überblick1.23

Ludwig Maximilians Universität München

Seminararbeit

im Seminar Programmanalyse

Thema: Algorithmen

eingereicht von: Steffen Ritter

Inhaltsverzeichnis

1 Einleitung

Im Folgenden sollen Algorithmen behandelt werden, die zum Lösen von Gleichungen und Ungleichungen dienen.
Dabei ist die Lösung einer Gleichung auch immer gleich die Lösung einer Ungleichung. Es muss lediglich das = in einem Gleichungssystem durch ein $\geq$ ersetzt werden, um ein adäquates Ungleichungssystem zu erhalten.
So haben z.B. das Ungleichungssystem $x \geq t_1, ..., x \geq t_n$ und das Gleichungssystem $x = x \cup t_1 \cup ... \cup t_n$ dieselbe Lösung. Jede Lösung des Gleichungssystems ist somit auch die kleinste Lösung des Ungleichungssystems.
Und da im folgenden durch Fixpunktanalyse mit Constraints der kleinstmögliche Fixpunkt gesucht werden soll, macht es keinen Unterschied, ob die zu betrachtenden Algorithmen Gleichungssysteme oder Ungleichungssysteme lösen, da der kleinstmögliche Fixpunkt ja auch die kleinstmögliche Lösung des Ungleichungssystems ist.

1.1 Beispiel 1

Nun werden für die Gleichungen eine Reihe von Fluss-Variablen eingesetzt, für die die Lösung des Gleichungssystems erarbeitet werden soll.
Das nun folgende Beispiel entstammt der Data Flow Analysis. Gegeben ist ein Programm, in dem eine While-Schleife existiert. Das Problem ist nun, dass Variablen beim Eintritt und Austritt in, bzw. aus der Schleife verschiedene Werte haben können.
So ist es insbesondere beim Eingang in die Schleife so, dass die entsprechende Variable 2 Werte annehmen kann:
Entweder den Wert, den sie durch den Programmfluss davor bekommen hat, oder den, den sie am Ende de letzten Abarbeitung der Schleife hatte.
Dies geschieht dadurch, dass die Schleife entweder das erste Mal betreten wird, oder dadurch, dass die Schleife schon einmal abgearbeitet wurde, die Bedingung aber noch „true“ ist und die Schleife somit noch einmal abgearbeitet wird.

Nun sei folgendes While-Programm gegeben:

$if[b_1]^1\ then\ (while[b_2]^2\ do[x := a_1]^3)$
$else\ (while[b_3]^4\ do[x := a_2]^5);$
$[x := a_3]^6$

Hierbei existieren für jeden Punkt im Programmfluss Eingangs- und Ausgangswerte der entsprechenden Fluss-Variablen.
Das Gleichungssystem, das durch Reaching Definitions Analysis aus diesem Programm entsteht, hat folgende Form:

$RD_{entry}(1) = X_?$; $RD_{exit}(1) = RD_{entry}(1)$
$RD_{entry}(2) = RD_{exit}(1) \cup RD_{exit}(3)$; $RD_{exit}(2) = RD_{entry}(2)$
$RD_{entry}(3) = RD_{exit}(2)$; $RD_{exit}(3) = (RD_{entry}(3)/X_{356?}) \cup X_3$
$RD_{entry}(4) = RD_{exit}(1) \cup RD_{exit}(5)$; $RD_{exit}(4) = RD_{entry}(4)$
$RD_{entry}(5) = RD_{exit}(4)$; $RD_{exit}(5) = (RD_{entry}(5)/X_{356?}) \cup X_5$
$RD_{entry}(6) = RD_{exit}(2) \cup RD_{exit}(4)$; $RD_{exit}(6) = (RD_{entry}(6)/X_{356?}) \cup X_6$

Ziel ist es nun, ein Constraint-System der Form $x \supseteq t$ mit Fluss-Variablen zu schaffen. Dazu werden die Ein- und Ausgangswerte durch $x_1, ..., x_{12}$ ersetzt.
Dadurch entsteht folgendes Gleichungssystem:

