - 2n_2 + \sum_{i=3}^{k} \left(\frac{n_i}{(i-1)(i-2)}\right)
\end{aligned}$$

$$
\begin{aligned}
&= \frac{kn}{k-1} - 2n_2 + \sum_{i=3}^{k}\left(\frac{n_i}{(i-1)(i-2)}\right) - \underbrace{\sum_{i=3}^{k}\left(\frac{1}{(i-1)(i-2)}\right) + \sum_{i=3}^{k}\left(\frac{1}{(i-1)(i-2)}\right)}_{=0} \\
&= \frac{kn}{k-1} - 2n_2 + \sum_{i=3}^{k}\left(\frac{n_i-1}{(i-1)(i-2)}\right) + \sum_{i=3}^{k}\left(\frac{1}{(i-1)(i-2)}\right) \\
&= \frac{kn}{k-1} + \sum_{i=3}^{k}\left(\frac{n_i-1}{(i-1)(i-2)}\right) - \underbrace{\left(\underbrace{2n_2}_{\geq 2} - \underbrace{\sum_{i=3}^{k}\left(\frac{1}{(i-1)(i-2)}\right)}_{<1}\right)}_{>0} \\
&< \frac{kn}{k-1} + \sum_{i=3}^{k}\left(\frac{n_i-1}{(i-1)(i-2)}\right) \\
&= \left(1+\frac{1}{k-1}\right)n + \sum_{i=3}^{k}\left(\frac{n_i-1}{(i-1)(i-2)}\right).
\end{aligned}
$$

Mit Hilfe der Gleichungen $\mathcal{OPT}(G) \geq n$ (Lemma 3.3.(i)) und $\mathcal{OPT}(G) \geq \frac{i-1}{i-2}(n_i-1)$ für alle $i \in \{3,\ldots,k\}$ (Lemma 6.2.) sowie der Bemerkung 6.2. ergibt sich folgende Abschätzung der Güte:

$$
\begin{aligned}
&\frac{(1+\frac{1}{k-1})n}{\mathcal{OPT}(G)} + \sum_{i=3}^{k}\frac{\frac{n_i-1}{(i-1)(i-2)}}{\mathcal{OPT}(G)} \\
\leq\ &\frac{(1+\frac{1}{k-1})n}{n} + \sum_{i=3}^{k}\frac{\frac{n_i-1}{(i-1)(i-2)}}{\frac{i-1}{i-2}(n_i-1)} \\
=\ &\frac{1}{k-1} + 1 + \sum_{i=3}^{k}\frac{(n_i-1)(i-2)}{(i-1)(i-2)(i-1)(n_i-1)} \\
=\ &\frac{1}{k-1} + 1 + \sum_{i=3}^{k}\frac{1}{(i-1)(i-1)} \\
=\ &\frac{1}{k-1} + 1 + \sum_{i=2}^{k-1}\frac{1}{i^2} \\
=\ &\frac{1}{k-1} + \sum_{i=1}^{k-1}\frac{1}{i^2}.
\end{aligned}
$$

Die Güte können wir also durch $c_k := \frac{1}{k-1} + \sum_{i=1}^{k-1}\frac{1}{i^2}$ abschätzen. Nun wollen wir die Ungleichungen $\frac{\pi^2}{6} \leq c_k \leq \frac{\pi^2}{6} + \frac{1}{(k-1)k}$ zeigen.

Wir beweisen zuerst die zweite Ungleichung:

$$\begin{aligned} c_k &= \frac{1}{k-1} + \sum_{i=1}^{k-1} \frac{1}{i^2} \\ &= \frac{\pi^2}{6} + \frac{1}{k-1} - \sum_{i=k}^{\infty} \frac{1}{i^2} \\ &\leq \frac{\pi^2}{6} + \frac{1}{k-1} - \sum_{i=k}^{\infty} \frac{1}{i(i+1)} \\ &= \frac{\pi^2}{6} + \frac{1}{k-1} - \frac{1}{k} \\ &= \frac{\pi^2}{6} + \frac{1}{(k-1)k}. \end{aligned}$$

Um zu zeigen, daß $\frac{\pi^2}{6} \leq c_k$ gilt, reicht es $\frac{1}{k-1} \geq \sum_{i=k}^{\infty} \frac{1}{i^2}$ zu zeigen.

Definiere dazu $f : \mathbb{R}^+ \to \mathbb{R}^+, x \mapsto \frac{1}{x^2}$. Offensichtlich ist f monoton fallend. Seien $k, n \in \mathbb{N}$ mit $k \leq n$. Dann gilt:

$$\sum_{i=k}^{n} \frac{1}{i^2} \leq \int_{k-1}^{n} \frac{1}{x^2} dx = \left[-\frac{1}{x} \right]_{k-1}^{n} = -\frac{1}{n} + \frac{1}{k-1}.$$

Mit dem Grenzübergang für $n \to \infty$ folgt $\sum_{i=k}^{\infty} \frac{1}{i^2} \leq \frac{1}{k-1}$. □

Es lassen sich also für die verschiedene Wahl von k unterschiedliche Approximationsgüten erreichen. Diese sind in Tabelle 6.3 beispielhaft dargestellt. Beachte dabei, daß für kleine k die Güten ohne die Abschätzung aus Theorem 6.1. exakt berechnet werden können.
Den theoretischen Werten c_k stehen hier praktisch erreichbare b_k's gegenüber. Es gibt also Graphen für die????? der Algorithmus `CONTRACT-CYCLES`$_k$ 'nur' Approximationen mit der Güte b_k liefert. Für $k = 3$ wird so ein Graph später im Abschnitt 6.3 präsentiert.
Die Unterschiede zwischen c_k und b_k bei großen k's machen deutlich, daß die Abschätzung der Güte noch besser sein könnte.

k	3	4	5	∞
c_k	1,75	1,694	1,674	1,645
b_k	1,75	1,666	1,625	1,5

Tabelle 6.2: Güten des Approximationsalgorithmus

Wir wollen nun zeigen, daß `CONTRACT-CYCLES`$_k$ für beliebiges k in polynomieller Zeit implementiert werden kann.

Dazu sei $k \in \mathbb{N}$ und $G = (V, E)$ ein stark zusammenhängender Graph, sowie $|V| = n$ und $|E| = m$. Um einen Kreis mit mindestens der Länge j zu finden, prüfen wir für jeden einfachen Weg $(v_1, \ldots, v_j)$ der Länge $j-1$ in G, ob es einen Weg von v_j nach v_1 gibt. Falls j gerade ist, gibt es höchstens $m^{j/2}$ solcher einfachen Wege, sonst höchstens $nm^{(j-1)/2}$. Der Test, ob es einen Weg gibt, benötigt $\mathcal{O}(m)$ Zeit. Also kann der Test für die Kreise der Länge j in $\mathcal{O}(m^{j/2}m) = \mathcal{O}(m^{1+j/2})$ bzw. $\mathcal{O}(nm^{(j-1)/2}m) = \mathcal{O}(nm^{(j+1)/2})$ erfolgen. Die innere **while**-Schleife wird insgesamt höchstens n-mal durchlaufen, die äußere **for**-Schleife k-mal. Es ergibt sich eine Gesamtlaufzeit für gerade k von $\mathcal{O}(n \sum_{i=1}^{k} m^{1+i/2}) = \mathcal{O}(nkm^{(1+k/2)}) = \mathcal{O}(nm^{1+k/2})$, für ungerade von $\mathcal{O}(n \sum_{i=1}^{k} nm^{(i+1)/2}) = \mathcal{O}(nknm^{(k+1)/2}) = \mathcal{O}(n^2m^{(k+1)/2})$.

Also kann `CONTRACT-CYCLES`$_k$ in polynomieller Zeit implementiert werden.

6.2.2 Ein fast-linearer Approximationsalgorithmus

Wir wollen nun den in Tabelle 6.1. dargestellten Algorithmus `CONTRACT-CYCLES`$_k$ für $k = 3$ implementieren. Dabei reduziert sich `CONTRACT-CYCLES`$_3$ auf zwei Teile: In der ersten Phase werden „lange“ Kreise der Länge ≥ 3 gesucht und kontrahiert, in der zweiten die „kurzen“ der Länge zwei.
Es sind vor allem das in 6.2.1 angesprochene Finden von Repräsentanten sowie das effiziente Suchen der Kreise zu implementieren.
Die erste Phase wollen wir mit Hilfe eines modifizierten DFS-Durchlaufes `DFS-CONTRACT` entwickeln, die zweite mit Hilfe der später vorgestellten Prozedur `ADD-2-CYCLES`.
Dabei speichern wir in der Variablen S das Ergebnis und erhalten den in Tabelle 6.3 dargestellten grundsätzlichen Aufbau.

```
function CONTRACT-CYCLES_3(G = (V, E) : graph)
  S = ∅;
  choose root ∈ V;
  DFS-CONTRACT(root);
  ADD-2-CYCLES();
  return (V, S)
```

Tabelle 6.3: Pseudocode `CONTRACT-CYCLES`$_3$

`DFS-CONTRACT`

Zunächst wollen wir nun den Tiefensuchdurchlauf `dfs` so modifizieren, daß wir alle langen Kreise finden. Bezeichne dazu im folgenden mit G' den aktuell bearbeiteten Graphen, d.h. die Knoten und Pfeile, die schon besucht wurden. Aus G' und S wird mit G'/S der aktuelle kontrahierte Graph bezeichnet, d.h. der abgearbeitete Graph in dem schon lange Kreise

kontrahiert wurden. Wenn wir einen Pfeil besuchen und er einen langen Kreis in G'/S bildet, sollen die Pfeile in S übernommen werden und die Knoten auf dem Kreis zu einem neuen kontrahiert werden. So können wir formal durch die Invariante aus Lemma 6.3. zusichern, daß der Graph G'/S nie lange Kreise enthält.
Für einen Knoten w bezeichnen wir mit W seinen Repräsentantenknoten in G'/S. Dieser wird durch eine union-find-Struktur [23] dargestellt. Die Initialisierung `MAKESET`(v) ordnet jedem Knoten sich selbst als Repräsentanten zu. Falls mit `UNION`(u, v) zwei Komponenten zu einer zusammengefaßt werden, kann danach mit Hilfe von `FIND`(v) der (neue) Repräsentant gefunden werden. Um uns von der konkreten Implementierung zu lösen, bezeichnen wir für einen Knoten $x \in V$ seinen Repräsentanten in G'/S mit X und die Menge der Knoten mit diesem Vertreterknoten mit $\hat{X}$.
Die Invariante sichert außerdem, daß der Graph G'/S nur aus einem Baum und eventuell dazugehörigen Rückwärtspfeilen besteht (siehe Abbildung 6.4). Daher gibt es immer einen eindeutigen Weg von der Wurzel zu jedem Knoten und höchstens einen von jedem Knoten zur Wurzel in G'/S. Diese eindeutigen Wege werden durch die Felder $from{-}root$ bzw. $to{-}root$ dargestellt.
Die Knoten, für die der DFS-Aufruf noch nicht beendet ist, bilden einen Weg in G'/S, den aktiven Weg. Er wird mit Hilfe des Feldes $to{-}active$ dargestellt.
Fassen wir die benötigten Felder noch einmal mit ihren formalen Eigenschaften zusammen, wobei wir mit `nil` und `current` zwei ausgezeichnete Konstanten bezeichnen:

- $from{-}root(W)$: $V(G'/S) \to E \cup \{\texttt{nil}\}$

  gibt für jeden Baumpfeil (U, W) in G'/S einen Pfeil $(u, w) \in (\hat{U} \times \hat{W}) \cap E$ zurück,

  sonst `nil`.

- $to{-}root(U)$: $V(G'/S) \to E \cup \{\texttt{nil}\}$

  gibt für jeden Rückwärtspfeil (U, W) in G'/S einen Pfeil $(u, w) \in (\hat{U} \times \hat{W}) \cap E$ zurück,

  sonst `nil`.

- $to{-}active(U)$: $V(G'/S) \to E \cup \{\texttt{current}, \texttt{nil}\}$

  Für jeden Knoten u auf dem aktiven Weg gilt: $to{-}active[U] = (u, w) \in (\hat{U} \times \hat{W}) \cap E$, wobei W Kind von U ist, und die Tiefensuche von w rekursiv durch u aufgerufen wird.

  Für den aktuellen Knoten u gilt: $to{-}active[U] = \texttt{current}$.

  Für alle anderen Knoten u gilt: $to{-}active[U] = \texttt{nil}$.

Die Felder sichern zu, daß wir im folgenden im Graphen G'/S stets Repräsentanten für jeden Pfeil finden können.
Für eine anschauliche Bedeutung der Felder sei auf den Abschnitt 6.3 verwiesen.

Wir wollen nun die verschiedenen Fälle in einem Tiefensuchdurchlauf bearbeiten. Dabei sei u der aktuelle Knoten.

Der erste Fall behandelt das Besuchen eines weißen Knotens w. Also ist (u, w) ein Baumpfeil. Es wird kein Kreis geschlossen, w wird der neue aktuelle Knoten, und der aktive Weg und der eindeutige Weg von der Wurzel aus werden um den Pfeil (u, w) verlängert.

```
MAKE-SET(w);
to−active[FIND(u)] := (u, w);
from−root[FIND(w)] := (u, w);
```

Im zweiten Fall, dem Abarbeiten eines besuchten Knotens w, wird nun ein Kreis gefunden. Zuerst wollen wir die Kreise der Länge 1, die Schlingen, ausschließen, die nur dann vorliegen, wenn die Knoten u und w denselben Vertreterknoten haben.

```
if FIND(u) ≠ FIND(w)
```

Außerdem wollen wir ausschließen, daß der Kreis die Länge zwei besitzt. Dies ist genau dann der Fall, wenn in G'/S W der Vater von U oder W der Sohn von U ist.

$(x, y) = from{-}root[U]$ sei der eindeutige Pfeil von der Wurzel aus, d.h. X ist der Vater von $Y = U$. Falls nun x und w dieselben Vertreterknoten besitzen, hat der Kreis $U = Y \to X = W \to U$ die Länge zwei. (u, w) ist daher ein Rückwärtspfeil, das Feld $to{-}root$ kann an der Stelle U zu (x, y) aktualisiert werden.

```
(x, y) := from−root[FIND(u)];
if FIND(x) = FIND(w)
   then to−root[FIND(u)] := (u, w);
end;
```

Um den anderen Fall abzudecken, betrachten wir $(x, y) = to{-}root[W]$. X ist der Vaterknoten von W in G'/S. Analog wie oben folgt im Fall $X = U$, daß der Weg $X = U \to Y = W \to U$ die Länge zwei besitzt. Da der Vorwärtspfeil (u, w) nicht für die Felder benötigt wird, findet keine Aktualisierung statt.

```
(x, y) := from−root[FIND(w)];
if FIND(x) = FIND(u)
   then /* nothing to do */
end;
```

In allen anderen Fällen wird ein langer Kreis geschlossen. Mit Hilfe der noch zu entwickelnden Prozedur `CONTRACT-CYCLE` werden die Knoten auf dem Kreis kontrahiert und die Pfeile zu S hinzugefügt.

```
CONTRACT-CYCLE(w);
S := S ∪ {(u, w)};
```

Das Feld *to–active* wird außerdem vor und nach jedem rekursiven Aufruf aktualisiert. Im Detail heißt dies, daß beim Aufruf zuerst $to{-}active[U]$ auf `current` gesetzt wird. Dies geschieht auch, wenn ein rekursiver Aufruf von `DFS-CONTRACT` beendet wird. Falls alle Nachfolger abgearbeitet sind, wird der aktive Weg durch $to{-}active[U] :=$ `nil` verkürzt.
Diese Ideen führen zum Algorithmus `DFS-CONTRACT`, der in Tabelle 6.4 dargestellt wird.

```
procedure DFS-CONTRACT(u : V)
  to−active[FIND(u)] := current;
  forall w ∈ V with (u, w) ∈ E do
    if w is not yet visited
      then MAKE-SET(w);
           to−active[FIND(u)] := (u, w);
           from−root[FIND(w)] := (u, w);
           DFS-CONTRACT(w);
           to−active[FIND(u)] := current;
      else if FIND(u) ≠ FIND(w)
             then (x, y) := from−root[FIND(u)];
                  if FIND(x) = FIND(w)
                    then to−root[FIND(u)] := (u, w);
                    else (x, y) := from−root[FIND(w)];
                         if FIND(x) ≠ FIND(u)
                           then CONTRACT-CYCLE(w);
                                S := S ∪ {(u, w)};
                         end;
                  end;
           end;
    end;
  end;
  to−active[FIND(u)] := nil;
```

Tabelle 6.4: Pseudocode `DFS-CONTRACT`

`CONTRACT-CYCLE`

Nun wollen wir die Prozedur `CONTRACT-CYCLE` entwickeln, die zu dem besuchten Knoten w alle Pfeile auf dem Weg nach u aufsammelt und alle Knoten zu einer neuen Komponente in G'/S verschmilzt.
Der Weg zu u ist dabei eindeutig durch die Felder *to–root* und *to–active* bestimmt. Er besteht aus einem Weg aus Rückwärtspfeilen zum nächsten gemeinsamen Vorfahren von u und w und aus dem aktiven Weg von diesem zum Knoten w. Dabei ist der nächste gemeinsame Vorfahre der erste Knoten auf diesem Weg, der auf dem aktiven Weg liegt.

Wir wollen dabei jeweils den aufgesammelten Knoten in die Komponente W aufnehmen. Deshalb wird mit w immer der aktuelle Knoten bezeichnet.
Wir sammeln solange Pfeile auf, bis der Knoten w in der Komponente des aktuellen Knotens u liegt.

while $to{-}active[\texttt{FIND}(w)] \neq \texttt{current}$ **do**

Nun prüfen wir, ob der aktuelle Knoten im ersten oder im zweiten Teil des Weges ist, d.h. ob wir uns schon auf dem aktiven Weg befinden oder nicht.

if $to{-}active[\texttt{FIND}(w)] = \texttt{nil}$

Im ersten Fall ist der aktive Weg noch nicht erreicht und der eindeutige Weg durch einen Rückwärtspfeil gegeben, den wir zu S hinzufügen. Die Rückwärtspfeile haben wir in $to{-}root$ gespeichert.
Wir werden im Lemma 6.3. formal zeigen, daß so ein Rückwärtspfeil immer existiert, d.h. daß $to{-}root$ an der Stelle W nicht `nil` ist. Der Grund dafür ist, daß dieser Teil des Graphen vom Tiefensuchdurchlauf schon vollständig besucht wurde. Aus dem starken Zusammenhang des Originalgraphen folgt, daß mindestens ein Pfeil von diesem Knoten W wegführt.

$(c, p) := to{-}root[\texttt{FIND}(w)];$
$S := S \cup \{(c, p)\};$

Im zweiten Fall ist der Pfeil durch den aktiven Weg selbst gegeben. Wir fragen das Feld $to{-}active$ an der Stelle W ab und fügen den gefundenen Pfeil hinzu. Dieser existiert, weil sich W auf dem aktiven Weg befindet.

$(p, c) := to{-}active[\texttt{FIND}(w)];$
$S := S \cup \{(p, c)\};$

Beachte dabei, daß der Knoten c in beiden Fällen in derselben Komponente liegt wie w. Es brauchen jetzt nur noch die drei Felder des Knotens p abgefragt zu werden. Die Knoten p und c werden vereinigt und die Daten werden in dem Knoten w aktualisiert. Dabei muß der aktive Weg dann aktualisiert werden, wenn sich w auf diesem befand. Deshalb nehmen wir hier eine Aktualisierung der beiden anderen Felder vor.

$f := from{-}root[\texttt{FIND}(p)];$
$t := to{-}root[\texttt{FIND}(p)];$
$\texttt{UNION}(p, c);$
$from{-}root[\texttt{FIND}(w)] := f;$
$to{-}root[\texttt{FIND}(w)] := t;$

Fassen wir die entwickelten Schritte zusammen und berücksichtigen die Aktualisierung des aktiven Weges, ergibt sich der in Tabelle 6.5 dargestellte Pseudocode `CONTRACT-CYCLE`.

```
procedure CONTRACT-CYLCE(w : V)
  while to−active[FIND(w)] ≠ current do
    if to−active[FIND(w)] = nil
      then (c, p) := to−root[FIND(w)];
           a := to−active[FIND(p)];
           S := S ∪ {(c, p)};
      else (p, c) := to−active[FIND(w)];
           a := to−active[FIND(w)];
           S := S ∪ {(p, c)};
    end;
    f := from−root[FIND(p)];
    t := to−root[FIND(p)];
    UNION(p, c);
    to−active[FIND(w)] := a;
    from−root[FIND(w)] := f;
    to−root[FIND(w)] := t;
  end;
```

Tabelle 6.5: Pseudocode CONTRACT-CYCLE

Invariantenbeweis

Durch die Herleitung ist klar, daß nur lange Kreise kontrahiert und in das Ergebnis S aufgenommen werden.
Es bleibt noch formal zu zeigen, daß der Graph G'/S nie einen langen Kreis enthält und für die Knoten, die nicht auf dem aktiven Weg liegen, immer Rückwärtspfeile vorhanden sind.

Lemma 6.3. *Nach dem Einfügen eines Pfeiles zu G' und der möglichen Kontraktion eines Kreises durch das Hinzufügen zu S gilt:*

(a) *Der Graph G'/S besteht aus Baumpfeilen und eventuell zugehörigen Rückwärtspfeilen.*

(b) *Die einzigen Rückwärtspfeile, die möglicherweise nicht vorhanden sind, befinden sich auf dem aktiven Weg vom Vertreterknoten mit dem Wurzelknoten root zum Vertreterknoten, der den aktuellen Knoten enthält.*

Beweis: Die Invarianten (a) und (b) sind zu Beginn korrekt, da der Graph G' noch keinen Pfeil und nur einen Knoten enthält. Sie werden also etabliert. Der allgemeine Aufbau der Tiefensuche sichert zu, daß der Graph G'/S immer einen Baum enthält.
Die Aufrechterhaltung von (a) und (b) zeigen wir in folgenden Fällen:

(i) Besuch eines Pfeiles (u, w), wobei w noch unbesucht ist: Der Knoten w und der Pfeil (u, w) werden zu G' hinzugefügt. Der Knoten w wird der aktuelle Knoten. In G'/S wird der Zweig durch den Pfeil (U, W) erweitert. Da kein anderer Pfeil oder Knoten

hinzugefügt wird, entsteht kein Kreis. Also ist Teil (a) der Invariante korrekt. Der aktive Weg enthält den ursprünglichen aktiven Weg, also trifft auch Teil (b) zu.

(ii) Besuch eines Pfeiles (u, w), wobei w schon besucht ist: Der Pfeil (u, w) wird zu G' hinzugefügt. Falls $U = W$ oder $(U, W) \in G'/S$, wird kein neuer Kreis erzeugt. Dann bleibt G'/S unverändert, also sind (a) und (b) korrekt. Sonst wird (u, w) zu G' hinzugefügt, und es entsteht ein Kreis aus folgenden Pfeilen: $P = w \xrightarrow{*} lca(u, w) \xrightarrow{*} u \longrightarrow w$. Also wird (U, W) zu G'/S hinzugefügt. Besitzt der Kreis eine Länge ≥ 3 wird der Kreis kontrahiert, und Teil (a) ist korrekt. Der aktive Weg wird höchstens durch die Kontraktion verkleinert, also ist Teil (b) korrekt.

(iii) Alle Pfeile von u aus sind abgearbeitet: Da kein Pfeil und kein Knoten hinzugefügt wird, bleibt Teil (a) erhalten. Sei nun w der neue aktuelle Knoten, d.h. w ist der Vater von u im DFS-Baum. Falls $W = U$ gilt, bleibt Teil (b) erhalten, da der aktive Weg unverändert ist. Sonst sei D die Menge der Nachfolger von u im DFS-Baum. Da G stark zusammenhängend ist, gibt es einen Pfeil (x, y) mit $x \in D$ und $y \in \hat{U} \backslash D$. Da alle Knoten in D abgearbeitet sind, gilt $(x, y) \in G'/S$. Nach Teil (a) ist $x \in U$ und $y \in W$, sonst bildet $X \to Y \to W \to U \to X$ einen langen Kreis in G'/S. Der aktive Weg wird verkürzt, aber in dem verkürzten Teil gibt es einen Rückwärtspfeil, nämlich $(x, y) \in (\hat{U}, \hat{W})$. □

`ADD-2-CYCLES`

Im folgenden wollen wir die kurzen Kreise zum Ergebnis S hinzufügen. Dies soll mit Hilfe der Prozedur `ADD-2-CYCLES` geschehen.
Nach Lemma 6.3. besteht der Graph G'/S nach der Kontraktion aller langen Kreise aus einer Baumstruktur und allen zugehörigen Rückwärtspfeilen. Dabei bilden ein Baumpfeil und der zugehörige Rückwärtspfeil einen Kreis der Länge zwei. Nun können wir die Felder *to–root* und *from–root* nochmals verwenden. Sie geben für jeden Vertreterknoten gerade ein Paar zusammengehöriger Pfeile an. Beachte dabei, daß die Felder für den Vertreterknoten mit dem Element *root* nicht definiert sind. Beachte auch, daß Pfeile eventuell mehrfach eingefügt werden, weil wir ja nur für alle Vertreterknoten Pfeile einfügen wollen. Dies ändert aber nichts am Ergebnis.
Fassen wir diese Ideen zusammen, so ergibt sich der Algorithmus `ADD-2-CYCLES` aus Tabelle 6.6.

6.2.3 Theoretische Eigenschaften

Wir wollen nun die theoretischen Eigenschaften des Algorithmus $\texttt{CONTRACT-CYCLES}_3$ zusammenfassen.

Lemma 6.4. *Seien $G = (V, E)$ ein stark zusammenhängender Graph, $|V| = n$ und $|E| = m$. Dann berechnet $\texttt{CONTRACT-CYCLES}_3(G)$ eine Approximation eines* MEG *von G mit Güte 1,75 in Zeit $\mathcal{O}(m\alpha(m, n))$, wobei $\alpha(m, n)$ die Inverse der Ackermann-Funktion ist.*

```
procedure ADD-2-CYCLES()
  forall v ∈ V
    if FIND[root] ≠ FIND[v]
      then (x, y) := to−root[FIND(v)];
           S := S ∪ {(x, y)};
           (x, y) := from−root[FIND(v)];
           S := S ∪ {(x, y)};
    end;
  end;
```

Tabelle 6.6: Pseudocode ADD-2-CYCLES

Beweis: Die Korrektheit wurde schon in der Herleitung gezeigt, die Güte folgt sofort aus dem Theorem 6.1.. Die Laufzeit ergibt sich aus der Laufzeit eines DFS-Durchlaufes, sowie der Tatsache, daß Zugriffe auf eine union-find-Struktur die Laufzeit $\alpha(m, n)$ benötigen [32]. □

Bemerkung 6.4. *Das Ergebnis von* CONTRACT-CYCLES$_3$ *ist i.a. keine transitive Reduktion.*

6.3 Beispiele

Wir betrachten in diesem Abschnitt ein erläuterndes Beispiel zum entwickelten Algorithmus CONTRACT-CYCLES$_3$, ein worst-case Beispiel und ein größeres Beispiel zur Illustration.

Beispieldurchlauf

Wir betrachten den Beispielgraphen in Abbildung 6.2. Die Knoten sind mit kleinen Buchstaben gekennzeichnet, um später die Repräsentanten als große Buchstaben kenntlich zu machen.

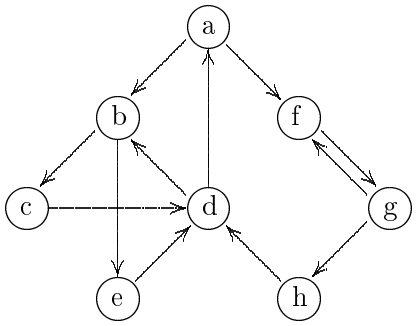
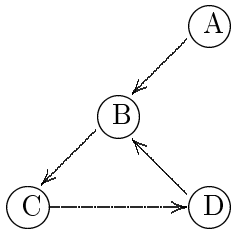


Abbildung 6.2: Originalgraph G

Wir wollen in einem beispielhaften Durchlauf nun den Algorithmus CONTRACT-CYCLES$_3$ nachvollziehen. Im ersten Schritt wird der modifizierte DFS-Durchlauf DFS-CONTRACT ausgeführt.

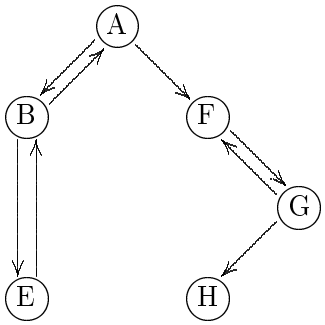
Dazu sei a der Startknoten der Tiefensuche und die Abarbeitungsreihenfolge der Pfeile durch (a,b), (b,c), (c,d), (d,b), (d,a), (b,e), (e,d), (a,f), (f,g), (g,f), (g,h), (h,d) festgelegt.
Nach dem Besuch des Pfeiles (d,b) wird der Kreis $B \to C \to D \to B$ gefunden und zu B kontrahiert.
Die Situation vor der Kontraktion sowie die Inhalte der drei Felder werden in Abbildung 6.3 dargestellt:



	A	B	C	D
$from-root$	`nil`	(a,b)	(b,c)	(c,d)
$to-root$	`nil`	`nil`	`nil`	`nil`
$to-active$	(a,b)	(b,c)	(c,d)	`nil`

Abbildung 6.3: Besuch von (d,b)

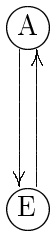
Die Abbildung 6.4 zeigt die Situation vor dem Besuch des letzten Pfeiles. Außerdem werden die Felder $to-active$, $to-root$ und $from-root$ dargestellt. Wir erkennen die Zuweisung der Pfeile des kontrahierten Graphen zu einem aus dem Originalgraphen.
Wir wollen nun die Wirkungsweise der Prozedur `CONTRACT-CYCLE` verdeutlichen. Sei dazu H der aktuelle Knoten und (h,d) der nächste zu besuchende Pfeil. Es wird ein langer Kreis geschlossen, denn es gilt: Die Vertreterknoten von $h(H)$ und $d(B)$ sind verschieden, und B und H stehen nicht in einer Vater-Kind-Beziehung.
Da E nicht auf dem aktiven Weg liegt, werden zuerst (e,d) und danach (d,a) zu S hinzugefügt. Da der Knoten A auf dem aktiven Weg liegt, werden nun Pfeile vom aktiven Weg aufgesammelt. Also kommen $(a,f),(f,g)$ und (g,h) hinzu. Die Knoten E,B,A,F,G,H werden zu einer Komponente A zusammengefaßt. Außerdem ist die baumähnliche Struktur des reduzierten Graphen in Abbildung 6.4 erkennbar.



	A	B	E	F	G	H
$from-root$	`nil`	(a,b)	(b,e)	(a,f)	(f,g)	(g,h)
$to-root$	`nil`	(d,a)	(e,d)	`nil`	(g,f)	`nil`
$to-active$	(a,f)	`nil`	`nil`	(f,g)	(g,h)	`current`

Abbildung 6.4: Vor dem Besuch von (h,b)

Der kontrahierte Graph nach dem DFS-Durchlauf ist in Abbildung 6.5 dargestellt.



	A	E
from−root	`nil`	(b,e)
to−root	`nil`	(e,d)
to−active	`nil`	`nil`

Abbildung 6.5: Nach dem Besuch von (h, b)

Es sind noch die verbleibenden Kreise der Länge zwei zum Ergebnis hinzuzufügen. Die Repräsentanten in den Felder *to−root* und *from−root*, die Pfeile (b, e) und (e, d), werden eingefügt. Man erhält das Ergebnis in Abbildung 6.6.

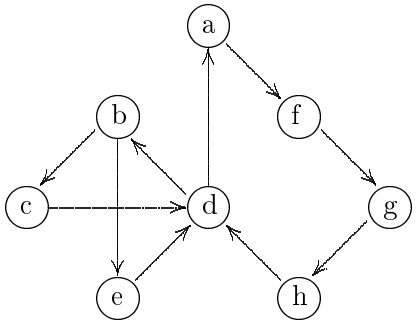
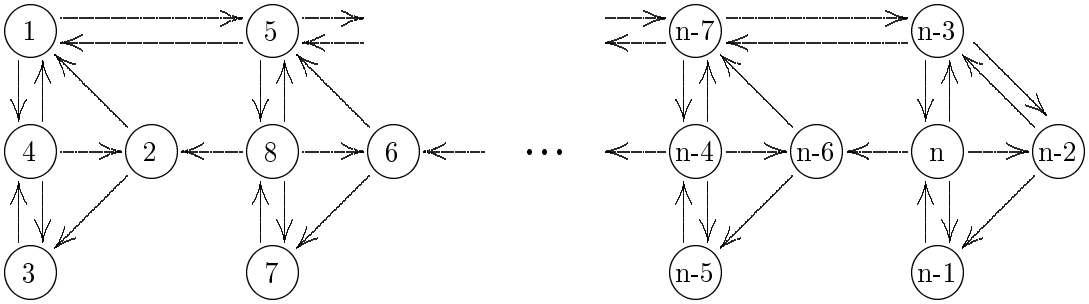


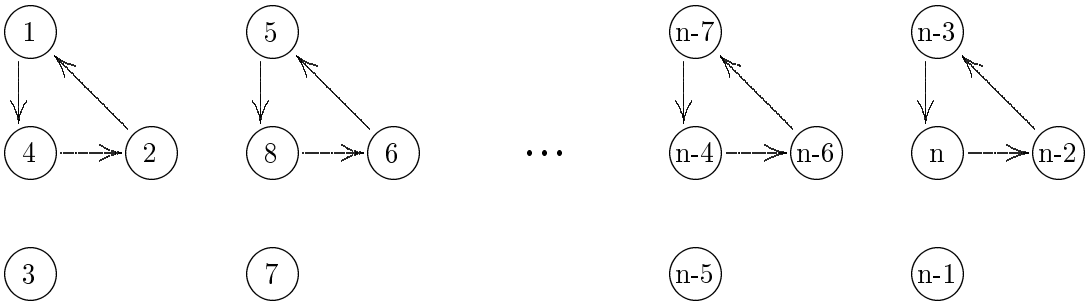
Abbildung 6.6: Ergebnis

Worst-case Beispiel

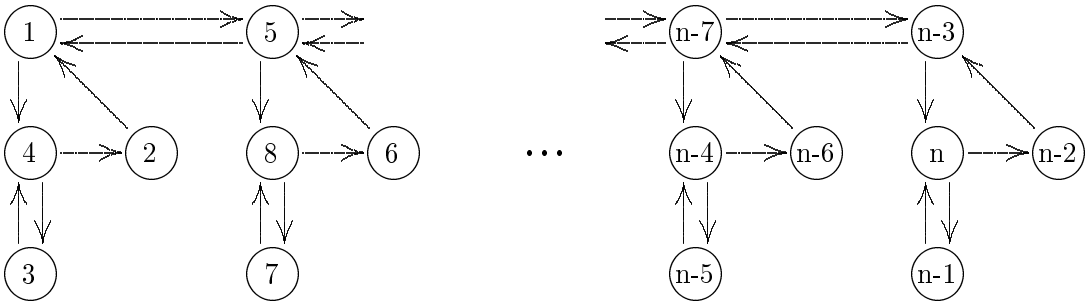
Wir wollen nun ein Beispiel für `CONTRACT-CYCLES`$_3$ angeben, in dem die in Tabelle 6.2 angegebene Güte erreicht wird. Die Güte, dort mit $c_3 = b_3$ bezeichnet, beträgt 1,75, d.h. ein Ergebnis des entwickelten Algorithmus besitzt 1,75mal so viele Pfeile wie ein MEG.
Für dieses allgemeine Beispiel sei $n \in 4\mathbb{N}$. Wir betrachten die Abbildung 6.7. Der Graph im Bild 1 besteht aus $\frac{n}{4}$ Gruppen mit jeweils 4 Knoten. Im ersten Schritt werden langen Kreise gesucht, dies können z.B. die Kreise im Bild 2 sein. Es werden nicht die längsten Kreise, sondern nur Kreise mit mindestens der Länge 3 gefunden. Der restliche kontrahierte Graph besteht nur noch aus 2-er Kreisen. Diese werden im folgenden Schritt hinzugefügt. Es ergibt sich das Ergebnis in Bild 3. Das letzte Bild zeigt einen MEG, der aus einem Hamiltonkreis besteht. Das Resultat enthält $3\frac{n}{4} + 2\frac{n}{4} + 2(\frac{n}{4} - 1) = \frac{7n}{4} - 2$ Pfeile. Die Güte beträgt also $\frac{7}{4} - \frac{2}{n}$. Mit dem Grenzübergang für $n \to \infty$ ergibt sich die Güte $\frac{7}{4} = 1,75$.
Dieses Beispiel ist auch für $k \in \mathbb{N}$ verallgemeinbar, dies wollen wir hier aber nicht darstellen. Damit können die in Tabelle 6.2 dargestellten b_k erreicht werden.



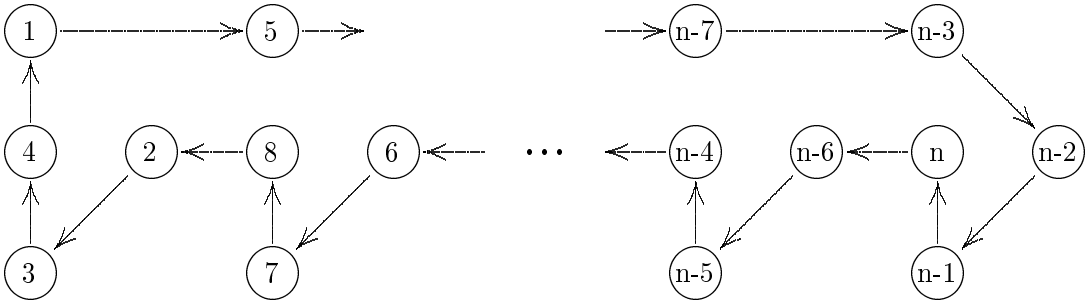
Originalgraph G



Kreise der Länge drei



Ergebnis



MEG

Abbildung 6.7: Worst case zu `CONTRACT-CYCLES`$_3$

Typisches Beispiel

Nachfolgendes Beispiel zeigt ein typisches Ergebnis des Ansatzes, ohne dabei aber $k \in \mathbb{N}$ oder eine konkrete Implementierung anzugeben. Aus Abbildung 6.8 ist nochmals die Idee des Algorithmus ersichtlich. Die Struktur des Graphen zeigt deutlich die im Laufe des Algorithmus gefundenen Kreise.

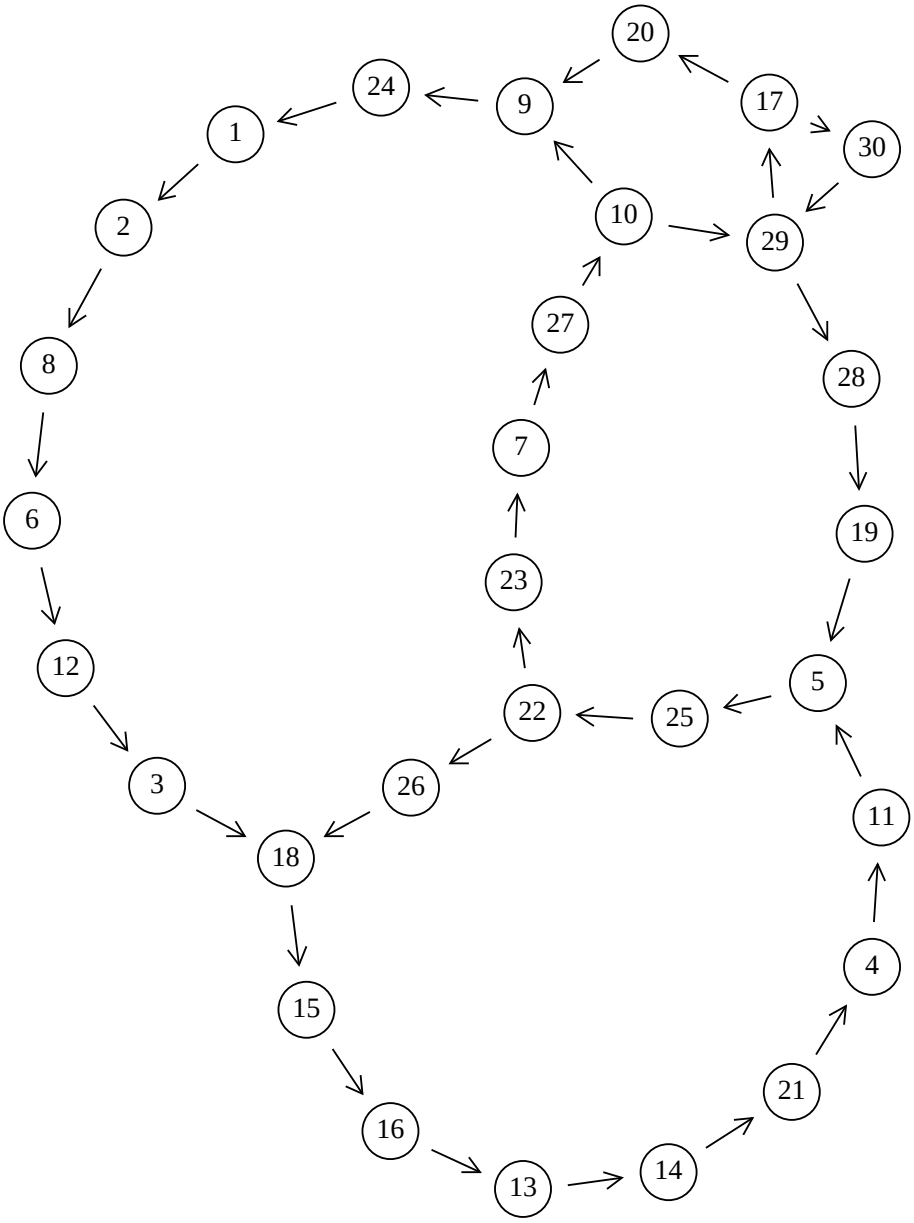


Abbildung 6.8: Typisches Ergebnis des Ansatzes von Khuller

Kapitel 7

Praktischer Vergleich der Algorithmen

Die Entwicklungen haben gezeigt, daß der approximative Ansatz theoretisch sowohl der schnellste ist als auch die besten Güten liefert. Nun wollen wir die in den Kapiteln 4 bis 6 entwickelten Algorithmen praktischen Tests unterziehen.
Dabei konzentrieren wir uns auf die Laufzeit der verschiedenen Ansätze und die Anzahl der Pfeile, die die Algorithmen als Ergebnis liefern.
Wir können in den praktischen Betrachtungen nur von der Anzahl der Pfeile aber nicht von der Güte sprechen, da eine Berechnung eines MEG bzw. von $\mathcal{OPT}(G)$ nicht effizient möglich und daher auch die Güte nicht praktisch berechenbar ist.

Wir betrachten folgende grundsätzliche Fragen:

- Welches der Verfahren hat die beste Laufzeit?
- Welches der Ansätze liefert die wenigsten Pfeile?
- Lassen sich die theoretischen Eigenschaften in den praktischen Ergebnissen wiedererkennen?
- Gelten die wichtigsten Eigenschaften eines Algorithmus unter allen Testbedingungen?
- Gibt es einen „optimalen“ Algorithmus?

Neben diesen offenen Fragen erwarten wir grundsätzlich die folgenden Eigenschaften:

- Je größer die Knotenzahl oder die Dichte des eingegebenen Graphen ist, desto länger ist die Laufzeit.
- Da mit steigender Dichte $\mathcal{OPT}(G)$ im Mittel kleiner wird, kann auch die Anzahl der zurückgegebenen Pfeile kleiner sein.

Zunächst vereinbaren wir eine Testumgebung und die wichtigsten dazugehörigen Parameter.
In den Abschnitten 1 bis 3 werden Einzelheiten der Implementierungen behandelt, d.h. es werden die Pseudocodes mit eventuell auftretenden Freiheiten konkret umgesetzt. Außerdem vergleichen wir dort jeweils verschiedene Umsetzungen untereinander.