$x_1 = X_?$ $x_7 = x_1$
$x_2 = x_7 \cup x_9$ $x_8 = x_2$
$x_3 = x_8$ $x_9 = (x_3/X_{356?}) \cup X_3$
$x_4 = x_7 \cup x_{11}$ $x_{10} = x_4$
$x_5 = x_{10}$ $x_{11} = (x_5/X_{356?}) \cup X_5$
$x_6 = x_8 \cup x_{10}$ $x_{12} = (x_6/X_{356?}) \cup X_6$

Da aber vor allem Interesse an den Eingangswerten besteht, wird das Gleichungssystem wie folgt vereinfacht:

$x_1 = X_?$
$x_2 = x_1 \cup (x_3/X_{356?}) \cup X_3$
$x_3 = x_2$
$x_4 = x_1 \cup (x_5/X_{356?}) \cup X_5$
$x_5 = x_4$
$x_6 = x_2 \cup x_4$

Durch diese Änderung der Darstellung entsteht offensichtlich kein Informationsverlust.

1.2 Überblick

Das aus Beispiel 1 gegebene endliche Gleichungssystem muss nun gelöst werden. Dazu wird zuerst ein abstrakter Worklist Algorithmus angegeben, da dieser leichter zu implementieren und zu verstehen ist und die Grundlagen verständlich macht. Danach wird die konkrete Implementierung der Worklist behandelt. Dabei wird zuerst auf LIFO eingegangen, da dies die naivste und einfachste Variante ist. Allerdings braucht der Algorithmus bei einer LIFO-Implementierung viele Durchläufe, bis er terminiert. Deshalb wird anschließend auf eine Implementierung mit Reverse Postorder und daraus resultierend mit einem Round Robin Algorithmus eingegangen. In diesem Fall terminiert der Algorithmus schon nach weniger Schritten als bei LIFO. Um die Laufzeit noch zu optimieren, wird am Ende noch eine Implementierung der Worklist mit Strongly Connected Components aufgezeigt.

2 Abstrakter Worklist Algorithmus

2.1 Struktur von Worklist Algorithmen

Die naivste Variante, um das Gleichungssystem zu lösen, wäre, eine Fixpunktiteration ausgehend vom Bottom-Element des zugrundeliegendne Verbandes durchzuführen und damit in jedem Schritt alle Constraints abzuarbeiten.
Die Verwendung einer Worklist beitet hierbei jedoch schon eine Verbesserung. So ist die grundlegende Idee einer Worklist, dass es Tasks gibt, die abgearbeitet werden müssen. Dazu werden sie in einer Worklist gespeichert. Bei der Durchführung der Abarbeitung wählt eine Iteration einen Task aus der Worklist aus, entfernt ihn aus der Liste und führt ihn aus.
Wenn nun bei der Durchführung eines Tasks neue Tasks entstehen, werden diese wiederum in die Worklist eingefügt.
Diese Prozedur wird solange wiederholt, bis die Worklist leer ist, es also keine Tasks mehr zu erledigen gibt.

2.2 Operationen auf Worklist

Im Folgenden wird die Worklist zuerst als unsortierte Menge von Constraints betrachtet.
Auf diese Menge existieren folgende Operationen:

- empty
- insert
- extract

„Empty“ bezeichnet die leere Worklist.
Mit „insert“ wird der Liste ein Constraint der Form $x \geq t$ hinzugefügt.
Und mit „extract“ kann schließlich ein Constraint aus der Liste entfernt werden.