In Abschnitt 4 werden dann die Ansätze miteinander verglichen.
Im letzten Teil wollen wir die gewählten Parameter rechtfertigen, indem wir einzelne Parameter verändern und die Auswirkungen beobachten.

Wir wollen nun eine Testumgebung mit verschiedenen Parametern vereinbaren.
Als Programmiersprache wurde `C` gewählt, da sie weit verbreitet ist und zu den schnellsten Sprachen zählt. Die `C`-Programme sowie einige ausgewählte relationale Programme sind im Anhang dargestellt.
Für eine sinnvolle graphische Darstellung bleibt, neben der Laufzeit bzw. der Anzahl der Pfeile, nur noch ein freier Parameter. Die beiden wichtigsten Parameter sind die Anzahl der Knoten und die Dichte des Graphen. Aufgrund der Ergebnisse werden wir zunächst die Dichte variabel lassen. Die Knotenzahl wird im Abschnitt 7.5.1 behandelt.
Da wir uns in der theoretischen Entwicklung auf stark zusammenhängende Graphen beschränkt haben, wollen wir dies auch hier tun. Deshalb wird die Dichte der Graphen im Bereich von 10%-100% liegen. Für geringere Dichten kann diese Eigenschaft nur bei wenigen Graphen garantiert werden. Allerdings ist diese Grenze abhängig von der Anzahl der Knoten. Graphen mit mehr Knoten sind schon bei geringerer Dichte stark zusammenhängend. Dabei ist eine Dichte von 100% im praktischen Einsatz nicht relevant, da dann sofort ein MEG angegeben werden kann. Die erhaltenen Werte dienen lediglich der Vervollständigung der Graphiken.
Wir wollen nun die restlichen Parameter der Testumgebung festlegen. Die eingegebenen Graphen haben 500 Knoten und werden als nachfolgerorientierte, geordnete, lineare Listen dargestellt. Die Testergebnisse sind auf einer SUN Ultra 5/10 Solaris 7 mit 333 MHz Prozessorgeschwindigkeit und 128 MB Hauptspeicher entstanden. Alle dargestellten Werte sind Durchschnittswerte von mindestens 50 Durchläufen pro Algorithmus und Dichte.

In diesem Abschnitt bezeichnen wir Graphen mit geringer Dichte als dünn, die mit hoher Dichte als dicke Graphen.

7.1 Minimierungsverfahren

Im Kapitel 4 haben wir den Algorithmus `TransRed` sowie die Verfeinerung `TransRed'` entwickelt. Für die zweite Version ist noch eine konkrete Implementierung der Vorberechnung anzugeben. Wir beschränken uns auf zwei wichtige Vertreter der Baumalgorithmen: den Tiefensuch- und Breitensuchbaum. Dabei ergeben sich aus den beiden Baumalgorithmen insgesamt 4 mögliche Kombinationen für die Vorberechnung, da wir auch gemischte Varianten zulassen können. Diese 4 Ansätze bezeichnen wir mit `DFS/DFS`, `DFS/BFS`, `BFS/DFS` und `BFS/BFS`. Der Vergleich zwischen diesen 4 Ansätzen ist in Tabelle 7.1 dargestellt, wobei dort schon eine für praktische Zwecke optimierte Version eingesetzt wurde, die wir in diesem Abschnitt entwickeln wollen.
Ein Vergleich des Algorithmus `TransRed` zu denen mit Vorberechnung liefert bezüglich der Zeit deutlich schlechtere Werte und wird daher nicht dargestellt. Dieser Ansatz benötigt bei dünnen Graphen etwa das Hundertfache, bei dickeren Graphen bis zum Tausendfachen der Zeit der Ansätze mit Vorberechnung. Die Vorberechnung erfüllt also auch praktisch ihren Zweck und verkleinert den eingegebenen Graphen schnell. Da `TransRed` auch bezüglich der

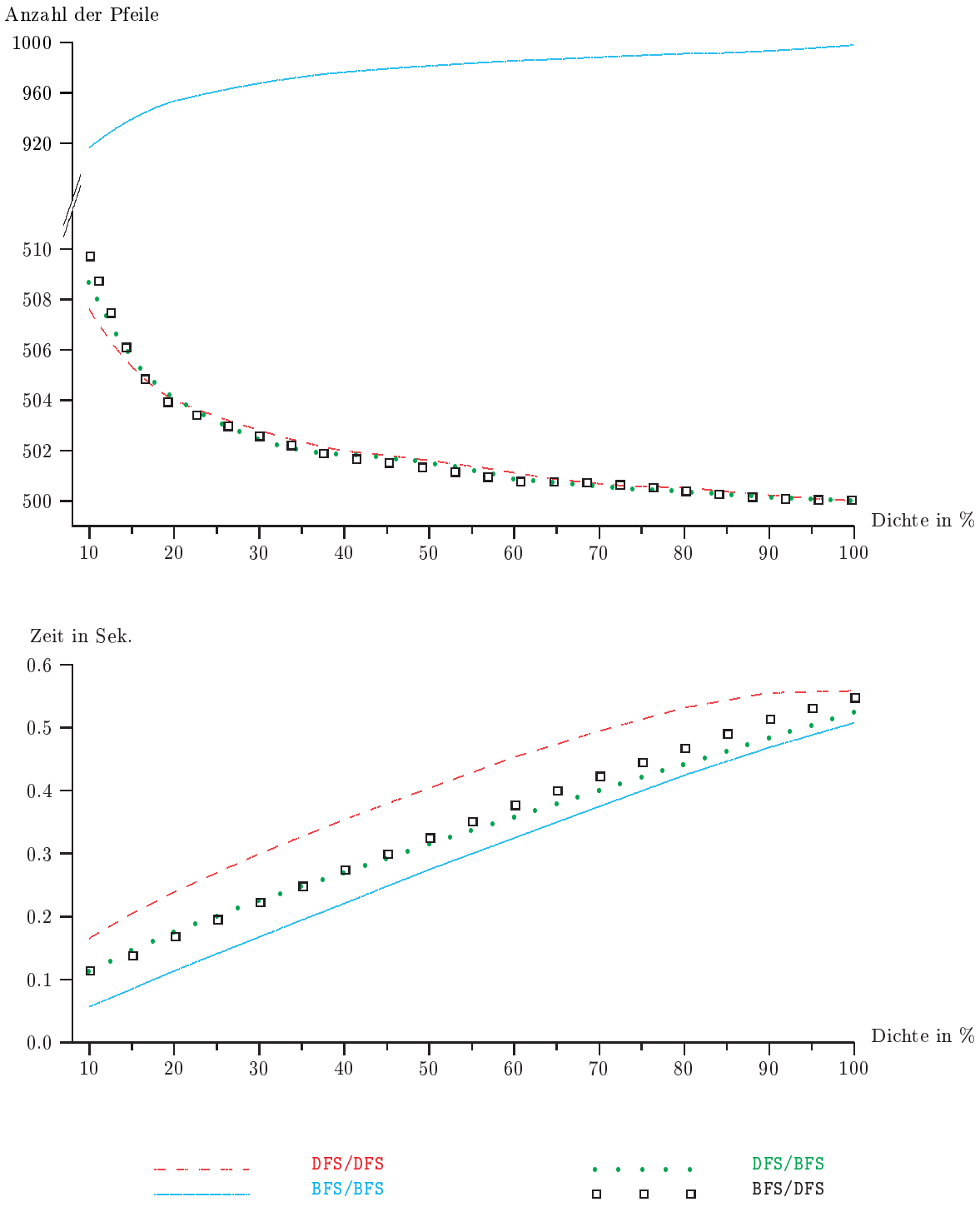


Abbildung 7.1: Minimierungsverfahren — Anzahl der Pfeile und Zeit

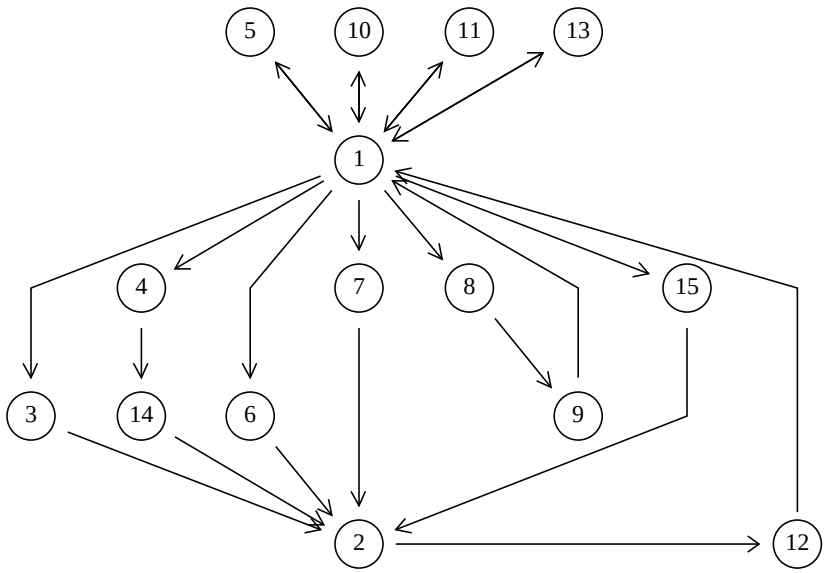


Abbildung 7.2: Beispiel `BFS/BFS`

Anzahl der Pfeile keine Steigerung liefert, werden wir diesen Ansatz nicht mehr verwenden.
Wir wollen nun die 4 Ansätze mit Vorberechnung vergleichen. Der Abbildung entnimmt man sofort, daß der einzige Ansatz ohne Tiefensuche `BFS/BFS` deutlich mehr Pfeile liefert als die anderen. Dies liegt darin begründet, daß bei der Breitensuche möglichst kurze Wege von der Wurzel zu allen Knoten berechnet werden. Je voller der Graph ist, umso mehr Pfeile enthält das Ergebnis. Die Vorberechnung liefert einen Graphen mit sehr kurzen Kreisen. Daher erhält man mit Hilfe von Lemma 6.1., daß das Ergebnis eine schlechte Güte besitzt.
Die drei anderen Ansätze liefern bezüglich der Anzahl der Pfeile sehr ähnliche Ergebnisse. Eine Begründung ist hier in der für diese Aufgabe guten Struktur des Tiefensuchbaumes zu suchen. Der Tiefensuchbaum liefert möglichst lange Wege von der Wurzel aus, es liegt eine Tendenz zu langen Kreisen vor.
Um den Unterschied zwischen den beiden Typen zu verdeutlichen, wird ein Beispielgraph auf das Verfahren `BFS/BFS` (Abbildung 7.2) und auf `DFS/BFS` (Abbildung 7.3) angewendet. Die Bilder zeigen die typischen enthaltenen Baumstrukturen der Breiten- bzw. der Tiefensuche. Der *breiten* Struktur eines Breitensuchbaumes steht die *tiefe* Struktur der Tiefensuche gegenüber. Es sind sowohl die Tendenz zu kurzen Kreisen in 7.2 als auch die zu langen Kreisen in 7.3 ersichtlich.
Bei Versuchen, bei denen Parameter variiert werden, fällt allerdings auf, daß das Verfahren `DFS/DFS` sehr abhängig von der zugrundeliegenden Ordnung ist. Sind alle Zwischenergebnisse geordnete Graphen oder testet man das Verfahren in der relationalen Version, so steigt mit wachsender Dichte auch bei diesem Verfahren die Anzahl der Pfeile an. Dies führt bei hohen Dichten zu ähnlich schlechten Ergebnissen wie beim Verfahren `BFS/BFS`. Da aber bei den beiden Verfahren `DFS/BFS` und `BFS/DFS` dieser Effekt nicht auftritt, sind diese `DFS/DFS` vorzuziehen. Ein Unterschied zwischen den beiden gemischten Verfahren bezüglich der Anzahl

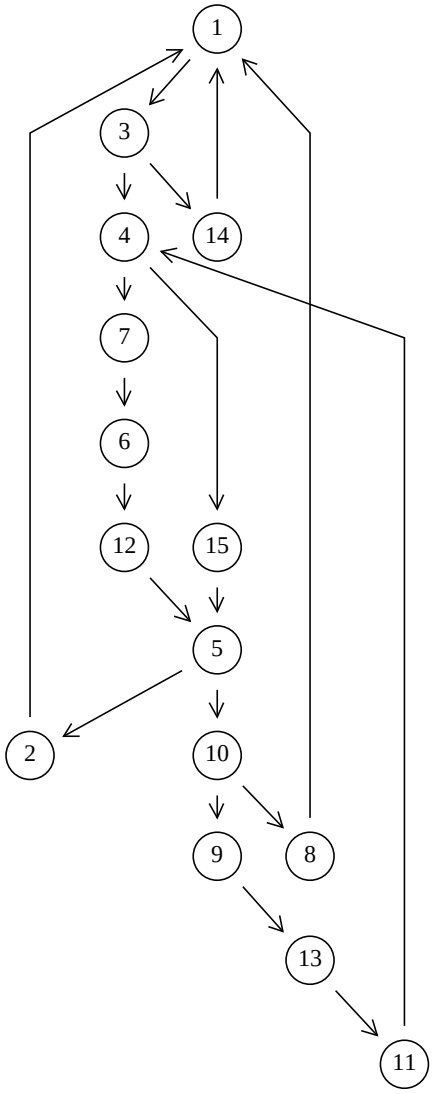


Abbildung 7.3: Beispiel `DFS/BFS`

der Pfeile ist kaum zu erkennen.

Die zeitliche Entwicklung in Abbildung 7.1 unten läßt erkennen, daß alle Verfahren eine ähnliche Laufzeit benötigen.
Unabhängig von der Vorberechnung ist die Implementierung der Prädikate $\mathcal{P}$ bzw. $\mathcal{Q}$, die aber für die verschiedenen Verfahren unterschiedliche Auswirkungen hat. Dabei ist eine Verwendung des Prädikates $\mathcal{P}$, die Berechnung der reflexiv-transitiven Hülle, aufgrund von längeren Laufzeiten nicht sinnvoll. Wir setzen daher das Prädikat $\mathcal{Q}$ um. Seien dazu ein Pfeil $e = (start, end)$ und ein Graph $G = (V, E)$ gegeben. Wir führen einen DFS-Durchlauf vom Knoten $start$ in $(V, E \backslash \{e\})$ aus. Das Prädikat $\mathcal{Q}$ ist genau dann erfüllt, wenn der Knoten end im DFS-Baum von $start$ enthalten ist.
Diese Vorgehensweise liefert zeitliche Ergebnisse, die vor allem von der Größe des Zwischen-

ergebnisses abhängen.

Um so einen Effekt zu vermeiden und die Auswertung insgesamt effizienter zu gestalten, kann eine notwendige Bedingung zur Reduzierung eines Pfeiles herangezogen werden. Danach ist e nur dann reduzierbar, wenn es zwei weitere Pfeile gibt, deren Start- bzw. Endpunkt auch *start* und *end* sind. Der Ausgangsgrad von *start* bzw. der Eingangsgrad von *end* muß also größer als 1 sein. Eine weitere Verbesserung des Tiefensuchdurchlaufes ist es, daß dieser nicht jedesmal den gesamten Graphen bearbeiten muß, sondern dann aufhören kann, wenn der Knoten *end* besucht wird. Mit dem vorgeschalteten Grad-Test sowie der kürzeren Tiefensuche wurden die Ergebnisse in Tabelle 7.1 erreicht. Diese optimierte Version des Prädikates $\mathcal{Q}$ wird auch im Anhang A.2 vorgestellt.
Der zeitliche Vergleich ergibt, daß das Verfahren `DFS/DFS` mehr Zeit benötigt als die anderen und `BFS/BFS` am schnellsten ist.

Zusammenfassend stellen wir fest, daß die Verfahren `BFS/BFS` und `DFS/DFS` schlechtere Ergebnisse als die beiden anderen liefern. Daher werden wir im folgenden das Minimierungsverfahren aufgrund von kleinen Vorteilen bei der Zeit und der Anzahl der Pfeile immer mit der Vorberechnung `DFS/BFS` sowie den oben angeführten Optimierungen verwenden.
Die jeweiligen imperativen bzw. relationalen Umsetzungen sind im Anhang A.2. bzw. B.2 enthalten. Die Implementierungen sind beide sehr einfach. Das Ziel dieses Ansatzes, einen einfachen Algorithmus zu entwickeln, kann hier deutlich an der Kürze des Codes abgelesen werden.

Die in der Einleitung angesprochenen Erwartungen werden nur teilweise erfüllt. So ist in der Abbildung 7.1 die Abhängigkeit zwischen der Dichte und der Laufzeit zu erkennen. Allerdings kann bei dem Verfahren `BFS/BFS`, und eingeschränkt auch bei `DFS/DFS`, aus steigender Dichte keine geringere Pfeilanzahl gefolgert werden.

Auffallend ist, daß eine gemischte Vorberechnung die besten Ergebnisse liefert. Dabei sind Anwendungen der Tiefen- oder Breitensuche in der Graphentheorie sehr verbreitet, Verfahren, die beides verwenden, aber sehr selten.

7.2 Algorithmus von Simon

Auch der Algorithmus von Simon wurde in `C` implementiert.
Wir wollen mit Hilfe der Abbildung 7.4 das Verfahren von Simon mit dem Minimierungsverfahren vergleichen. Es ist zu erkennen, daß die beiden Verfahren bezüglich der zurückgegebenen Pfeile dieselben Werte liefern. Allerdings ist der Zeitaufwand beim Ansatz von Simon deutlich größer. Bei dünnen Graphen beträgt er ungefähr das Doppelte, bei dicken sogar das Sechsfache des Minimierungsalgorithmus.
Der Grund hierfür ist nicht darin zu suchen, daß man durch den korrigierten Ansatz eine quadratische Laufzeit erhält, sondern darin, daß der Code lang ist und daß alle Pfeile einzeln nacheinander betrachtet werden. Außerdem sind sehr viele Ergebnisse zwischenzuspeichern. Eine Implementierung des Originalalgorithmus liefert zwar keine transitiven Reduktionen,

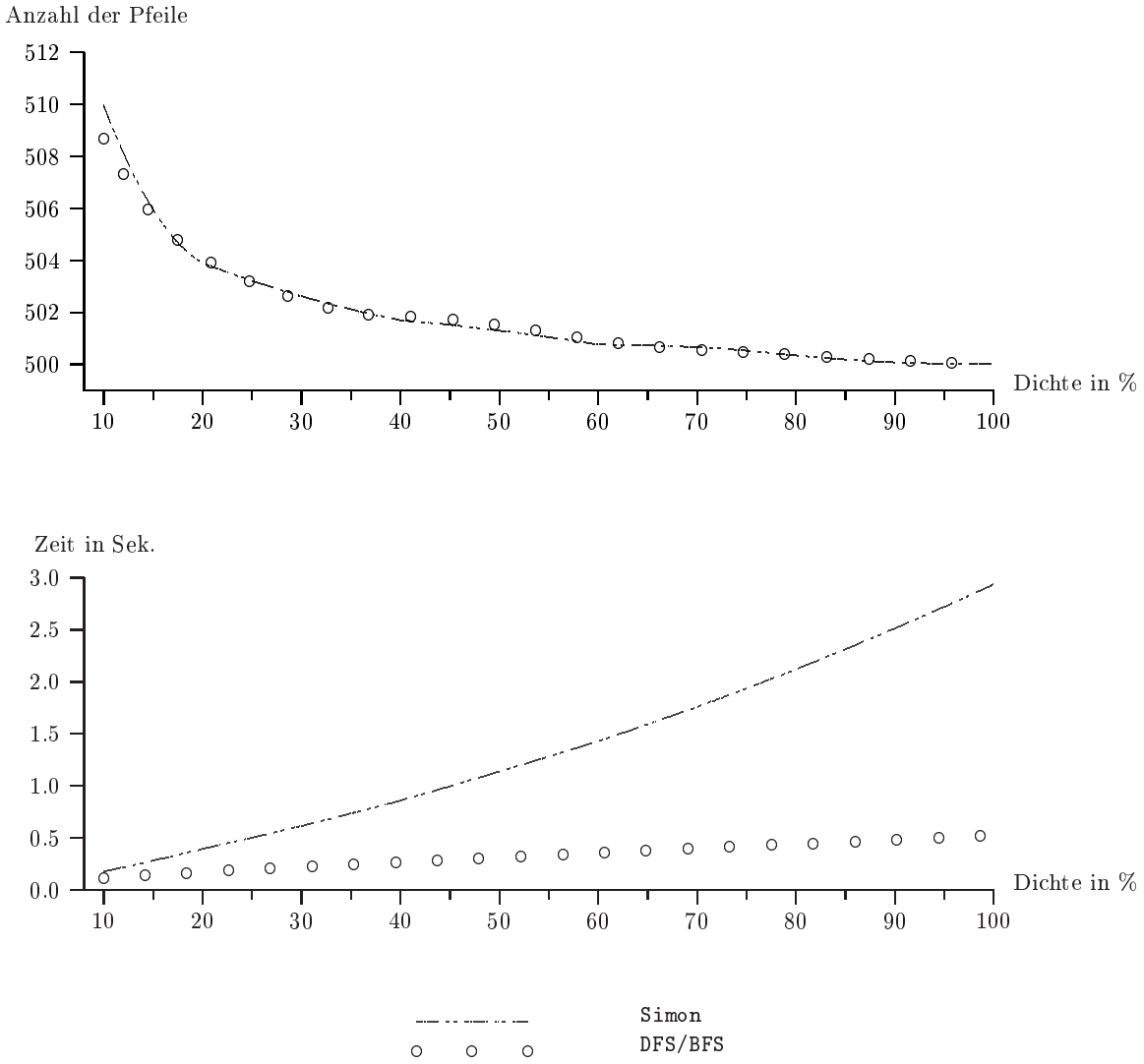


Abbildung 7.4: `Simon` vs. `DFS/BFS` — Anzahl der Pfeile und Zeit

zeigt aber auch, daß die Laufzeit in keinem Fall durch den neu hinzugefügten Teil verschlechtert wird. Der ursprüngliche Ansatz ist noch langsamer als der korrigierte.
Eine Verbesserungsmöglichkeit des Verfahrens ist wie beim Minimierungsverfahren die Einführung einer Vorberechnung. Dadurch ergeben sich bessere Laufzeiten, diese reichen aber trotzdem nicht an das Verfahren `DFS/BFS` heran, so daß auf einen Einsatz des Verfahrens von Simon verzichtet werden kann.

Ähnlich den Erläuterungen zum Verfahren `DFS/BFS` ergeben sich auch bei `Simon` die erwarteten Abhängigkeiten zwischen der Dichte und der Zeit bzw. der Anzahl der Pfeile.

7.3 Approximativer Algorithmus von Khuller et al.

In diesem Abschnitt wollen wir den in Kapitel 6 entwickelten Algorithmus `CONTRACT-CYCLES`$_k$ speziell für den fast-linearen Fall $k = 3$ praktisch umsetzen.
Eine Implementierung ist im Anhang A.4. und B.2. dargestellt. Wir wollen im folgenden einige Variationen vorstellen, um Schwächen des Verfahrens zu vermeiden.

Original

Eine direkte Umsetzung des Pseudocodes, wir nennen sie im folgenden `Contract`, ergibt eine i.a. schlechte Approximation eines MEG.
Die Anzahl der Pfeile beträgt ungefähr das 1,5fache der Knotenzahl, dies zeigt die Abbildung 7.5.
Zur Begründung dieser Tatsache, sei $G = (V, V \times V)$ der volle Graph. Da der eingegebene Graph geordnet ist, ist die Abarbeitungsreihenfolge der Pfeile (ohne die Schlingen) wie folgt gegeben: $(1,2), (2,1), (2,3), (3,1), (3,2), (3,4), (4,1), \ldots$. Aufgrund der vorgegebenen Ordnung werden zuerst die Pfeile zu besuchten, dann die zu unbesuchten Knoten abgearbeitet. Nach dem Besuch des Pfeiles $(3,1)$ wird ein Kreis geschlossen, ebenso nach $(5,1)$, $(7,1)$ usw.. Dies ergibt, daß die Kreise, die geschlossen werden, immer nur die Länge 3 besitzen, der letzte geschlossene eventuell auch nur die Länge 2. Durch Abzählen bei $|V|$ Knoten erhalten wir ungefähr $\frac{3|V|}{2}$ Pfeile. Diese Argumentation gilt auch für nicht volle Graphen, da die Tendenz zu kleinen Kreisen erhalten bleibt.
Zu bemerken ist außerdem, daß die Pfeilanzahl bei steigender Dichte wächst, also auch die Güte schlechter wird.

Ordnungen

Aus den Schwächen des vorherigen Verfahrens `Contract` wollen wir einen Algorithmus entwickeln, der bessere Approximationen liefert, wobei wir in Kauf nehmen müssen, daß dadurch die Laufzeit wächst.
Ideengeber dieses Ansatzes ist zum einem die Argumentation zur schlechten Approximation von `Contract` und zum anderen die in B.3 dargestellte relationale Implementierung des Verfahrens, in dem keine rekursive Tiefensuchversion, sondern eine entrekursivierte Version als Basis dient. Wir wollen die Nachfolger eines Knotens nicht in einer vorgegebenen Ordnung abarbeiten, sondern zuerst die weißen und dann die restlichen besuchen.
Wir nennen diesen Ansatz `Contract_white`, seine Implementierung ist in A.4 und relational in B.3 dargestellt. Es handelt sich dabei um eine direkte Umsetzung der dargestellten Idee, die im Pseudocode auch in Tabelle 7.1 zu sehen ist. An dieser Stelle muß nur die Prozedur `DFS-CONTRACT` geändert werden. Dabei ist klar, daß alle Knoten und Pfeile abgearbeitet werden, die **forall**-Schleife also aufgeteilt wird. Die Korrektheit bleibt dadurch erhalten.
Aus der Abbildung 7.5 ist zu erkennen, daß die Approximation, die der Algorithmus `Contract_white` liefert, deutlich besser ist als die der Originalumsetzung.

```
procedure DFS-CONTRACT-WHITE(u : V)
  to−active[FIND(u)] := current;
  forall w ∈ V with (u, w) ∈ E do
    if w is not yet visited
      then MAKE-SET(w);
           to−active[FIND(u)] := (u, w);
           from−root[FIND(w)] := (u, w);
           DFS-CONTRACT(w);
           to−active[FIND(u)] := current;
    end;
  end;
  forall w ∈ V with (u, w) ∈ E do
    if (v, w) is not yet visited
      then if FIND(u) ≠ FIND(w)
             then (x, y) := from−root[FIND(u)];
                  if FIND(x) = FIND(w)
                    then to−root[FIND(u)] := (u, w);
                    else (x, y) := from−root[FIND(w)];
                         if FIND(x) ≠ FIND(u)
                           then CONTRACT-CYCLE(w);
                                S := S ∪ {(u, w)};
                         end;
                  end;
           end;
    end;
  end;
  to−active[FIND(u)] := nil;
```

Tabelle 7.1: Pseudocode DFS-CONTRACT-WHITE

Iteratives Ausführen

Eine weitere Verbesserungsmöglichkeit besteht darin, den Algorithmus Contract iterativ auszuführen, d.h. Contract wieder auf die Ausgabe des vorherigen Durchlaufes anzuwenden. Wir nennen diesen Ansatz im folgenden Contract_iter. Dabei stellt sich die Frage, wie oft er hintereinander ausgeführt werden soll. Die erste Möglichkeit besteht darin, den Algorithmus solange anzuwenden, bis sich das Ergebnis nicht mehr verändert. Die zweite Möglichkeit ist es, die Anzahl der Ausführungen vorher zu bestimmen, also konstant zu halten. Dabei wird bei der ersten Variante die theoretische Laufzeit verändert. Dort können bis zu $\mathcal{O}(|V|)$ Durchläufe auftreten.
Da die praktischen Tests gezeigt haben, daß es höchstens 4 solcher Durchläufe gibt, fallen die beiden Ansätze in der Praxis zusammen. Um die Approximationsgüte zu verbessern, müssen noch die Schwächen des Originalansatzes ausgeräumt werden. Wenden wir auf das Ergebnis er-

```
function CONTRACT-ITER(G : graph)
  I := G;
  bool := true;
  repeat
    H := I;
    if bool
      then I := CONTRACT − CYCLE3−ORDNUNG(I, <);
      else I := CONTRACT − CYCLE3−ORDNUNG(I, >);
    end;
    bool := ¬bool;
  until H = I;
  return I
```

Tabelle 7.2: Pseudocode CONTRACT-ITER

neut den Algorithmus an, so werden dieselben Pfeile nach der vorgegebenen Ordnung in derselben Reihenfolge besucht. Die Tendenz, dieselben Kreise zu schließen, ist dabei sehr hoch. Daher fügen wir dem Algorithmus eine zugrundeliegende Ordnung zu, die wir nach jedem Durchlauf ändern. Aus diesen Ideen erhält man den Algorithmus CONTRACT-ITER, dargestellt als Pseudocode in Tabelle 7.2. Dabei setzen wir eine Version des Algorithmus CONTRACT-CYCLE$_3$-ORDNUNG voraus, der die Nachfolger gemäß einer gegebenen Ordnung besucht.
Auch dieser Ansatz verbessert die Anzahl der zurückgegebenen Pfeile gegenüber dem Originalalgorithmus, benötigt allerdings eine längere Laufzeit.

Vergleich der Khuller-Ansätze

Dieser Abschnitt dient dazu, die drei erarbeiteten Ansätze zu vergleichen. Die Ergebnisse sind in Abbildung 7.5 zusammengefaßt.
Im oberen Bild sind die schon erwähnten Eigenschaften bezüglich der Anzahl der Pfeile abgebildet. Der Ansatz `Contract` gibt ungefähr das 1,5fache der Knotenzahl zurück, die beiden anderen Ansätze ergeben deutlich bessere Werte. Bei diesen ist auch die erwartete Abhängigkeit zwischen steigender Dichte und geringerer Pfeilanzahl zu erkennen.
Der Unterschied zwischen den modifizierten Verfahren bezüglich der Anzahl der Pfeile ist gering, bei Graphen mit hohen Dichten ist der Unterschied mit einem Pfeil am größten. Der Algorithmus `Contract_iter` gibt in einem vollen Graphen einen Pfeil zuviel zurück. Dies ergibt sich bei gerader Knotenanzahl, bei ungerader Knotenzahl erhält man die optimale Anzahl.
Die Begründung dafür ist in der Argumentation zu `Contract` zu suchen, da bei ungerader Anzahl der letzte geschlossene Kreis die Länge 3 besitzt. Da aber hohe Dichten in der Praxis eher selten vorkommen und der Unterschied von einem Pfeil nicht groß ist, wollen wir diesen Effekt hier vernachlässigen und gehen also von etwa gleichen Pfeilanzahlen bei den beiden modifizierten Verfahren aus.

Bei den Laufzeiten ist zu erkennen, daß der Ansatz `Contract` am schnellsten ist. Dies ist nach

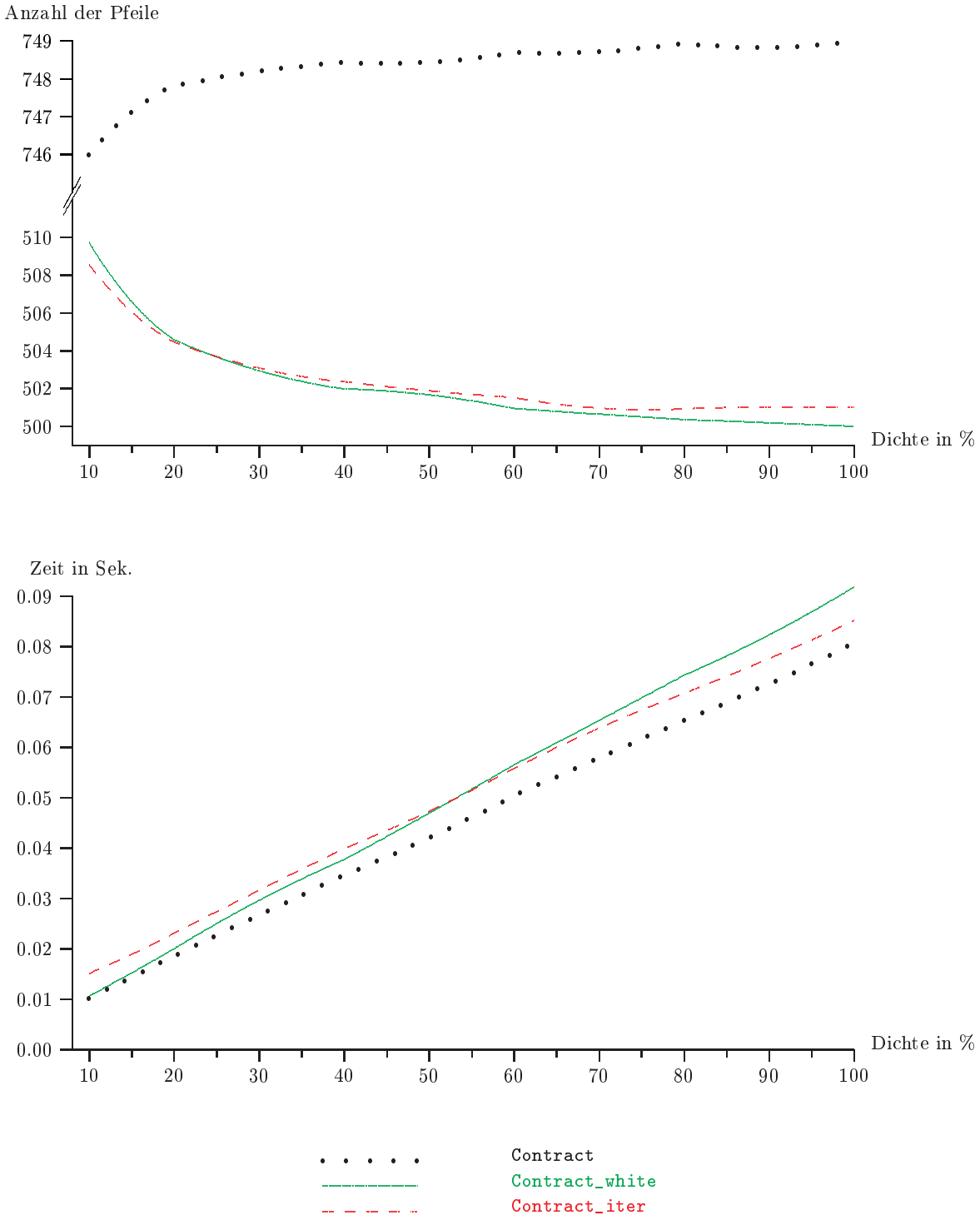


Abbildung 7.5: `Contract` vs. `Contract_iter` vs. `Contract_white` — Anzahl der Pfeile und Zeit

der Entwicklung der beiden modifizierten Ansätze klar, da sich dort ein erhöhter Aufwand ergibt. Beim Ansatz `Contract_white` ist dies durch die Unterscheidung von besuchten und unbesuchten Pfeilen klar, bei `Contract_iter` durch die mehrmalige Anwendung.
Dabei ist durch diese Argumentation auch der Verlauf der beiden modifizierten Ansätze klar. Der Ansatz `Contract_white` ist abhängig von der Dichte, denn bei ihm müssen alle Pfeile zweimal durchlaufen werden. Dagegen benötigt `Contract_iter` nur jeweils weitere Durchläufe auf Graphen mit weniger als 750 Pfeilen. Nur der erste Durchlauf ist abhängig von der Dichte. Dies drückt sich schon im Verlauf von `Contract` aus. Man erhält also einen von der Dichte unabhängigen konstanten Zeitunterschied zwischen der einmaligen und der mehrmaligen Anwendung von `Contract`.
Daher ergeben sich für dünne Graphen bessere Laufzeiten für `Contract_white`, bei dickeren für `Contract_iter`. Unabhängig davon ist bei steigender Dichte bei allen Verfahren eine längere Laufzeit zu beobachten.

Zusammenfassend stellen wir fest, daß sich die deutlich geringere Anzahl der Pfeile in dem höheren Zeitaufwand der modifizierten Verfahren niederschlägt. Bei dünnen Graphen ist also `Contract_white`, bei dicken Graphen das Verfahren `Contract_iter` zu empfehlen. Ist man an einer möglichst schnellen, aber nicht unbedingt an einer guten Approximation interessiert, so bietet sich die Verwendung des Verfahrens `Contract` an.

7.4 Vergleich der Ansätze

Wir wollen nun die verschiedenen Ansätze samt ihrer Variationen vergleichen. Dabei berücksichtigen wir den Ansatz von Simon nicht, da der Ansatz `DFS/BFS` bei gleicher Anzahl der Pfeile zeitlich deutlich besser ist.
Da die Ansätze aus Abschnitt 7.3 nur Approximationen berechnen, i.a. aber keine transitiven Reduktionen, ist die Anzahl der zurückgegebenen Pfeile größer als die von `DFS/BFS`. Um die Ansätze vergleichbar zu machen, wollen wir die approximativen Ansätze um eine Nachminimierung erweitern. Dazu ersetzen wir die Vorberechnung des Minimierungsansatzes durch die jeweiligen Ansätze. Am Beispiel wird dies in Tabelle 7.3 dargestellt. Ein Vorteil dieser Ansätze ist es, daß dabei die theoretische Schranke aus Theorem 6.1. gilt. Wir nennen diese im folgenden `TransRed_Contract`, `TransRed_Contract_iter` und `TransRed_Contract_white`.
Ein Vergleich der Algorithmen für transitive Reduktionen ist aus Abbildung 7.6 ersichtlich. Bei allen vier Verfahren ist im oberen Bild zu erkennen, daß die Anzahl der zurückgegebenen Pfeile sehr ähnlich ist. Die Unterschiede beschränken sich auf höchstens einen Pfeil und lassen sich wie im letzten Abschnitt begründen.

function `TransRed-CONTRACT-CYCLE`$_3$`-WHITE`$(G : graph)$
  $H :=$ `CONTRACT` $-$ `CYCLE`$_3-$`WHITE`(G);
  $I :=$ `TransRed`$'(H)$;
  return I

Tabelle 7.3: Pseudocode `TransRed-CONTRACT-CYCLE`$_3$`-WHITE`

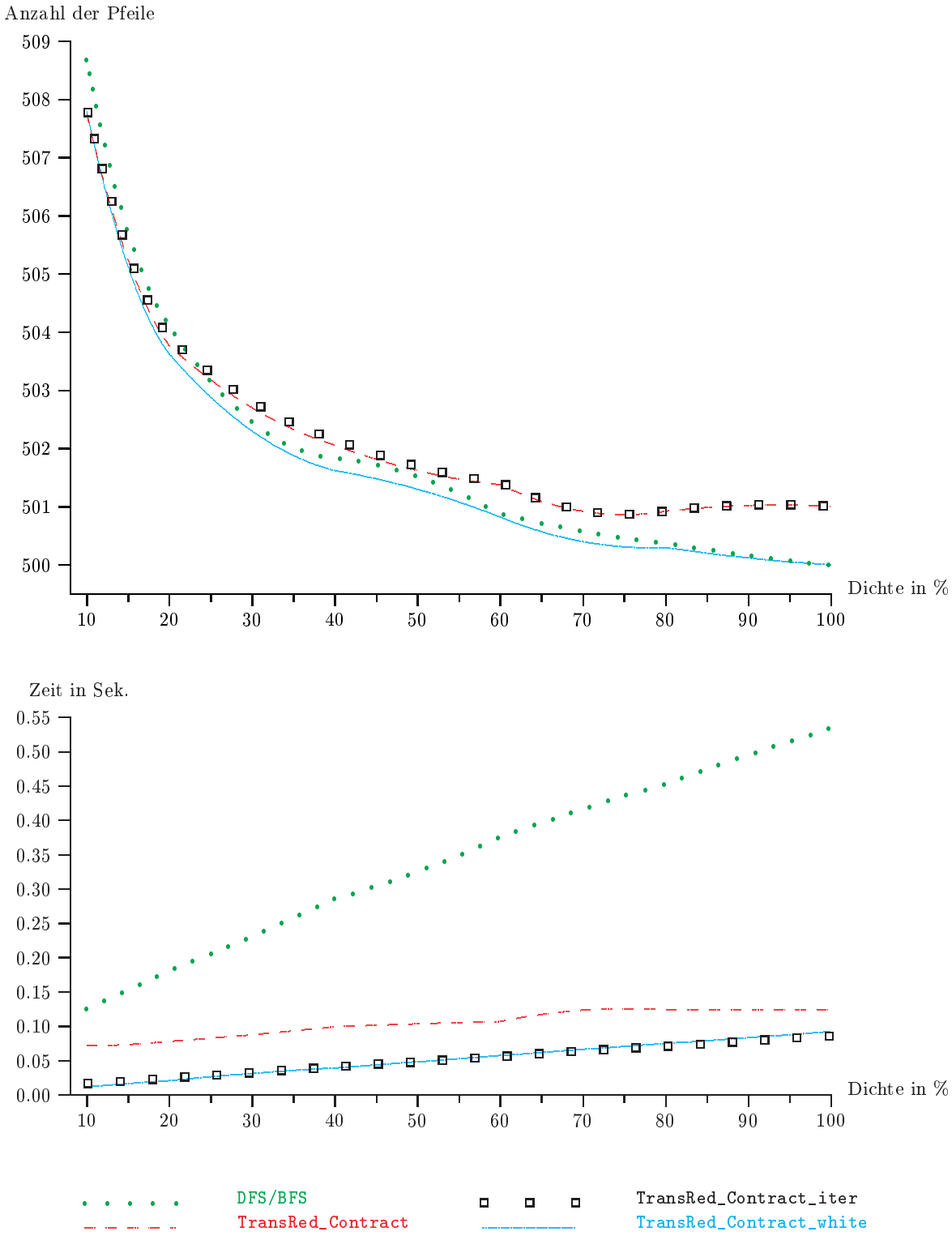


Abbildung 7.6: `DFS/BFS` vs. `TransRed_Contract` vs. `TransRed_Contract_iter` vs. `TransRed_Contract_white` — Anzahl der Pfeile und Zeit

Die Güte ist zwar nicht berechenbar, aber an der absoluten Anzahl der Pfeile ist zu erkennen, daß der Algorithmus eine sehr gute praktische Näherung liefert. Es kann vermutet werden, daß die meisten Graphen bis auf ein oder zwei Pfeile an das Optimum heranreichen.
Ein Vergleich der Zeiten im unteren Bild zeigt, daß das Verfahren `DFS/BFS` deutlich gegenüber den anderen abfällt. Die beiden modifizierten Verfahren `TransRed_Contract_iter` und `TransRed_Contract_white` sind am schnellsten. Es ergeben sich bei verschiedenen Dichten dieselben Unterschiede, die bei den Algorithmen `Contract_iter` und `Contract_white` beobachten werden. Der Ansatz `TransRed_Contract` fällt zeitmäßig gegen diese Verfahren ab, da sich dort nach der Vorberechnung mehr Pfeile im Zwischenergebnis befinden.
Bei der nachgeschalteten Minimierung wird die in Abschnitt 7.1 entwickelte optimierte Version `TransRed'` eingesetzt. Dabei ist zu beachten, daß bei Graphen, die nur wenige Pfeile mehr als Knoten enthalten, fast alle Knoten den Ein- und Ausgangsgrad 1 besitzen. Daher müssen nur sehr wenige aufwendige DFS-Durchläufe durchgeführt werden. Diese Argumentation trifft auch auf `DFS/BFS` zu. So sind nach der Vorberechnung bis zu doppelt soviele Pfeile wie Knoten im Zwischenergebnis enthalten. Außerdem benötigt die Berechnung zweier Bäume schon mehr Zeit als ein, wenn auch modifizierter, Tiefensuchdurchlauf, der dem Ansatz von Khuller entspricht. Außerdem muß bei dem langsamsten Verfahren zweimal eine Transpositionsoperation angewendet werden, die sehr zeitaufwendig ist.
Ein Vergleich zeigt, daß der Ansatz von Simon bei dünnen Graphen das Fünfzehnfache, bei dicken Graphen das bis zu Vierzigfache der Zeit des besten Ansatzes benötigt und deshalb nicht akzeptabel ist.