2.3 Abstrakter Algorithmus

Der nun folgende abstrakte Algorithmus ist in zwei Schritte unterteilt.
Im 1. Schritt wird die Worklist initialisiert. Ihr werden alle Constraints hinzugefügt. Des Weiteren wird eine Menge initialisiert, die Constraints enthält, deren Werte durch die Auswertung einer Fluss-Variable beeinflusst werden und die deshalb noch einmal ausgeweret werden müssen. Und zuletzt wird noch die Ergebnisfunktion initialisiert. Im 2. Schritt wird nun durch die Worklist iteriert. Dabei werden die Werte der Ergebnisfunktion geupdatet und die Constraints, deren Werte durch die Auswertung der aktuellen Fluss-Variable beeinflusst werden, der Worklist hinzugefügt.

```
INPUT: A system S of constraints: x1 >= t1,...xN >= tN
OUTPUT: The least solution: Analysis
METHOD:

Step 1: Initialisation (of W, Analysis and infl)

W:= empty;
for all x>=t in S do
W:= insert((x>=t),W)
Analysis[x]:= _;
infl[x]:={};
for all x>=t in S do
for all x' in FV(t) do
infl[x']:=infl[x'] U {x>=t};

Step 2: Iteration (updating W and Analysis)

while W != empty do
((x>=t),W):= extract(W);
new:= eval(t,Analysis);
if Analysis[x]!>= new then
Analysis[x]:= Analysis[x] U new;
for all x'>=t' in infl[x] do
W:= insert((x'>=t'),W);


USING:
function eval(t, Analysis)
return[|t|](Analysis)

value empty

function insert((X>=t), W)
return...

function extract(W)
return...
```

2.4 Beispiel 2

Die Grundlage dieses Beispiels ist das vereinfachte Gleichungssystem aus Beispiel 1.
Nachdem nun auf diesem Gleichungssystem basierend der 1. Schritt des Algorithmus durchgeführt wurde, enthält die Worklist alle Gleichungen und die Menge der beeinflussten Variablen sieht wie folgt aus:

	x_1	x_2	x_3	x_4	x_5	x_6
beeinflusst	$\{x_2, x_4\}$	$\{x_3, x_6\}$	$\{x_2\}$	$\{x_5, x_6\}$	$\{x_4\}$	$\oslash$

2.5 Eigenschaften des Algorithmus

Die oben beschriebene Initialisierung der Menge der beeinflussten Constraints soll nun näher betrachtet werden.
Angenommen, es existieren N Constraints. Dann benötigt die Initialisierung der Menge O(N) Schritte.
Sei nun die Größe der Constraints auf der rechten Seite $M \geq 1$. Somit benötigt die Auswertung einer rechten Seite eines Constraints O(M) Schritte.
Damit benötigt die Auswertung von allen rechten Seiten aller Constraints O(N*M) Schritte. Und somit beträgt die Gesamtkomplexität der Initialisierung und Auswertung der Menge O(N + N*M).

3 Implementierung der Worklist

3.1 LIFO

3.1.1 Algorithmus

Bisher wurden noch keine Details zur konkreten Initialisierung der Worklist und der Operatoren behandelt. Um jedoch eine besser Laufzeit zu erhalten, ist es nötig, die Worklist möglichst effizient zu implementieren.
Die einfachste Art dies zu tun, ist, die Worklist einfach als Liste zu betrachten, die wie ein Stack verwendet wird. Das Element, das als letztes eingefügt wurde, also auf dem Stack ganz oben liegen würde, wird als erstes wieder entfernt, wenn es zur Abarbeitung der Liste kommt.
Somit ergibt sich ein Last In First Out Algorithmus.

3.1.2 Beispiel 3

In diesem Beispiel wird die Funktionsweise des LIFO Algorithmus betrachtet. Grundlage sind wieder das Gleichungssystem aus Beispiel 1 und die Menge mit beeinflussten Constraints aus Beispiel 2. In der ersten Spalte der Tabelle wird die Worklist dargestellt und in den restlichen Spalten die Werte der Variablen in der Ergebnisfunktion. Ein „-“ bedeutet dabei, dass sich an dem Wert nichts geändert hat.
Der Algorithmus operiert nun wie folgt:
Die erste Variable wird aus der Worklist genommen und ausgewertet. Der Wert wird in die Ergebnisfunktion eingetragen. Wenn bei der Auswertung andere Variablen gemäß der Menge aus Beispiel 2 beeinflusst werden, werden diese am Anfang der Worklist wieder eingefügt. Diese Prozedur wird solange wiederholt, bis die Liste leer ist.