Zusammenfassend kann gesagt werden, daß zur Berechnung von transitiven Reduktionen die beiden Verfahren `TransRed_Contract_white` und `TransRed_Contract_iter` zu empfehlen sind. Werden solche Verfahren in der Praxis eingesetzt, ist es Geschmackssache, ob überhaupt transitive Reduktionen oder nur Approximationen berechnet werden. Der zeitliche höhere Aufwand der Nachminimierung schlägt sich bei den beiden schnellsten Verfahren `TransRed_-Contract_iter` und `TransRed_Contract_white` im Schnitt nur durch die weitere Reduzierung eines Pfeiles nieder. Die approximativen Verfahren `Contract_white` und `Contract_iter` sind daher genauso zu empfehlen.

7.5 Testparameter

In diesem Abschnitt wollen wir nun die in der Einleitung zu diesem Kapitel festgelegten Parameter verändern und die Auswirkungen auf die Testergebnisse darstellen.
Dabei ist es nicht möglich, alle Kombinationen zu betrachten. Daher werden wir jeweils einen Parameter verändern und die anderen konstant halten. Die Argumentation in diesem Abschnitt wird den Einsatz der speziell gewählten Testumgebung rechtfertigen. Wir können die Ergebnisse aus den Abschnitten 7.1 bis 7.3 also auch auf andere Umgebungen erweitern.

7.5.1 Knotenzahl

Zuerst wollen wir uns mit der Knotenzahl beschäftigen. Diese hatten wir auf 500 festgesetzt. Wir betrachten dazu die Abbildungen 7.7–7.10, in denen grundsätzliche Ergebnisse auf Graphen mit 100 und 1000 Knoten dargestellt sind.
Es wird sich zeigen, daß sich sowohl die Anzahl der Pfeile als auch die Zeiten entsprechen.
Bevor wir dies im einzelnen erläutern, wollen wir bemerken, daß eine 100%ige Übereinstimmung aus folgendem Grund nicht zu erwarten ist: Graphen mit größerer Knotenzahl sind in der Regel schon bei geringer Dichte stark zusammenhängend, so daß ein direkter Vergleich der Dichten untereinander, gerade bei geringen Dichten, nicht möglich ist. Die exakte Übereinstimmung ist aber auch notwendig. Wichtig ist, daß die Verfahren relativ zueinander gleiche Ergebnisse liefern.

Die Abbildungen 7.7 und 7.8 zeigen die Testergebnisse des Minimierungsverfahrens, in denen die vier verschiedenen Vorberechnungen gegenübergestellt werden.
Die erste Abbildung stellt die Anzahl der zurückgegebenen Pfeile dar, oben für Graphen mit 100 Knoten, unten für solche mit 1000 Knoten. Ein Unterschied der Kurvenverläufe ist kaum zu erkennen. Eine prozentuale Skalierung würde hier kleine Unterschiede zeigen. So ist beim Verfahren `BFS/BFS` zu erkennen, daß bei 10%iger Füllung das 1,9fache bzw. das 1,5fache der Knotenzahl zurückgegeben werden. Der Unterschied ist aber durch das obige Argument erklärbar.
Die Abbildung 7.8 zeigt die zu 7.7 entsprechenden Zeitverläufe des Minimierungsverfahrens. Auch hier zeigt sich, daß die Verläufe der Kurven sehr ähnlich sind. Die Unterschiede bei geringen Dichten lassen sich mit demselben Argument wie oben erklären.

Die Abbildungen 7.9 und 7.10 zeigen das Verhalten bei verschiedenen Knotenzahlen der drei grundsätzlichen Verfahren: dem Minimierungsverfahren, dem Ansatz von Simon und der ersten Variante des Verfahrens von Khuller.
Sowohl die Anzahl der Pfeile in Abbildung 7.9 als auch die Zeitverläufe in 7.10 zeigen, daß diese Verfahren jeweils unabhängig von der Anzahl der Knoten sind. Außer Unterschieden bei geringen Dichten (s.o.) sind der kurvige Verlauf bei Khuller (100 Knoten) zu erkennen und im Zeitdiagramm beim Verfahren von Simon der lineare bzw. nichtlineare Verlauf. Der erste Unterschied erscheint zwar groß, zeigt aber nur die Abweichung um einen Pfeil und läßt sich durch die Argumentation in Abschnitt 7.3 erklären. Der zweite Unterschied ist ein Effekt, der im Abschnitt Hardware zum Tragen kommt. Die Speicherung vieler Zwischenergebnisse erfordert viel Hauptspeicher. Die Auswirkungen werden hier bei Graphen mit 1000 Knoten und hoher Dichte sichtbar. Ein voller Graph mit 1000 Knoten benötigt bereits etwa 8 MB Speicher. Da das Verfahren `Simon` aufgrund vieler Zwischenergebnisse viel Speicherplatz benötigt, steigt die Ausführungszeit erheblich.

Die Abbildungen sind nur beispielhaft ausgewählt. Auch alle anderen schon dargestellten Graphiken lassen sich auf andere Knotenzahlen übertragen. Wir können also unabhängig von der Knotenzahl argumentieren.
Allerdings kann dies nur dann gelten, wenn die Speichergröße und die Größe des Graphen in einem zueinander sinnvollen Rahmen bleibt. So ergibt sich, wie schon in der Einleitung

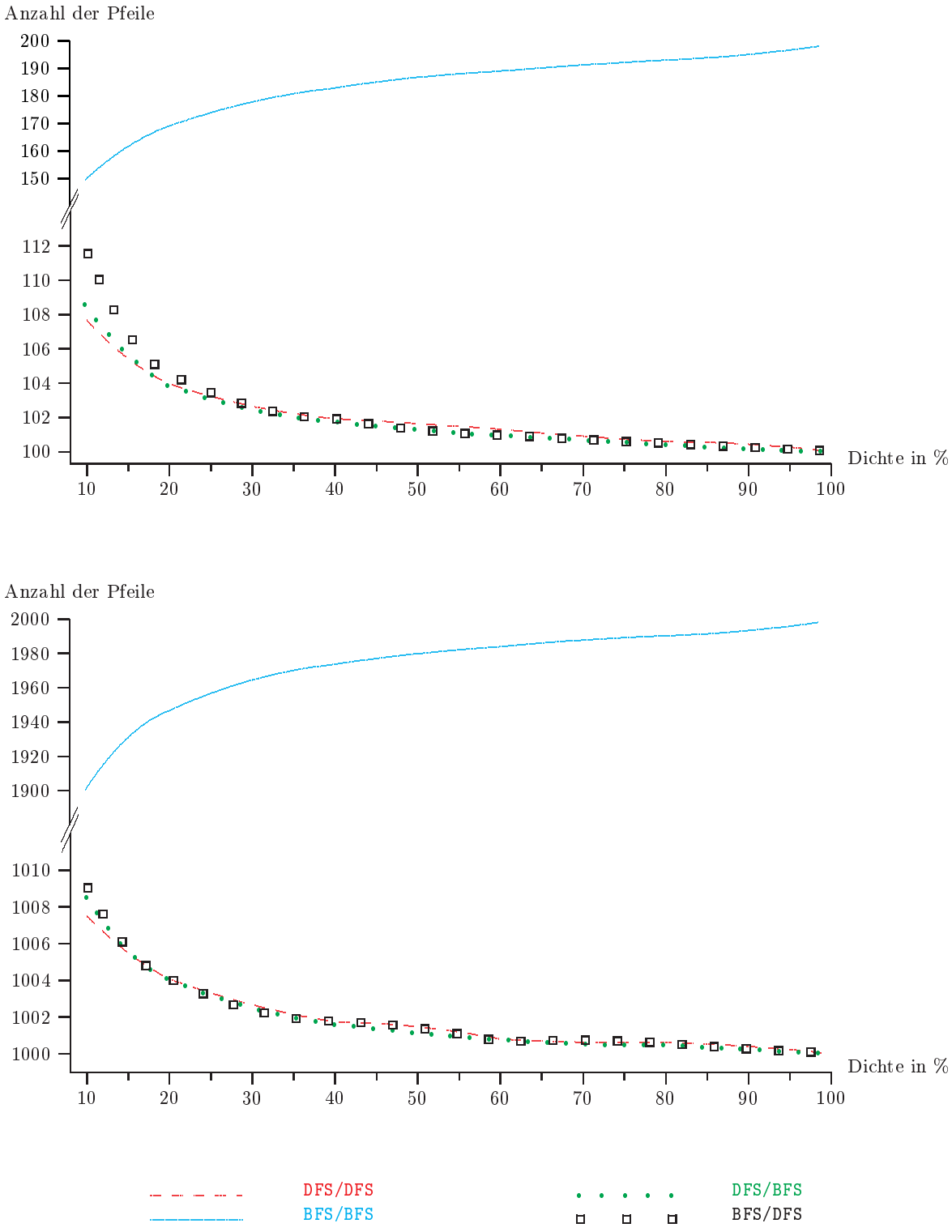


Abbildung 7.7: Minimierungsverfahren — Anzahl der Pfeile (oben 100, unten 1000 Knoten)

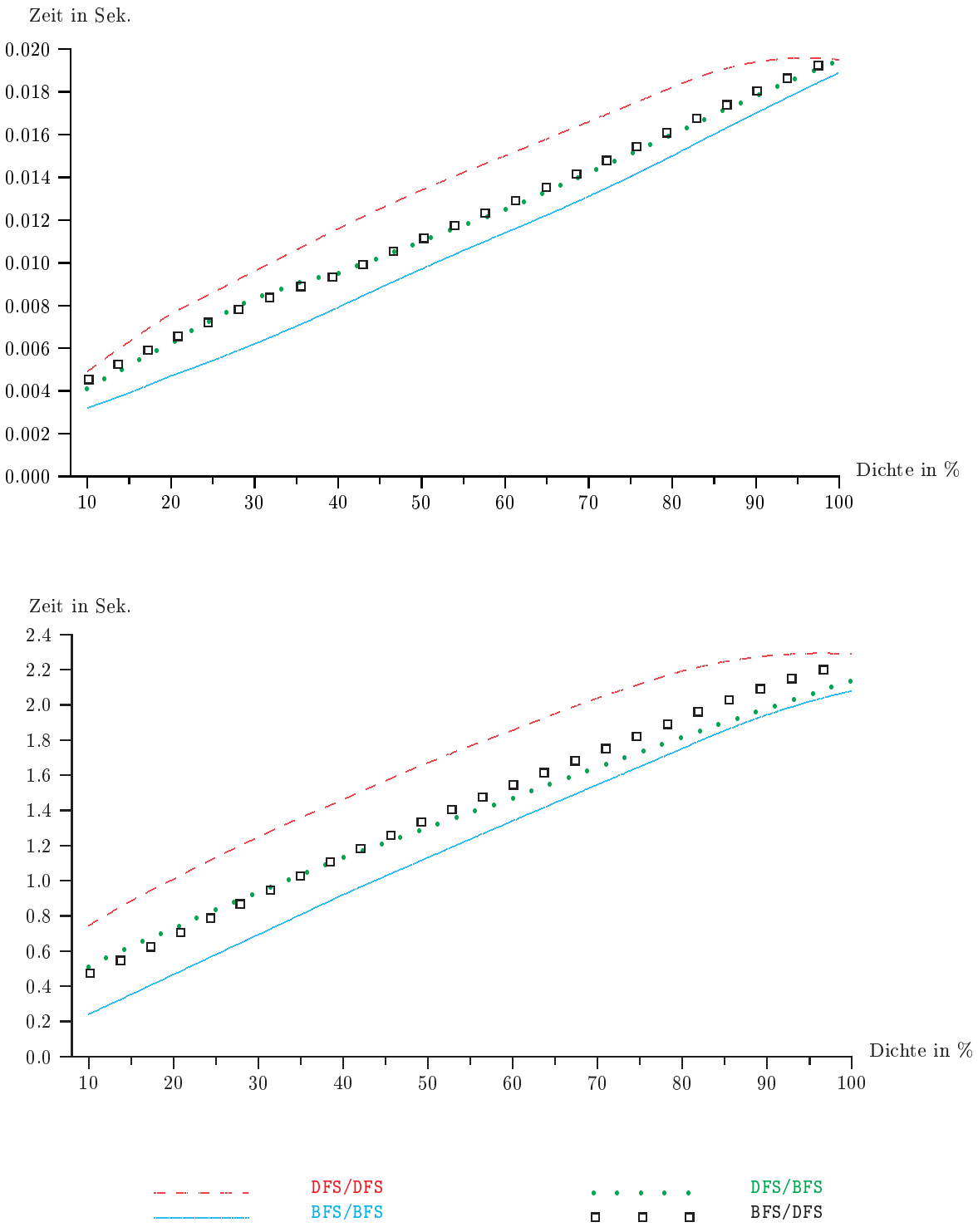


Abbildung 7.8: Minimierungsverfahren — Zeit (oben 100, unten 1000 Knoten)

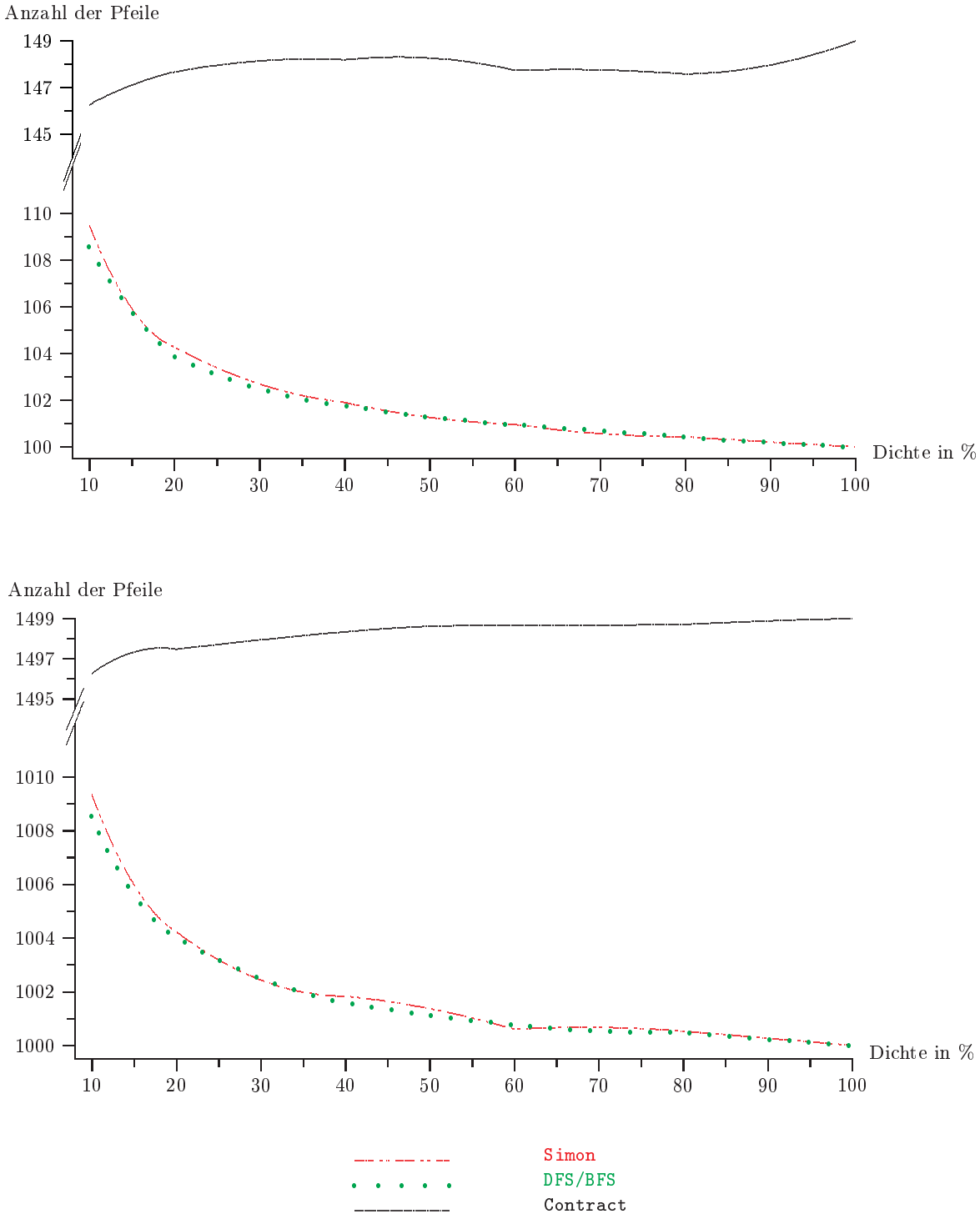


Abbildung 7.9: `DFS/BFS` vs. `Simon` vs. `Contract` — Anzahl der Pfeile (oben 100, unten 1000 Knoten)

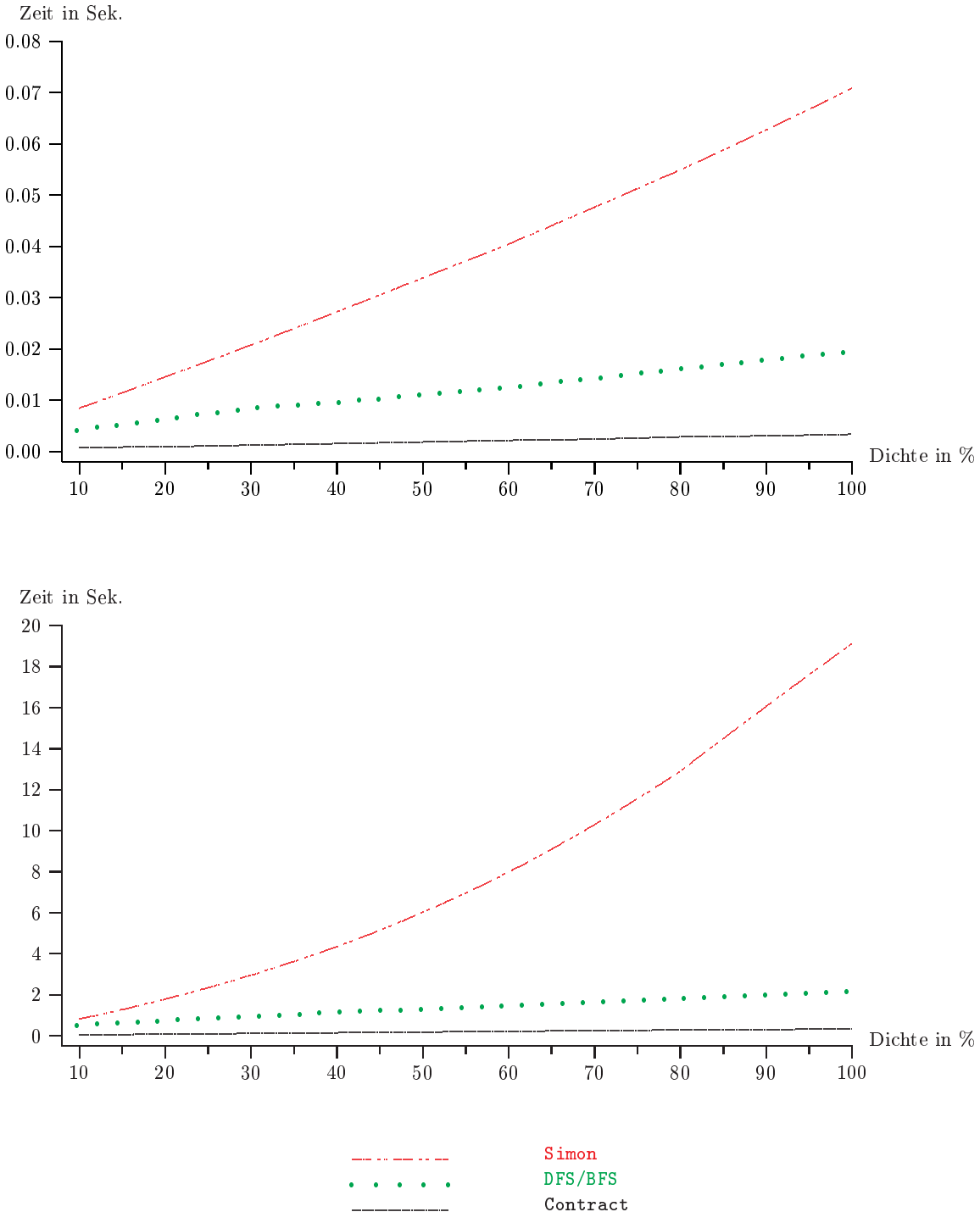


Abbildung 7.10: `DFS/BFS` vs. `Simon` vs. `Contract` — Zeit (oben 100, unten 1000 Knoten)

erwartet, daß bei größerer Knotenzahl auch die Bearbeitungszeit wächst. Bei Knotenzahlen bis 500 ergibt sich ein linearer Zusammenhang zwischen der Dichte des Graphen und der Laufzeit. Dies kann bei größeren Graphen nicht mehr erwartet werden, da die zu speichernden Zwischenergebnisse nicht alle gleichzeitig in den Hauptspeicher passen. Dieser Effekt konnte schon in Abbildung 7.10 beim Verfahren von Simon beobachtet werden.
Bei Dichten unter 20% und der vorgegebenen Hauptspeichergröße von 128 MB ergeben sich so bei den verschiedenen Verfahren unterschiedliche Schwellen der Knotenzahlen, die den linearen Zusammenhang zerstören. Von diesen Knotenzahlen ab steigen die Laufzeiten erheblich schneller an. Bei dem speicherintensivsten Verfahren (`Simon`) sind dies rund 3000 Knoten, beim Minimierungsverfahren ungefähr 5000, beim Ansatz von Khuller bis zu 10000 Knoten. Bei geringeren Dichten von unter 1% sind beim letzten Ansatz sogar Graphen mit 20000 Knoten in rund 10 Sekunden abarbeitbar.

7.5.2 Zufallsgraphen

Wir haben in unserer Testumgebung eine geordnete Darstellung der eingegebenen Zufallsgraphen angenommen. Dies ist auch für Testzwecke sinnvoll, da dann ein Graph nur eine Darstellung besitzt. Wir lassen nun auch ungeordnete Graphen zu und betrachten die Auswirkungen auf die Testergebnisse.

Es sind keine Veränderungen in der Laufzeit der einzelnen Algorithmen zu erkennen. Allerdings ergibt sich beim ersten Ansatz von Khuller eine Änderung der Pfeilanzahl. Das in Abschnitt 7.3 beobachtete Verhältnis von 1,5:1 der Pfeile zu der Anzahl der Knoten, das wir auf das Schließen von kurzen Kreisen zurückgeführt hatten, tritt nun nicht so stark auf. Wir erhalten also weder eine hohe Wahrscheinlichkeit dafür, einen schwarzen Knoten noch einen weißen Knoten zu besuchen. Die Auswahl ist zufällig. Das Verhältnis verringert sich dadurch auf durchschnittlich 1,3:1, ist aber damit noch deutlich schlechter als die beiden modifizierten Verfahren. Durch die geringere Pfeilanzahl benötigt eine eventuelle Nachminimierung weniger Zeit, aber die grundsätzlichen Ergebnisse des Abschnittes 7.4 bleiben erhalten.

7.5.3 Graphenimplementierung

Graphen werden in der Praxis intern meist als verkettete Listen oder als boolesche Matrizen dargestellt. Wir hatten uns für die Testumgebung für die nachfolgerorientierten Listen als Darstellung entschieden.
Wir wollen hier nicht grundsätzlich einen Vergleich dieser beiden Ansätze vornehmen, aber kurz die für die folgende Argumentation wichtigsten Eigenschaften zusammenstellen.
Für die Speicherung von dünnen Graphen eignet sich eher die Listenimplementierung, bei dicken Graphen ist die Matrix-Variante speichersparender. Die Tiefensuche, ein in allen entwickelten Algorithmen verwendetes Verfahren, benötigt das Besuchen aller Nachfolger. Hierfür ist die (nachfolgerorientierte) Liste geradezu prädestiniert.
Allgemein sind die Zwischenergebnisse, die als Graphen gespeichert werden, schon klein, was die Liste als allgemeine Implementierung bevorzugt. Würden wir das Problem maximieren, also die transitive Hülle berechnen wollen, so hätte die Matrix-Implementierung Vorteile.

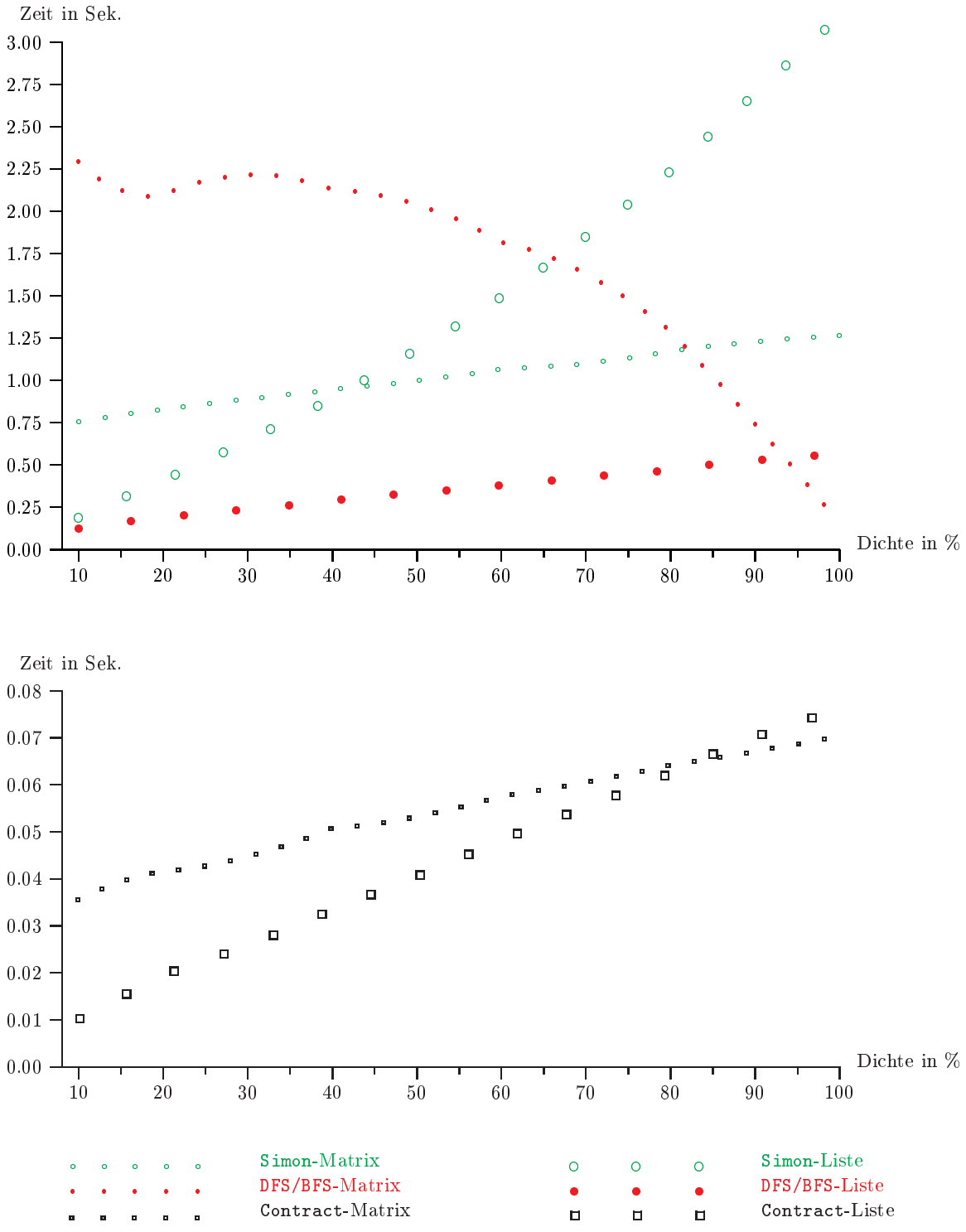


Abbildung 7.11: `DFS/BFS` vs. `Simon` vs. `Contract` — Matrix und Liste

Wir betrachten hier nur den Zeitaspekt, ein Unterschied in der Anzahl der Pfeile ist nicht vorhanden.
Dazu betrachten wir die Abbildung 7.11, in der die drei grundsätzlichen Verfahren jeweils in der Matrix- und in der Listen-Variante verglichen werden.
Allgemein ist zu erkennen, daß die Matrix-Variante bei großer Dichte Vorteile besitzt. Bei Simon tritt dieser Effekt schon bei 40% ein, da das Verfahren sehr speicherintensiv ist. Bei den beiden anderen ist dies erst bei rund 90% der Fall.
Der zeitliche Verlauf des Verfahrens `DFS/BFS` ist dabei sehr auffällig, da dort bei steigender Dichte geringere Laufzeiten zu beobachten sind. Die Begründung ist in der sinkenden Pfeilanzahl des Ergebnisses und in der langen Laufzeit der für die Matriximplementierung ungünstigen Tiefensuche zu suchen.

Nach diesen Betrachtungen ist klar, daß bei bekannter Dichte für jedes Verfahren die schnellere Graphimplementierung gewählt werden kann. Betrachten wir für jedes Verfahren jeweils das Minimum beider Implementierungen, so ergibt sich das grundsätzliche Verhältnis der Verfahren untereinander.

Diese Ergebnisse rechtfertigen den Einsatz der Listen-Implementierungen für den grundsätzlichen Vergleich, zeigen aber auch, daß bei bekannter Dichte noch zeitliche Verbesserungen möglich sind.

7.5.4 Hardware

Die entwickelten Verfahren wurden auch unter anderen Betriebssystemen (Windows, Linux), Speichergrößen und Prozessorleistungen getestet.
Dabei sind Unterschiede in der Anzahl der zurückgegebenen Pfeile natürlich nicht zu erkennen. Andererseits ergeben sich andere absolute Laufzeiten der Verfahren. So sind bei steigender Prozessorleistung bzw. Speichergröße schnellere Zeiten erzielt worden. Die Verwendung unterschiedlicher Betriebssysteme hatte dagegen kaum Auswirkungen auf die Ergebnisse.

Relativ gesehen sind die Testergebnisse nahezu identisch mit den bisher dargestellten. Die angesprochenen Abhängigkeiten bei der Speichergröße führen zu anderen Verhältnissen bei großer Knotenzahl und sind die auffälligsten Veränderungen.

7.5.5 Standardabweichung

Alle bisher dargestellten Abbildungen basierten auf einer Vielzahl von Durchläufen und sind das arithmetische Mittel. Es stellt sich die Frage, ob sowohl die Anzahl der Pfeile als auch die Laufzeit Mittelwerte von stark schwankenden oder von fast gleichen Werten sind. Mit Hilfe der Abbildungen 7.12 und 7.13 zeigen wir, daß die Abweichung vom Mittel gering ist.
Die Abbildungen zeigen die Anzahl der zurückgegebenen Pfeile von 50 Durchläufen jeweils bei den Dichten 10%, 30%, 50% und 70% für das Verfahren `DFS/BFS` in 7.12 und für `TransRed_Contract_white` in 7.13.
Es ist zu erkennen, daß bei steigender Dichte die Schwankungen um den Mittelwert geringer werden. Allerdings sind keine Ausreißer erkennbar. Die etwas breitere Streuung bei dünnen

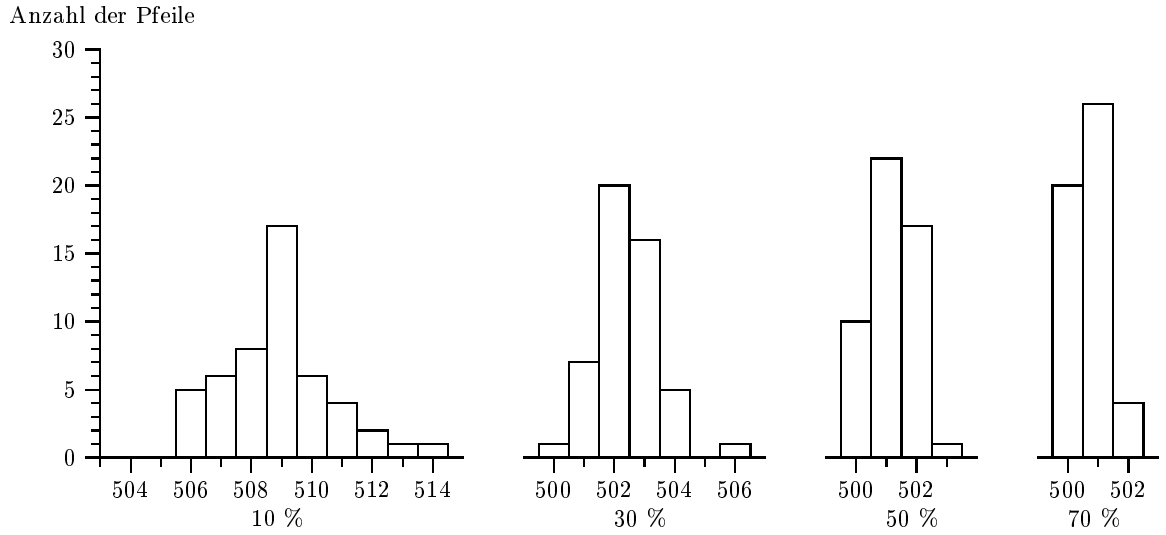


Abbildung 7.12: Standardabweichung `DFS/BFS` — Anzahl der Pfeile

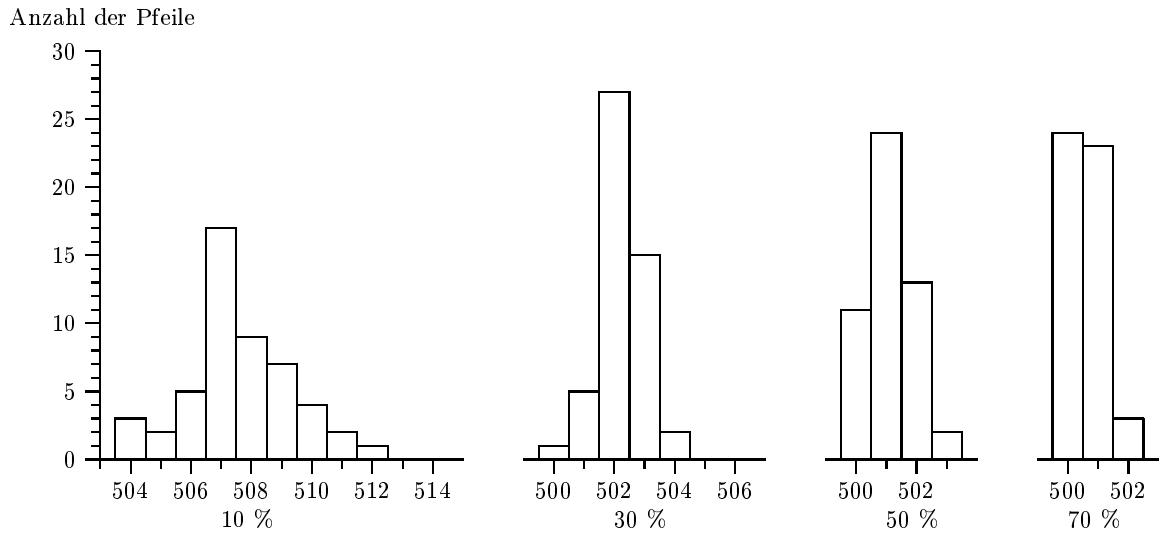


Abbildung 7.13: Standardabweichung `TransRed_Contract_white` — Anzahl der Pfeile

Graphen ist kein Zeichen von Ausreißern, sondern nur auf die unterschiedliche optimale Anzahl $\mathcal{OPT}(G)$ zurückzuführen.

Wie bei den Pfeilanzahlen ergibt sich auch beim Vergleich der Laufzeiten eine geringe Abweichung vom Mittelwert. Diese ist außerdem noch unabhängig von der Dichte.

Kapitel 8

Zusammenfassung

In dieser Arbeit wurden Algorithmen aus dem Themenbereich der minimalen Äquivalenzgraphen und der transitiven Reduktionen sowohl theoretisch entwickelt als auch praktisch umgesetzt und miteinander verglichen.

In den theoretischen Entwicklungen wurden drei unterschiedliche Ansätze für stark zusammenhängende Graphen gezeigt.
Der erste Ansatz, das Minimierungsverfahren, basierte auf einem generischen Programm für inklusionsminimale Probleme und berechnete eine transitive Reduktion in quadratischer Laufzeit.
Der zweite, der aus einer Arbeit von Simon stammt, wurde aufgrund eines Fehlers, der bei der Untersuchung entdeckt wurde, in einer korrigierten Fassung vorgestellt. Auch dieser Algorithmus benötigte quadratische Laufzeit zur Berechnung einer transitiven Reduktion.
Eine Arbeit von Khuller et. al. bildete die Grundlage des letzten Ansatzes zur Entwicklung eines approximativen Algorithmus, der Güten bis zu $\frac{\pi^2}{6}$ erreicht und in polynomieller Zeit implementierbar ist. Der dargestellte Spezialfall für $k = 3$ ist fast-linear mit einer Güte von 1,75.

Die Untersuchung der Arbeit von Simon zeigte, daß die Entwicklung theoretisch schneller Algorithmen auch Nachteile mit sich bringt. Da solche Ansätze meist sehr komplex sind, bergen sie oft die Gefahr, fehlerhaft zu sein, weil sie schwer nachvollziehbar bzw. nicht formal in einem Kalkül entwickelbar sind.

Es wird aber auch in Zukunft ein Ziel der theoretischen Informatik bleiben, ein lineares Verfahren zur Berechnung transitiver Reduktionen in stark zusammenhängenden Graphen zu entwickeln. Weiterhin ist sie an approximativen Algorithmen interessiert, die die Güten verbessern.

Aus den praktischen Betrachtungen wurde deutlich, daß sich nicht alle theoretischen Eigenschaften direkt in die Praxis übertragen lassen. Dies läßt sich hauptsächlich dadurch begründen, daß sich die theoretischen Laufzeit- und Güteabschätzungen immer auf den schlechtesten

Fall beziehen und daher kaum eine Aussagekraft für den praktischen Einsatz haben. Daher ist es wünschenswert und notwendig, daß die Anzahl der theoretischen Arbeiten zunimmt, in denen sich praktische Vergleiche mit anderen Ansätzen finden.

So zeigte der praktische Vergleich, daß das Verfahren von Simon, ob in korrigierter oder in falscher Originalversion, wesentlich langsamer als alle anderen Ansätze ist.
Das in dieser Arbeit vorgestellte Minimierungsverfahren bietet zusätzlich den Vorteil der Kürze und damit der Übersichtlichkeit, der formalen Verifikation und der schnellen Implementierbarkeit.
Theoretische Entwicklungen bieten oft Freiheiten für ihre Implementierung, die aber weder angesprochen noch umgesetzt werden. So lieferte eine direkte Umsetzung der theoretischen Arbeit von Khuller (`Contract`) eine schlechte Approximation eines minimalen Äquivalenzgraphen. Mit dem nur durch Änderung der Abarbeitungsreihenfolge entstandenen Verfahren `Contract_white` konnten erheblich bessere Güteeigenschaften gegenüber dem ursprünglichen erzielt werden, wobei die Laufzeit nur geringfügig verschlechtert wurde.
Die iterative Anwendung bei gleichzeitiger Änderung der zugrundeliegenden Ordnung wirkte sich positiv auf die Eigenschaften aus. So ist auch das erarbeitete Verfahren `Contract_iter` für den praktischen Einsatz zu empfehlen.
Auch durch die Kombination mehrerer Algorithmen konnten bessere praktische Werte erzielt werden. So wirkte sich das quadratische Verfahren der Nachminimierung der approximativen Ansätze (`TransRed_Contract_white`, `TransRed_Contract_iter`) aufgrund der vorgeschalteten Gradtests sowie der geringen Pfeilanzahl nach der Vorberechnung kaum negativ auf das Laufzeitverhalten aus. Die beiden Verfahren sind daher gut zur Berechnung einer transitiven Reduktion geeignet.

Die Tests zeigten aber auch, daß es kein optimales Programm für alle Situationen gibt. Sind aber wesentliche Eigenschaften der zu bearbeitenden Graphen bekannt oder spezielle Problemstellungen gegeben, so können wahlweise die entwickelten Verfahren `Contract`, `Contract_iter`, `Contract_white`, `TransRed_Contract_white` oder `TransRed_Contract_iter` empfohlen werden.

Anhang A

C-Implementierung

In diesem Kapitel wird der in den Kapiteln 4 bis 7 entwickelte Pseudocode in C-Code [19] umgesetzt.
Dabei sind die entwickelten Programme alle modular aufgebaut, d.h. sie basieren auf denselben klassenartigen grundlegenden Datentypen. Dies sichert die Vergleichbarkeit der Ansätze.
Dazu werden im ersten Abschnitt die grundlegenden Module (Graphen, Mengen, ...) vorgestellt. Dies geschieht, ähnlich einer Klassenbibliothek, so daß die zur Verfügung stehenden Operationen mit Spezifikation der Ein- und Ausgabe und der Funktionalität angegeben werden ohne aber ihre konkrete Implementierung anzugeben.
Anschließend werden dann die in dieser Arbeit vorgestellten Algorithmen umgesetzt.

Wir verweisen noch auf einige grundlegenden Implementierungsgrundlagen. Die Knotenmenge V wird mit den Zahlen $\{1, \ldots, |V|\}$ identifiziert und entspricht dem Datentyp `Node`. Dabei müssen wir beachten, daß in `C` alle Felder mit 0 beginnen. Wir allozieren deshalb stets $|V| + 1$ Elemente. Außerdem geschieht aus Effizienzgründen die Allozierung immer mit `calloc`, einer Speicheranforderung, die alle Elemente mit 0 initialisiert.
Analog zur Allozierung wird auch stets auf die Freigabe des verwendeten Speichers geachtet. Dazu haben alle grundlegenden Module neben der natürlich benötigten Initialisierungsfunktion eine Freigabefunktion.

Um die Übersichtlichkeit zu erhöhen, werden nur die Funktionen vorgestellt, die später auch benötigt werden. Es sollte also nicht verwundern, wenn ein Modul unvollständig wirkt.

Die C-Funktionen setzen meist direkt den Pseudocode um. Wir nennen sie daher ähnlich, die kleinen Änderungen sind durch die C-Syntax bedingt. Um die Lesbarkeit der Algorithmen zu erhöhen, bestehen diese meist aus entsprechenden Aufrufen anderer Funktionen und sind deshalb ähnlich kompakt wie der Pseudocode. Falls allerdings die Umsetzung stark von den Vorgaben abweicht, werden wir auf die Änderungen speziell eingehen.

A.1 Grundlegende Funktionen

Zuerst wollen wir die Graphen vorstellen. Es folgen eine Klasse zur Modellierung der standfor-Struktur aus 5.2.1, die Breiten- und Tiefensuche, Mengen, union-find-Strukturen und Pfeile. Auf konkrete Implementierungen wird hier nicht eingegangen. Diese können, wie im Beispiel der Graphen, auch ausgewechselt werden.