W	x_1	x_2	x_3	x_4	x_5	x_6
$[x_1, x_2, x_3, x_4, x_5, x_6]$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
$[x_2, x_4, x_2, x_3, x_4, x_5, x_6]$	$X_?$	-	-	-	-	-
$[x_3, x_6, x_4, x_2, x_3, x_4, x_5, x_6]$	-	$X_{3?}$	-	-	-	-
$[x_2, x_6, x_4, x_2, x_3, x_4, x_5, x_6]$	-	-	$X_{3?}$	-	-	-
$[x_4, x_2, x_3, x_4, x_5, x_6]$	-	-	-	-	-	$X_{3?}$
$[x_5, x_6, x_2, x_3, x_4, x_5, x_6]$	-	-	-	$X_{5?}$	-	-
$[x_4, x_6, x_2, x_3, x_4, x_5, x_6]$	-	-	-	-	$X_{5?}$	-
$[x_2, x_3, x_4, x_5, x_6]$	-	-	-	-	-	$X_{35?}$
$[x_3, x_4, x_5, x_6]$	-	-	-	-	-	-
$[x_4, x_5, x_6]$	-	-	-	-	-	-
$[x_5, x_6]$	-	-	-	-	-	-
$[x_6]$	-	-	-	-	-	-
$[]$	-	-	-	-	-	-

Der Nachteil dieser Implementierung ist, dass beim Einfügen nicht überprüft wird, ob die Constraints schon in der Worklist vorhanden sind. Deshalb kommt es zu unnötigen Mehrfachberechnungen.
Um dieses Problem zu lösen, könnte die LIFO-Strategie so implementiert werden, dass beim Einfügen zuerst geprüft wird, ob ein Constraint bereits vorhanden ist.

3.2 Reverse Postorder

Die Idee einer Reverse Postorder Implementierung der Worklist basiert auf der Problemlösung des Nachteils von LIFO.
Den Constraints wird eine feste Reihenfolge zugeordnet. Danach werden alle Constraints der Reihenfolge nach ausgewertet. Und um alle Änderungen zu berücksichtigen, die durch die Auswertung in anderen Constraints entstehen, werden die geänderten Constraints danach abgearbeitet.

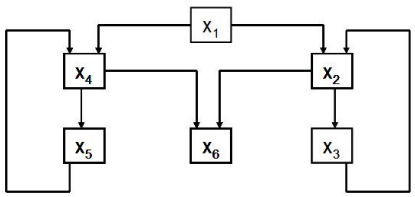
3.2.1 Graph-Struktur

Um eine passende Reihenfolge für die Constraints zu erhalten, bietet sich die Graph-Struktur an.
Hierbei handelt es sich um die grafische Darstellung der Abhängigkeiten zwischen Constraints. Dabei existiert für jedes Constraint ein Knoten. Eine Kante zwischen zwei Knoten existiert, falls die Auswertung des einen Constraints den Wert des anderen beeinflusst. Des Weiteren existiert eine Wurzel, wobei es sich um einen Knoten handelt, von dem aus jeder andere Knoten erreichbar ist.

3.2.2 Beispiel 4

Um die Abhängigkeit der Constraints darzustellen, wird als Grundlage wieder die Menge der beeinflussten Constraints aus Beispiel 2 herangezogen: Die daraus abgeleitete grafische Darstellung sieht wie folgt aus:

	x_1	x_2	x_3	x_4	x_5	x_6
beeinflusst	$\{x_2, x_4\}$	$\{x_3, x_6\}$	$\{x_2\}$	$\{x_5, x_6\}$	$\{x_4\}$	$\oslash$



Daraus resultieren die Reverse Postorder: $x_1, ..., x_6$.

3.2.3 Algorithmus

Um eine Reverse Postorder Implementierung zu erreichen, wird nun der zweite Schritt das abstrakten Algorithmus verändert, indem die Worklist als Paar von zwei Listen implementiert wird. Die erste Liste beinhaltet dabei alle aktuell abzuarbeitenden Knoten. Die zweite Liste ist eine Liste mit Knoten, die durch die Auswertung von Knoten aus der ersten Liste verändert werden. Dabei sind beide Listen in der festgelegten Reverse Postorder sortiert. Nachdem die erste Liste abgearbeitet ist, wird die zweite Liste in die erste geladen und ebenfalls abgearbeitet. Sollten dabei wieder Änderungen in Constraints entstehen, muss wiederum eine neue zweite Liste angelegt werden. Diese Prozedur wiederholt sich so lange, bis bei der Auswertung der zweiten Liste keine neuen Werte mehr entstehen und somit auch keine neue zweite Liste mehr angelegt werden muss.