A.1.1 Graphen

Die meisten Funktionen sind selbsterklärend. Es sei aber besonders auf die Makros verwiesen. Sie stellen eine einfache Schnittstelle dar, um alle Knoten oder Pfeile abzuarbeiten.

```
/* Eingabe  : gerichteter Graph g                                    */
/* Ausgabe  : gerichteter Graph                                      */
/* Funktion : Transponiert einen gerichteten Graphen g               */
extern graph * transpose_graph ( graph *g );

/* Eingabe  : Dimension dim                                          */
/* Ausgabe  : gerichteter Graph                                      */
/* Funktion : Allokiert Speicher fuer gerichteten Graphen            */
/*            mit Dimension dim.                                     */
extern graph * new_graph       ( int dim );

/* Eingabe  : Knoten x, Knoten y, gerichteter Graph g                */
/* Ausgabe  : ---                                                    */
/* Funktion : Fuegt einen Pfeil von x nach y in Graph g ein. Falls   */
/*            dieser vorhanden ist, erfolgt keine Veraenderung.      */
extern void    arrow_insert    ( Node x, Node y, graph *g );

/* Eingabe  : Knoten x, Knoten y, gerichteter Graph g                */
/* Ausgabe  : ---                                                    */
/* Funktion : Loescht den Pfeil von x nach y in Graph g. Falls       */
/*            dieser nicht vorhanden ist, erfolgt keine Veraenderung.*/
extern void    arrow_delete    ( Node x, Node y, graph *g );

/* Eingabe  : Knoten x, Knoten y, gerichteter Graph g                */
/* Ausgabe  : Wahrheitswert                                          */
/* Funktion : Prueft, ob im Graph g ein Pfeil von x nach y existiert */
extern bool    is_arrow        ( Node x, Node y, graph *g );
```

```
/* Eingabe : gerichteter Graph g                                          */
/* Ausgabe : (gemaess Eingabe:) gerichteter/ungerichteter Graph           */
/* Funktion: Kopiert den Graphen g                                        */
extern graph * copy_graph      ( graph *g );

/* Eingabe : gerichteter Graph g                                          */
/* Ausgabe : ganze Zahl                                                   */
/* Funktion: Liefert die Dimension des Graphen g                          */
extern int     graph_dim       ( graph *g );

/* Eingabe : gerichteter Graph g                                          */
/* Ausgabe : ganze Zahl                                                   */
/* Funktion: Liefert die Anzahl der Pfeile                                */
extern int     numberofarrows  ( graph *g );

/* Eingabe : gerichteter Graph g                                          */
/* Ausgabe : ganze Zahl                                                   */
/* Funktion: Loescht Graph g aus dem Speicher. Liefert als                */
/*           Resultat immer 1.                                            */
extern int     free_graph      ( graph *g );

/* Eingabe : Graphen g, h                                                 */
/* Ausgabe : Vereinigungsgraph g u h                                      */
/* Funktion: Liefert die Vereinigung der Graphen g und h                  */
extern void    union_graph   ( graph *g, graph *h );

/* MAKROS */

/* Die Knoten werden nacheinander node zugeordnet                         */
forall_nodes( node, dim )

/* Allen Nachfolgern von start in graph wird end nacheinander zugewiesen  */
forall_adj_edges( start, end, graph )

/* Allen Nachfolgern von start in graph wird end nacheinander zugewiesen;
   dabei wird die Ordnung aus forall_adj_edges umgedreht                  */
forall_adj_edges_down( start, end, graph )

/* Allen Pfeilen in graph werden nacheinander (start, end) zugeordnet     */
forall_edges( start, end, graph )
```

A.1.2 Standfor-Struktur

Wir modellieren hier die in Abschnitt 5.2.1 benötigte Abbildung $standfor : E \to E$. Dazu benötigen wir neben der Initialisierung und der Freigabe die Abfrage und das Verändern der Abbildung.

```
/* Eingabe  : standfor-Graph g                                          */
/* Ausgabe  : gerichteter Graph vom Typ standfor                        */
/* Funktion : Kopiert den Graphen g in einen standfor-Graphen           */
extern graph_sf *to_graph_sf   ( graph *g );

/* Eingabe  : standfor-Graph g                                          */
/* Ausgabe  : ---                                                       */
/* Funktion : Loescht standfor-Graph g aus dem Speicher. Liefert als    */
/*            Resultat immer 1.                                         */
extern void     free_graph_sf  ( graph_sf *g );

/* Eingabe  : standfor-Graph g, Pfeil von start nach end                */
/* Ausgabe  : ---                                                       */
/* Funktion : Liefert in start_sf und end_sf den Pfeil fuer den         */
/*            (start, end) steht                                        */
extern void    get_start_end_sf( graph_sf *g_sf,
                                 Node      start,
                                 Node      end,
                                 Node     *start_sf,
                                 Node     *end_sf );

/* Eingabe  : standfor-Graph g, Pfeil von start nach end                */
/* Ausgabe  : ---                                                       */
/* Funktion : Aendert den Pfeil von (start, end) in (newstart, newend)  */
extern void    change_standfor ( Node      start,
                                 Node      end,
                                 Node      newstart,
                                 Node      newend,
                                 graph_sf *g );
```

A.1.3 Breiten- und Tiefensuche

In diesem Abschnitt werden Funktionen aus dem Themenbereich der Breiten- und Tiefensuche vorgestellt, unter anderem die Baumfunktionen, Klassifikationsfunktionen und spezielle Erreichbarkeitsfunktionen.

```
enum COLOUR {weiss, grau, schwarz};

/* Eingabe  : Graph g und Knoten x                                    */
/* Ausgabe  : Graph                                                   */
/* Funktion : DFS-Baum von g mit Wurzel x                             */
extern graph *  DFS_Tree( graph *g, Node x );

/* Eingabe  : Graphen g, h                                            */
/* Ausgabe  : Graph                                                   */
/* Funktion : BFS-Baum von g mit Wurzel x                             */
extern graph *  BFS_Tree( graph *g, Node x );

/* Eingabe  : Graph g und Knoten start und end, feld colour           */
/* Ausgabe  : Wahrheitswert                                           */
/* Funktion : Erreichbarkeit von start nach end in g mit Hilfe eines  */
/*            DFS-Durchlaufes ohne den Pfeil (start, end )            */
extern bool     dfs_reach( graph *g,
                           Node   actual_node,
                           int   *colour,
                           Node   to_found );

/* Eingabe  : Knoten start und feld ( z.B aus search( g ) )           */
/* Ausgabe  : Integer                                                 */
/* Funktion : DFS-Zahl des Knoten start laut feld                     */
extern int      dfs_zahl( Node start, felder feld );

/* Eingabe  : Knoten start und end sowie feld (z.B aus search( g ) )  */
/* Ausgabe  : Boolescher Wert                                         */
/* Funktion : Klassifikation des Pfeiles ( start, end ) laut feld     */
extern bool     is_treeedge( Node start, Node end, felder feld );

extern bool     is_backwardedge( Node start, Node end, felder feld );

extern bool     is_forwardedge( Node start, Node end, felder feld );

extern bool     is_crossedge( Node start, Node end, felder feld );
```

```
/* Eingabe  : Knoten start und end, sowie feld ( z.B aus search( g ) )     */
/* Ausgabe  : Knoten                                                       */
/* Funktion : Naechster gemeinsamer Vorfahren der Knoten start und end
              laut feld ( also gemaess dem DFS-Baum )                      */
extern Node     lca( Node start, Node end, felder feld );

/* Eingabe  : Graph g                                                      */
/* Ausgabe  : Felder in feld                                               */
/* Funktion : DFS_Durchlauf fuer alle Knoten und Abspeichern der           */
/*            Informationen in feld                                        */
extern felder   search( graph *g );
```

A.1.4 Union-find-Struktur

Es sei hier nur erwähnt, daß es keine direkte Umsetzung der `MAKE-SET`-Funktionalität gibt. Dafür gibt es eine Initialisierung aller Elemente mit Hilfe von `init_union_find`. Um Verwechslungen mit der internen `C`-Funktion `union` zu vermeiden, nennen wir sie `manipulate`.

```
/* Eingabe  : Dimension dim                                                */
/* Ausgabe  : initialisierte Union-Find-Struktur mit dim-vielen Elementen  */
/* Funktion : Initialisierung                                              */
extern union_find init_union_find ( int dim );

/* Eingabe  : Union-Find-Struktur C, Knoten x und y                        */
/* Ausgabe  : Union-Find-Struktur, in der die beiden Komponenten,          */
/*            in der x bzw. y lagen, nach der Groesse vereinigt wurden     */
/* Funktion : Verschmelzen                                                 */
extern union_find manipulate ( union_find C, Node x, Node y );

/* Eingabe  : Union-Find-Struktur C, Knoten x                              */
/* Ausgabe  : Repraesentantenknoten fuer die Komponente, in der x liegt    */
/* Funktion : Repraesentanten finden                                       */
extern Node       find ( union_find C, Node x);

/* Eingabe  : Union-Find-Struktur uf, die augeloest werden soll            */
/* Ausgabe  : 1, falls alles geklappt hat; 0 sonst                         */
/* Funktion : Speicherfreigabe                                             */
extern int        free_uf (union_find uf );
```

A.1.5 Mengen

Dieser Abschnitt führt den Typ Menge ein. Wir stellen hier nur die benötigten Funktionen vor.

```
/* Eingabe  : Dimension                                                  */
/* Ausgabe  : leere Menge mit maximal dim Elementen                      */
/* Funktion : Initialisierung                                            */
extern Set      empty_set     ( int dim );

/* Eingabe  : Element x, Menge M                                         */
/* Ausgabe  : Wahrheitswert, ob Operation erfolgreich ausgefuehrt wurde  */
/* Funktion : Einfuegen eines Elementes                                  */
extern bool     insertElement ( Node x, Set *M );

/* Eingabe  : Element x, Menge M                                         */
/* Ausgabe  : Wahrheitswert, ob Element x in der Menge M vorhanden ist   */
/* Funktion : Einfuegen eines Elementes                                  */
extern bool     isElement     ( Node x, Set M );

/* Eingabe  : Menge M                                                    */
/* Ausgabe  : Wahrheitswert, ob Operation erfolgreich ausgefuehrt wurde  */
/* Funktion : Speicherfreigabe                                           */
extern bool     free_set      ( Set S );
```

A.1.6 Pfeile

Als letztes benötigen wir einen Typ, um Pfeile zu modellieren. Es werden Funktionen zum Setzen und Abfragen einer Menge von Pfeilen vorgestellt sowie die im Abschnitt 6 benötigten Konstanten `nil` und `current`.

```
typedef struct Edge
{
  Node begin;
  Node end;
} edge;

#define current NUM_MAX
#define nil     0

void get_edge( Node *u, Node *w, edge *to_get, Node v )
{
  *u = to_get[ v ].begin;
  *w = to_get[ v ].end;
}

void set_edge( Node u, Node w, edge *to_set, Node v )
{
  to_set[ v ].begin = u;
  to_set[ v ].end = w;
}
```

A.2 Minimierungsalgorithmus

Im Kapitel 7.1 benötigen wir insgesamt 4 Kombinationen von Baumalgorithmen. Es wird hier nur eine exemplarisch vorgestellt.

```
graph * DFSBFS( graph *R )
{
  graph *S, *RT, *help;
  Node root = 1;

  S = DFS_Tree( R, root );
  RT = transpose_graph( R );
  help = BFS_Tree( RT, root );
  union_graph( S, help );

  free_graph( help );
  free_graph( RT );

  return S;
}
```

Es bleibt noch die **while**-Schleife mitsamt dem Prädikat $\mathcal{Q}$ zu implementieren. Beachte, daß wir auf die Variable B verzichten können; dies ermöglicht das Makro `forall_edges`.
Außerdem sei auf die spezielle Funktionalität von `dfs_reach` verwiesen. Sie berechnet die Erreichbarkeit zwischen zwei Knoten, ohne den direkten Weg zu berücksichtigen.

```
void Min_while( graph *A )
{
  Node start, end;

  forall_edges( start, end, A )
    {
      if( dfs_reach( A, start, end ) )
      {
        arrow_delete( start, end, A );
      }
    }
}
```

Aus den Betrachtungen in 7.1 ergibt sich folgende optimierte Version, die auch für die Tests eingesetzt wurde. Dazu berechnen wir vorab den Ausgangs- und Eingangsgrad für jeden Knoten in den Variablen `succs` und `preds`.

```
void Min_while( graph *A )
{
  Node start, end;
  Node *preds, *succs;

  preds = ( Node* )calloc( graph_dim( A ) + 1, sizeof( Node ) );
  succs = ( Node* )calloc( graph_dim( A ) + 1, sizeof( Node ) );

  forall_edges( start, end, A )
    {
      preds[ end ]++;
      succs[ start ]++;
    }

  forall_edges( start, end, A )
    {
      if( ( succs[ start ] > 1 ) && ( preds[ end ] > 1 ) )
      {
        if( dfs_reach( A, start, end ) )
        {
          arrow_delete( start, end, A );
          succs[ start ]--;
          preds[ end ]--;
    } } }

  free( succs );
  free( preds );
}
```

Aus den erarbeiteten Funktionen ergibt sich der folgende grundsätzliche Aufbau, der hier exemplarisch mit einer Vorberechnung dargestellt wird:

```
graph * TransRed_DFS_BFS( graph *R )
{
  graph *S;

  S = DFSBFS( R );
  Min_while( S );

  return S;
}
```

A.3 Algorithmus von Simon

Schon die Entwicklung des Algorithmus von Simon hat gezeigt, daß der entstehende Code sehr lang und kompliziert ist. Wir werden deshalb nur den erarbeiteten Pseudocode direkt umsetzen und auf unübersichtliche Optimierungen verzichten. Für die Tests wurden effizienzsteigernde Maßnahmen natürlich berücksichtigt.
Dabei muß klar sein, daß bei der theoretischen Entwicklung die Knoten mit ihren DFS-Zahlen identifiziert wurden und so auch die entsprechenden Minima gebildet wurden.

A.3.1 A

Es werden zwei Graphen zurückgegeben. Der DFS-Baum T muß indirekt zurückgegeben werden.

```
graph* AlgA( graph *g, graph **T, Node root )
{
  graph *C, *B, *res;
  graph_sf *g_sf;
  felder feld;
  int dim;
  Node *backpoint;
  Node start;
  Node end;
  Node w;
  Node start_sf;
  Node end_sf;

  dim = graph_dim( g );

  backpoint = ( Node* )malloc( ( dim + 1 ) * sizeof( Node ) );

  feld = search( g );

  g_sf = to_graph_sf( g );

  C = new_graph( dim );
  B = new_graph( dim );
  *T = new_graph( dim );
  res = new_graph( dim );

  forall_nodes( start, dim )
    {
      backpoint[ start ] = start;
    }
```

```
forall_edges( start, end, g )
  {
    /* Baumpfeile */
    if( is_treeedge( start, end, feld ) )
    {
      arrow_insert( start, end, *T );
      arrow_insert( start, end, res );
    }
    /* Rueckwaertspfeile */
    else if( is_backwardedge( start, end, feld ) )
    {
      arrow_insert( start, end, B );
      arrow_insert( start, end, res );
      if( dfs_zahl( end, feld ) < dfs_zahl( backpoint[ start ], feld ) )
      {
        backpoint[ start ] = end;
      }
    }
    /* Vorwaertspfeile */
    else if( is_forwardedge( start, end, feld ) )
    {
      /* gleich entfernen */
      ;
    }
    /* Querpfeile */
    else
    {
      arrow_insert( start, end, C );
      arrow_insert( start, end, res );
    }
  }

/* forall e \in B */
forall_edges( start, end, B )
  {
    if( end != backpoint[ start ] )
    {
      arrow_delete( start, end, res );
    }
  }
free_graph( B );
```

```
  /* forall e \in C */
  forall_edges( start, end, C )
    {
      w = lca( start, end, feld );
      if( dfs_zahl( w, feld ) < dfs_zahl( backpoint[ start ], feld ) )
      {
        arrow_insert( start, w, res);
        arrow_delete( start, backpoint[ start ], res );
        change_standfor( start, w, start, end, g_sf );
        backpoint[ start ] = w;
      }
      arrow_delete( start, end, res );
    }
  free_graph( C );

  /* R-test */
  R_test( root, res, *T, g_sf, backpoint, feld );
  free_felder( feld );
  free( backpoint );

  /* standfor */
  forall_edges( start, end, res )
    {
      get_start_end_sf( g_sf, start, end, &start_sf, &end_sf );
      if( ( start != start_sf ) || ( end != end_sf ) )
      {
        arrow_delete( start, end, res );
        arrow_insert( start_sf, end_sf, res );
      }
    }

  free_graph_sf( g_sf );

  return res;
}
```

A.3.2 R-test

Die Funktion `ReduceBackward` ist direkt eingebaut.

```
void R_test( Node v, graph *res, graph *T, graph_sf *g_sf,
             Node *backpoint, felder feld )
{
  Node z, end, start_sf, end_sf;
  bool leaf = true, first = true;

  forall_adj_edges( v, end, T )
    {
      leaf = false;
      R_test( end, res, T, g_sf, backpoint, feld );
    }
  if( !leaf )
  {
    forall_adj_edges( v, end, T )
      {
        if( first )
        {
          z = backpoint[ end ];
          first = false;
        }
        else
        {
          if( dfs_zahl( backpoint[ end ], feld ) < dfs_zahl( z, feld ) )
          {
            z = backpoint[ end ];
      } } }
    if( v != z )
    {
      if(dfs_zahl( backpoint[ v ], feld ) < dfs_zahl( backpoint[ z ], feld ))
      {
        arrow_delete( z, backpoint[ z ], res );
        arrow_insert( z, backpoint[ v ], res );
        get_start_end_sf( g_sf, v, backpoint[ v ], &start_sf, &end_sf );
        change_standfor( z, backpoint[ v ], start_sf, end_sf, g_sf );
        backpoint[ z ] = backpoint[ v ];
      }
      arrow_delete( v, backpoint[ v ], res );
      backpoint[ v ] = z;
} } }
```

A.3.3 `ReduceTreeEdges`

Es werden zwei Graphen zurückgegeben: einer direkt, der andere indirekt in der Variablen `res`.
Die Funktion `dfs_reach` prüft die Erreichbarkeit zwischen zwei Knoten ohne den direkten Pfeil. Deshalb ist die Umsetzung der ersten **if**-Schleife etwas komplizierter.

```
graph * ReduceTreeEdges( graph *g, graph *T1, graph *T2,
                         Node root, graph **res )
{
  Node start;
  Node end;
  graph *nonredsinktree;

  nonredsinktree = copy_graph( g );
  *res = copy_graph( g );

  forall_edges( start, end, T1 )
    {
      if( is_arrow( start, end, T2 ) )
      {
        arrow_delete( start, end, nonredsinktree );
        if( !is_arrow( start, root, nonredsinktree ) &&
            !dfs_reach( nonredsinktree, start, root ) )
        {
          arrow_insert( start, end, nonredsinktree );
        }

        if( dfs_reach( *h, start, end ) )
        {
          arrow_delete( start, end, *h );
        }
      }
    }

  return nonredsinktree;
}
```

A.3.4 Theo2

```
graph *Theo2( graph *g, graph **nonredsinktree, Node root )
{
  graph *res;
  graph *Ts;
  graph *T;
  graph *gstrich;
  graph *Gr;
  graph *GrT;

  Gr = transpose_graph( g );
  GrT = AlgA( Gr, &T, root );

  Ts = transpose_graph( T );
  free_graph( T );

  gstrich = transpose_graph( GrT );

  g2 = AlgA( gstrich, &T, root );

  *nonredsinktree = ReduceTreeEdges( g2, T, Ts, *res );

  free_graph( GrT );
  free_graph( gstrich );
  free_graph( Gr );
  free_graph( Ts );
  free_graph( T );
  free_graph( g2 );

  return res;
}
```

A.3.5 TransRed-Simon

```
graph * TransRed_Simon( graph *g )
{
  graph *res1;
  graph *res2;
  graph *res1T;
  graph *nonredtree1;
  graph *nonredtree2;
  graph *nonredtree2T;
  Node root = 1;

  res1 = Theo2( g, &nonredtree1, root );

  res1T = transpose_graph( res1 );

  res2 = Theo2( res1T, &nonredtree2T, root );

  nonredtree2 = transpose_graph( nonredtree2T );

  union_graph( nonredtree1, nonredtree2 );

  free_graph( nonredtree2T );
  free_graph( res1T );
  free_graph( res2 );
  free_graph( res1 );
  free_graph( nonredtree2 );

  return nonredtree1;
}
```

A.4 Approximativer Algorithmus von Khuller et al.

Wir wollen in diesem Abschnitt die Algorithmen aus dem Kapitel 6 sowie die Erweiterungen aus dem 7. Abschnitt vorstellen. Dabei ist vor allem zu beachten, alle Variablen an die Prozeduren zu übergeben sowie die Ergebnisse zurückzubekommen.