3.2.4 Beispiel 5

In der 1. Spalte ist die Liste mit den aktuell abzuarbeitenden Knoten dargestellt. Die 2. Liste enthält die Knoten, die durch Abarbeitung der 1. Liste verändert werden. Und die restlichen Spalten stellen wieder die Werte der Ergebnisfunktion dar.Der Algorithmus operiert dann wie folgt: Zuerst wir die Liste mit den Constraints in Reverse Postorder in Liste 1 geladen. Danach wird das erste Element aus der 1. Liste genommen und ausgewertet. Die sich dabei ändernden Constraints werden in Liste 2 eingefügt und nach Reverse Postorder sortiert. Wenn dann die erste Liste leer ist, wird die zweite Liste in die erste geladen und abgearbeitet, wobei keine neuen Werte entstehen. Dadurch terminiert der Algorithmus nach insgesamt weniger Iterationsschritten wie bei der LIFO Strategie.

W.c	W.p	x_1	x_2	x_3	x_4	x_5	x_6
[]	$\{x_1, x_2, x_3, x_4, x_5, x_6\}$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
$[x_2, x_3, x_4, x_5, x_6]$	$\{x_2, x_4\}$	$X_?$	-	-	-	-	-
$[x_3, x_4, x_5, x_6]$	$\{x_2, x_3, x_4, x_6\}$	-	$X_{3?}$	-	-	-	-
$[x_4, x_5, x_6]$	$\{x_2, x_3, x_4, x_6\}$	-	-	$X_{3?}$	-	-	-
$[x_5, x_6]$	$\{x_2, x_3, x_4, x_5, x_6\}$	-	-	-	$X_{5?}$	-	-
$[x_6]$	$\{x_2, x_3, x_4, x_5, x_6\}$	-	-	-	-	$X_{5?}$	-
$[x_2, x_3, x_4, x_5, x_6]$	$\oslash$	-	-	-	-	-	$X_{35?}$
$[x_3, x_4, x_5, x_6]$	$\oslash$	-	-	-	-	-	-
$[x_4, x_5, x_6]$	$\oslash$	-	-	-	-	-	-
$[x_5, x_6]$	$\oslash$	-	-	-	-	-	-
$[x_6]$	$\oslash$	-	-	-	-	-	-
[]	$\oslash$	-	-	-	-	-	-

3.2.5 Round Robin Algorithmus

Um die Verwaltung der zwei Listen nun zu vereinfachen, wird die Liste, die die geänderten Constraints enthält, nicht mehr explizit als Liste verwaltet. Viel mehr wird sie durch einen boolschen Wert ersetzt. Die Idee dahinter ist, dass der Wert immer auf „true“ gesetzt wird, wenn sich bei der Auswertung eines Constraints ein neuer Wert ergeben hat. Dann wird davon ausgegangen, dass sich auch andere Constraints dadurch verändert haben könnten und somit wird der Wert auf „true“ gesetzt. Ändert sich der Wert in der Ergebnisfunktion hingegen bei der Auswertung nicht, wird der Wert auf „false“ gesetzt, da sich dann ja auch kein weiteres Constraint verändert haben kann.

Der dadurch entstehende Algorithmus ist wieder in zwei Schritte unterteilt. Im ersten Schritt werden wieder die Worklist und die Ergebnisfunktion initialisiert. Des weiteren wird diesmal noch ein boolscher Wert initialisiert, der von Beginn an „true“ ist.
Im zweiten Schritt wird zuerst in einer While-Schleife geprüft, ob der boolsche Wert „true“ ist. Daraufhin wird der Wert auf „false“ gesetzt. Anschließend werden alle Constraints in einer For-Schleife ausgewertet und dabei die Ergebnisfunktion geupdatet.
Wenn sich dabei für einen Constraint ein neuer Wert ergibt, wird die boolsche Variable wieder auf „true“ gesetzt und die While-Schleife wird erneut abgearbeitet.
Sollte sich bei der Abarbeitung der Constraints kein neuer Wert mehr ergeben, bleibt der boolsche Wert „false“ und der Algorithmus terminiert.

```
INPUT:
A system S of constraints: x1>=t1,..., xN>=tN
ordered 1 to N in reverse postorder

OUTPUT:
The least solution: Analysis

METHOD:

Step 1:

Initialisation

for all x of X do
Analysis[x]:=_;
change:=true;

Step 2:

Iteration (updating Analysis)

while change do
change:=false;
for i:=1 to N do
new:=eval(ti, Analysis);
if Analysis[xi]!>=new then
change:=true;
Analysis[xi]:=Analysis[xi] U new;

USING:

function eval(t, Analysis)
return[|t|](Analysis
```