A.4.1 CONTRACT-CYCLE

```
void contract_cycle( Node w, edge *to_active, edge *from_root,
                     edge *to_root, union_find uf, graph *S )
{
  Node c, p;
  Node f_start, f_end;
  Node t_start, t_end;
  Node a_start, a_end;
  Node start, end;

  get_edge( &start, &end, to_active, find( uf, w ) );
  while( ( start != current ) || ( end != current ) )
  {
    if( ( start == nil ) && ( end == nil ) )
    {
      get_edge( &c, &p, to_root, find( uf, w ) );
      get_edge( &a_start, &a_end, to_active, find( uf, p ) );
      arrow_insert( c, p, S );
    }
    else
    {
      get_edge( &p, &c, to_active, find( uf, w ) );
      get_edge( &a_start, &a_end, to_active, find( uf, c ) );
      arrow_insert( p, c, S );
    }
    get_edge( &f_start, &f_end, from_root, find( uf, p ) );
    get_edge( &t_start, &t_end, to_root, find( uf, p ) );
    uf = manipulate( uf, p, c );
    set_edge( a_start, a_end, to_active, find( uf, w ) );
    set_edge( f_start, f_end, from_root, find( uf, w ) );
    set_edge( t_start, t_end, to_root, find( uf, w ) );
    get_edge( &start, &end, to_active, find( uf, w ) );
  }
}
```

A.4.2 DFS-CONTRACT

```
void dfs_contract( graph *g, graph *S, Node u, Node *colour, edge *to_active,
                   edge *from_root, edge *to_root, union_find uf )
{
  Node w, x, y;

  colour[ u ] = grau;
  set_edge( current, current, to_active, find( uf, u ) );

  forall_adj_edges( u, w, g  )
    {
      if( colour[ w ] == weiss )
      {
        set_edge( u, w, from_root, find( uf, w ) );
        set_edge( u, w, to_active, find( uf, u ) );
        dfs_contract( g, S, w, colour, to_active, from_root, to_root, uf );
        set_edge( current, current, to_active, find( uf, u ) );
      }
      else
      {
        if( find( uf, u ) != find( uf, w ) )
        {
          get_edge( &x, &y, from_root, find( uf, u ) );
          if( find( uf, x ) == find( uf, w ) )
          {
            set_edge( u, w, to_root, find( uf, u ) );
          }
          else
          {
            get_edge( &x, &y, from_root, find( uf, w ) );
            if( find( uf, x ) != find( uf, u ) )
            {
              contract_cycle( w, to_active, from_root, to_root, uf, S );
              arrow_insert( u, w, S );
    } } } } }

  colour[ u ] = schwarz;
  set_edge( nil, nil, to_active, find( uf, u ) );
}
```

A.4.3 CONTRACT-CYCLES$_3$

```
graph * contract_cycles_3( graph *g )
{
  int dim;
  graph * S;
  union_find uf;
  Node * colour;
  edge * to_active;
  edge * to_root;
  edge * from_root;
  Node root = 1;

  dim = graph_dim( g );
  uf = init_union_find( dim );
  S = new_graph( dim );

  colour = (Node *)calloc( dim + 1, sizeof( Node ) );
  to_active = (edge *)calloc( dim + 1, sizeof( edge ) );
  to_root = (edge *)calloc( dim + 1, sizeof( edge ) );
  from_root = (edge *)calloc( dim + 1, sizeof( edge ) );

  dfs_contract( g, S, root, colour, to_active, from_root, to_root, uf );

  add_2_cycles( S, g, uf, from_root, to_root, root );

  free( colour );
  free( from_root );
  free( to_root );
  free( to_active );
  free_uf( uf );

  return S;
}
```

A.4.4 ADD-2-CYCLES

Eine Effizienzsteigerung kann dadurch erreicht werden, daß Pfeile wie in 6.2.1 beschrieben nicht doppelt eingefügt werden. Daher dient die Menge `T` dazu, sich die Vertreterknoten zu merken.

```
void add_2_cycles( graph *S, graph *g, union_find uf,
                   edge *from_root, edge *to_root, Node root )
{
  int dim;
  Node i;
  Node a, b;
  Node findNode;
  Set T;

  dim = graph_dim( g );

  T = empty_set( dim );

  insertElement( find( uf, root ), &T );

  forall_nodes( i, dim )
    {
      findNode = find( uf, i );
      if( !isElement( findNode, T ) )
      {
        insertElement( findNode, &T );
        get_edge( &a, &b, from_root, findNode );
        arrow_insert( a, b, S );
        get_edge( &a, &b, to_root, findNode );
        arrow_insert( a, b, S );
      }
    }
  free_set( T );
}
```

A.4.5 DFS-CONTRACT-ORDUNG

Wir betrachten die Prozedur `dfs_contract` und ihre Abarbeitungsreihenfolge der Pfeile als die „normale“ Ordnung und stellen hier die dazu entgegengesetzte Ordnung vor.

```
void dfs_contract_down( graph *g, graph *S, Node u, Node *colour,
                        edge *to_active, edge *from_root, edge *to_root,
                        union_find uf )
{
  Node w, x, y;

  colour[ u ] = grau;
  set_edge( current, current, to_active, find( uf, u ) );
  forall_adj_edges_down( u, w, g )
    {
      if( colour[ w ] == weiss )
      {
        set_edge( u, w, from_root, find( uf, w ) );
        set_edge( u, w, to_active, find( uf, u ) );
        dfs_contract_down( g, S, w, colour, to_active,
                           from_root, to_root, uf );
        set_edge( current, current, to_active, find( uf, u ) );
      }
      else
      {
        if( find( uf, u ) != find( uf, w ) )
        {
          get_edge( &x, &y, from_root, find( uf, u ) );
          if( find( uf, x ) == find( uf, w ) )
          {
            set_edge( u, w, to_root, find( uf, u ) );
          }
          else
          {
            get_edge( &x, &y, from_root, find( uf, w ) );
            if( find( uf, x ) != find( uf, u ) )
            {
              contract_cycle( w, to_active, from_root, to_root, uf, S );
              arrow_insert( u, w, S );
    } } } } }

  colour[ u ] = schwarz;
  set_edge( nil, nil, to_active, find( uf, u ) );
}
```

A.4.6 CONTRACT-CYCLES$_3$-ORDNUNG

Diese Funktion wählt nur zwischen den Funktionen `dfs_contract` und `dfs_contract_down` mit den verschiedenen Ordnungen aus.

```
graph * contract_cycles_3_ord( graph *g, bool ord )
{
  int dim;
  graph * S;
  union_find uf;
  Node * colour;
  edge * to_active;
  edge * to_root;
  edge * from_root;
  Node root = 1;

  dim = graph_dim( g );
  uf = init_union_find( dim );
  S = new_graph( dim );

  colour = (Node *)calloc( dim + 1, sizeof( Node ) );
  to_active = (edge *)calloc( dim + 1, sizeof( edge ) );
  to_root = (edge *)calloc( dim + 1, sizeof( edge ) );
  from_root = (edge *)calloc( dim + 1, sizeof( edge ) );

  if( ord )
    dfs_contract( g, S, root, colour, to_active, from_root, to_root, uf );
  else
    dfs_contract_down( g, S, root, colour, to_active,
                       from_root, to_root, uf );

  add_2_cycles( S, g, uf, from_root, to_root, root );

  free( colour );
  free( from_root );
  free( to_root );
  free( to_active );
  free_uf( uf );

  return S;
}
```

A.4.7 CONTRACT-CYCLES$_3$-ITER

Anstelle des Vergleiches $g = h$ testen wir aus Effizienzgründen die Kardinalität der beiden Graphen.

```
graph * contract_cycles_3_iter( graph *g )
{
  graph * res;
  graph * help;
  int noa_old;
  int noa_act;
  bool ord = false;

  res = contract_cycles_3_ord( g, ord );
  noa_act = numberofarrows( res );

  do
  {
    noa_old = noa_act;
    ord = !ord;
    help = contract_cycles_3_ord( res, ord );
    noa_act = numberofarrows( help );
    if( noa_old > noa_act )
    {
      free_graph( res );
      res = help;
      help = NULL;
    }
  }
  while( noa_old > noa_act );

  free_graph( help );

  return res;
}
```

A.4.8 DFS-CONTRACT-WHITE

Wir teilen das Besuchen der Pfeile auf. Die erste Schleife besucht nur weiße Knoten, die zweite Schleife die restlichen. Eine direkte Umsetzung des Pseudocodes führt zu einer speicherintensiven Variante, da eine Speicherung der besuchten Pfeile notwendig ist. Bei der vorgestellten Variante müssen wir nur sicherstellen, daß durch das erneute Besuchen eines Pfeiles keine Veränderungen der Variablen vorgenommen wird.
Eine entsprechende Einbettung in die übergeordnete Funktion `contract_cycles_3_white` ist leicht und wird hier nicht dargestellt.

```
void dfs_contract_white( graph      *g,
                         graph      *S,
                         Node        u,
                         Node       *colour,
                         edge       *to_active,
                         edge       *from_root,
                         edge       *to_root,
                         union_find  uf )
{
  Node w;
  Node x;
  Node y;


  colour[ u ] = grau;
  set_edge( current, current, to_active, find( uf, u ) );

  forall_adj_edges( u, w, g )
    {
      if( colour[ w ] == weiss )
      {
        set_edge( u, w, from_root, find( uf, w ) );
        set_edge( u, w, to_active, find( uf, u ) );
        dfs_contract_white( g, S, w, colour, to_active,
                            from_root, to_root, uf );
        set_edge( current, current, to_active, find( uf, u ) );
      }
    }
```

```
  forall_adj_edges( u, w, g )
    {
      if( find( uf, u ) != find( uf, w ) )
      {
        get_edge( &x, &y, from_root, find( uf, u ) );
        if( find( uf, x ) == find( uf, w ) )
        {
          set_edge( u, w, to_root, find( uf, u ) );
        }
        else
        {
          get_edge( &x, &y, from_root, find( uf, w ) );
          if( find( uf, x ) != find( uf, u ) )
          {
            contract_cycle( w, to_active, from_root, to_root, uf, S );
            arrow_insert( u, w, S );
          }
        }
      }
    }

  colour[ u ] = schwarz;
  set_edge( nil, nil, to_active, find( uf, u ) );
}
```

A.4.9 TransRed-CONTRACT-CYCLES$_3$-WHITE

Die Berechnung einer transitiven Reduktion auf der Basis des approximativen Verfahrens haben wir in Kapitel 7.4 mit Hilfe einer Nachminimierung mit dem Verfahren aus Kapitel 4 umgesetzt. Diese wird an einem Beispiel vorgestellt, die anderen Verfahren können entsprechend implementiert werden.

```
graph * TransRed_contract_white( graph *R )
{
  graph *A;

  A = contract_cycles_3_white( R );
  Min_while( A );

  return A;
}
```

Anhang B

Relationale Implementierung

In diesem Kapitel werden relationale Implementierungen einiger Verfahren dieser Arbeit vorgestellt. Dazu wurde das an der Christian-Albrechts-Universität zu Kiel entwickelte System RelView verwendet. Zur Bedienung und auf implementierungstechnische Einzelheiten sei auf [25, 3, 4, 6] verwiesen.

B.1 Grundlagen

B.1.1 Breitensuche

```
BFS(R,s)
  DECL a, p, x, y, w, z, T
  BEG
    x = s;
    y = R^*x & -x;
    T = x*y^;
    WHILE -empty(y) DO
      x = x | y;
      w = R^*x & -x;
      z = O(x);
      WHILE -eq(w,z) DO
        p = point(y);
        y = y & -p;
        a = R^*p & -(x | z);
        T = T | p*a^;
        z = z | a
      OD;
      y = R^*x & -x
    OD
    RETURN T
END.
```

B.1.2 Tiefensuche

```
DFS(R,s)
  DECL p, q, w, b, g, T
  BEG
    T = O(R);
    b = On1(R);
    g = s;
    p = s;
    WHILE -empty(p) DO
      w = R^*p & -(b|g);
      IF empty(w) THEN
        b = b | p;
        g = g & -p;
        p = T*p
      ELSE
        q = point(w);
        g = g | q;
        T = T | p*q^;
        p = q
      FI
    OD

    RETURN T
END.
```

B.2 Minimierungsalgorithmus

Wir stellen hier nur eine mögliche Variante vor. Die anderen Vorberechnungen können leicht durch Ersetzen der Aufrufe `DFS` und `BFS` erzeugt werden.

```
TransRed_DFS_BFS(R)
  DECL A, B, edge, p
  BEG
    p = point(Ln1(R));
    A = DFS(R,p) | (BFS(R^,p))^;
    B = A;
    WHILE -empty(B) DO
      edge = atom(B);
      IF incl(L(R),rtc(A & -edge)) THEN
        A = A & -edge;
        B = B & -edge
      ELSE
        B = B & -edge
      FI
    OD

    RETURN A
END.
```

B.3 Approximativer Algorithmus von Khuller et al.

In diesem Abschnitt wird eine relationale Implementierung des approximativen Ansatzes vorgestellt. Die Entwicklung des relationalen Programms auf der Basis eines nicht rekursiven Tiefensuchdurchlaufes war dabei der Ideengeber für die Variante, zuerst nur die unbesuchten Knoten abzuarbeiten.

B.3.1 Union-Find-Struktur

```
InitUF(R) = I(R).

findUF(p,UF) = UF^*p.

unionUF(p,q,UF)
  DECL fp, fq
  BEG
    fp = findUF(p,UF);
    fq = findUF(q,UF)

    RETURN (UF & -( (UF*fp)*fp^ ) ) | (UF*fp)*fq^
END.
```

B.3.2 CONTRACT-CYCLES$_3$

```
setedges(R, x, y) = (R & -(x*L1n(R))) | x*y^.

Khuller(R)
  DECL a, b, b1, p, q, b, fp, fq, fr, fx, fy, fstart, fend, g, r, tstart,
       tend, toactive, w, x, y, A, ArcSet, E, S, T, ToRootStart, ToRootEnd,
       FromRootStart, FromRootEnd, ToActiveStart, ToActiveEnd, UF
  BEG
    r = point(Ln1(R));
    ArcSet = R;
    S = 0(R);
    T = 0(R);
    UF = InitUF(R);
    ToRootStart   = 0(R);
    ToRootEnd     = 0(R);
    FromRootStart = 0(R);
    FromRootEnd   = 0(R);
    ToActiveStart = 0(R);
    ToActiveEnd   = 0(R);
    toactive = r;
    b = 0(r);
```

```
g = r;
p = r;
WHILE -empty(ArcSet) DO
  w = R^*p & -(b|g);
  IF empty(w) THEN
    A = p*L1n(R) & ArcSet;
    WHILE -empty(A) DO
      E = atom(A);
      q = ran(E);
      A = A & -E;
      ArcSet = ArcSet & -E;
      fp = findUF(p, UF);
      toactive = fp;
      fq = findUF(q, UF);
      IF -eq(fp, fq) THEN
        x =  FromRootStart^*fp;
        IF eq(findUF(x, UF), fq) THEN
          ToRootStart = setedges(ToRootStart, fp, p);
          ToRootEnd   = setedges(ToRootEnd, fp, q)
        ELSE
          x = FromRootStart^*fq;
          IF -eq(findUF(x, UF), fp) THEN
            WHILE -eq(toactive, fq) DO
              fq = findUF(q, UF);
              IF empty(ToActiveStart^*fq) THEN
                x = ToRootStart^*fq;
                y = ToRootEnd^*fq;
                fy = findUF(y, UF);
                a = ToActiveStart^*fy;
                b1 = ToActiveEnd^*fy;
                S  = S | x*y^
              ELSE
                y  = ToActiveStart^*fq;
                x  = ToActiveEnd^*fq;
                fx = findUF(x, UF);
                a  = ToActiveStart^*fx;
                b1 = ToActiveEnd^*fx;
                S  = S | y*x^
              FI;
              fy = findUF(y, UF);
              fstart = FromRootStart^*fy;
              fend   = FromRootEnd^*fy;
```

```
            tstart = ToRootStart^*fy;
            tend   = ToRootEnd^*fy;
            UF = unionUF(x, y, UF);
            toactive = findUF(p, UF);
            fq = findUF(q, UF);
            ToActiveStart = setedges(ToActiveStart, fq, a);
            ToActiveEnd   = setedges(ToActiveEnd, fq, b1);
            FromRootStart = setedges(FromRootStart, fq, fstart);
            FromRootEnd   = setedges(FromRootEnd, fq, fend);
            ToRootStart   = setedges(ToRootStart, fq, tstart);
            ToRootEnd     = setedges(ToRootEnd, fq, tend)
          OD;
          S = S | p*q^
        FI
      FI
    FI
  OD;

  ToActiveStart = ToActiveStart & -(findUF(p, UF)*L1n(R));
  b = b | p;
  g = g & -p;
  p = T*p
ELSE
  q = point(w);
  fp = findUF(p, UF);
  fq = findUF(q, UF);
  ToActiveStart = setedges(ToActiveStart, fp, p);
  ToActiveEnd   = setedges(ToActiveEnd, fp, q);
  FromRootStart = setedges(FromRootStart, fq, p);
  FromRootEnd   = setedges(FromRootEnd, fq, q);
  g = g | q;
  T = T | p*q^;
  ArcSet = ArcSet & -(p*q^);
  toactive = fq;
  p = q
FI
OD;

x = Ln1(R);
fr = findUF(r, UF);
WHILE -empty(x) DO
  p = point(x);
  x = x & -p;
```

```
      fp = findUF(p, UF);
      IF -eq(fp, fr) THEN
        a  = ToRootStart^*fp;
        b1 = ToRootEnd^*fp;
        S  = S | a*b1^;
        a  = FromRootStart^*fp;
        b1 = FromRootEnd^*fp;
        S  = S | a*b1^
      FI
    OD

    RETURN S
END.
```

Literaturverzeichnis

[1] Alfred V. Aho, M. R. Garey und J. D. Ullman: *The transitive reduction of a directed graph.* SIAM Journal on Computing, 1(2):131–137, Juni 1972.

[2] Alfred V. Aho, John E. Hopcroft und J. D. Ullman: *The design and analysis of computer algorithms.* Addison-Wesley, Reading, 1974.

[3] Ralf Behnke, Rudolf Berghammer, Erich Meyer und Peter Schneider: RelView — *A system for calculating with relations and relational programming.* In: Egidio Astesiano (Herausgeber): *Fundamental approaches to software engineering (First international conference, FASE '98, Held as part of the joint european conferences on theory and practice of software, EATPS '98, Lissabon, Portugal, 28. März – 4. April 1998)*, Band 1382 der Reihe *Lecture Notes in Computer Science*, Seiten 318–321. Springer, Berlin, 1998.

[4] Ralf Behnke, Rudolf Berghammer und Peter Schneider: *Machine support of relational computations: The Kiel* RelView *System.* Technischer Bericht 9711, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 1997.

[5] Rudolf Berghammer: *Ein generisches Programm für inklusionsminimale Teilmengen mit einer graphentheoretischen Anwendung.* In: Wolfgang Goerigk (Herausgeber): *Programmiersprachen und Rechnerkonzepte, Bericht 2007*, Seiten 141–150, Bad Honnef, 8.–10. Mai 2000. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel.

[6] Rudolf Berghammer und Thorsten Hoffmann: *Relational depth-first search with applications.* In: J. Desharnais (Herausgeber): *Relational methods in computer science (Proceedings of 5th international seminar on relational methods in computer science)*, Seiten 11–20, Quebec, 9.–14. Januar 2000. Dept. d'Informatique, Laval University.

[7] I. N. Bronstein und K. A. Semendjajew: *Taschenbuch der Mathematik.* Harri Deutsch, Zürich und Frankfurt/M., 7. durchgesehene und verbesserte Auflage, 1967.

[8] Thomas H. Cormen, Charles E. Leiserson und Ronald R. Rivest: *Introduction to algorithms.* The MIT electrical engineering and computer science series. The MIT Press, 1990.

[9] E. W. Dijkstra: *A discipline of programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[10] Michael R. Garey und David S. Johnson: *Computers and intractability: a guide to the theory of NP-completeness.* W.H. Freeman and Co., San Francisco, 1979.

[11] Phillip Gibbons, Richard Karp, Vijaya Ramachandran, Danny Soroker und Robert Endre Tarjan: *Transitive compaction in parallel via branchings.* Journal of Algorithms, 12(1):110–125, März 1991.

[12] David Gries: *The science of computer programming.* Springer, New York, 1981.

[13] Jonathan L. Gross und Jay Yellen: *Graph theory and its applications.* CRC Press, Boca Raton, 1990.

[14] Dov Harel: *A linear time algorithm for finding dominators in flow graphs and related problems.* In: *Proceedings of the seventeenth annual ACM Symposium on theory of computing*, Seiten 185–194, Providence, Rhode Island, 6.–8. Mai 1985.

[15] Dov Harel und Robert Endre Tarjan: *Fast algorithms for finding nearest common ancestors.* SIAM Journal on Computing, 13(2):338–355, Mai 1984.

[16] Ellis Horowitz und Sartaj Sahni: *Fundamentals of computer algorithms.* Computer Science Press, Rockville, Maryland, 1984.

[17] Harry T. Hsu: *An algorithm for finding a minimal equivalent graph of a digraph.* Journal of the ACM, 22(1):11–16, Januar 1975.

[18] Dieter Jungnickel: *Graphen, Netzwerke und Algorithmen.* BI-Wissenschaftsverlag, Mannheim, 3. vollständig überarbeitete und erweiterte Auflage, 1994.

[19] Brian W. Kernighan und Dennis M. Ritchie: *Programmieren in C.* Carl Hanser, München, 2. Auflage, 1990.

[20] Samir Khuller, Balaji Raghavachari und Neal Young: *Approximating the minimum equivalent digraph.* SIAM Journal on Computing, 24(4):859–972, August 1995.

[21] Dennis M. Moyles und Gerald L. Thompson: *An algorithm for finding a minimum equivalent graph of a digraph.* Journal of the ACM, 16(3):455–460, Juli 1969.

[22] Hartmut Noltemeier: *Reduktion von Präzedenzstrukturen.* Zeitschrift für Operations Research, 20:151–159, 1976. Physica-Verlag, Würzburg.

[23] Thomas Ottmann und Peter Widmayer: *Algorithmen und Datenstrukturen*, Band 70 der Reihe *Informatik.* BI-Wissenschaftsverlag, Mannheim, 1990.

[24] Jesús N. Ravelo: *Two graph algorithms derived.* Acta Informatica, 36(6):489–510, 1999.

[25] RelView. Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, URL: `http://www.informatik.uni-kiel.de/~progsys/relview.html`.

[26] Sartaj Sahni: *Computationally related problems.* SIAM Journal on Computing, 3(4):262–279, Dezember 1974.

[27] Gunther Schmidt und Thomas Ströhlein: *Relationen und Graphen.* Springer, Berlin, 1993.

[28] Klaus Simon: *Finding a minimal transitive reduction in a strongly connected digraph within linear time.* In: M. Nagl (Herausgeber): *Graph-theoretic concepts in computer science (Proceedings 15th international workshop on graph-theoretic concepts in computer science)*, Band 411 der Reihe *Lecture Notes in Computer Science*, Seiten 245–259. Springer, Berlin, Juni 1990.

[29] Klaus Simon. e-mail, 29. Januar 2001.

[30] Volker Strassen: *Gaussian elimination is not optimal.* Numerische Mathematik, 13:354–356, 1969.

[31] Robert Endre Tarjan: *Depth-first search and linear graph algorithms.* SIAM Journal on Computing, 1(2):146–160, Juni 1972.

[32] Robert Endre Tarjan: *Data structures and networks algorithms*, Band 44 der Reihe *Cbms/nsf regional conference series in applied mathematics.* Society for industrial and applied mathematics, Philadelphia, 1983.

Abbildungsverzeichnis

Tabellenverzeichnis

Hiermit erkläre ich an Eides Statt, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 30. April 2001 (Christian Kasper)