3.2.6 Beispiel 6

Es wir nun wieder das Gleichungssystem von Beispiel 1, sowie die Reverse Postorder von Beispiel 4 zu Grunde gelegt.
Die erste Spalte der Tabelle beinhaltet die boolsche Variable „change“, die für das Anzeigen einer Änderung des Wertes eines Constraints zuständig ist. In den restlichen Spalten werden wieder die Werte der Variablen in der Ergebnisfunktion dargestellt.
In der ersten Zeile wurde die Worklist und „change“ initialisiert. Dann wird in der 2. Zeile die While-Schleife das erste Mal betreten und „change“ wird

auf „false“ gesetzt.
In der dritten Zeile wird dann mit der For-Schleife, also der Auswertung der Constraints begonnen. Solange die Auswertung immer wieder neue Werte erbringt, bleibt „change“ auf „true“.
Nachdem dann alle Constraints einmal ausgewertet wurden, die For-Schleife also zu Ende ist, wird die While-Schleife erneut abgearbeitet, da „change“ ja noch „true“ ist. Im Folgenden wird „change“ wieder „auf „false“ gesetzt und die For-Schleife wird erneut betreten. Bei der Abarbeitung der Constraints ergeben sich jedoch keine neuen Werte mehr. somit bleibt „change“ „false“, die While-Schleife wird nicht mehr betreten und der Algorithmus terminiert. Der Algorithmus braucht zwar mehr Schritte als bei den vorigen Strategien, ist dafür aber einfacher zu handhaben.

change	x_1	x_2	x_3	x_4	x_5	x_6
ture	⊘	⊘	⊘	⊘	⊘	⊘
*false						
ture	$X_?$	-	-	-	-	-
ture	-	$X_{3?}$	-	-	-	-
ture	-	-	$X_{3?}$	-	-	-
ture	-	-	-	$X_{5?}$	-	-
ture	-	-	-	-	$X_{5?}$	-
ture	-	-	-	-	-	$X_{35?}$
*false						
false	-	-	-	-	-	-
false	-	-	-	-	-	-
false	-	-	-	-	-	-
false	-	-	-	-	-	-
false	-	-	-	-	-	-
false	-	-	-	-	-	-

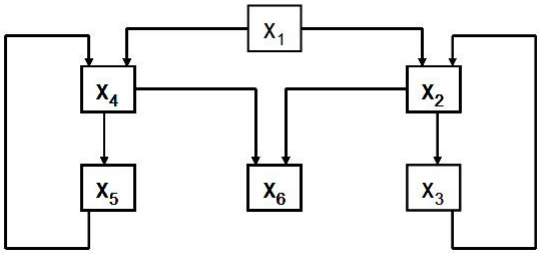
3.3 Strongly Connected Components

Eine letzte Möglichkeit, die Laufzeit des Algorithmus zu verbessern ist, mit sogenannten Strongly Connected Components zu arbeiten.
Dabei geht man von der Darstellung der Constraints als Graph aus. Die Strongly Connected Components sind die maximal stark verbundenen Untergraphen, also diejenigen Knoten, von denen aus man andere Knoten erreichen kann und die wiederum von diesen anderen Knoten erreichbar sind. Die Verbindung zwischen den Strongly Connected Components (SCC) kann in einem reduzierten Graph dargestellt werden. Aus dieser Darstellung können die SCC nach dem Schema $scc_1 \leq scc_2$ angeordnet werden, wenn eine Kante von scc_1 zu scc_2 existiert.
Eine Kante existiert dann, wenn in der normalen Darstellung als Graph eine

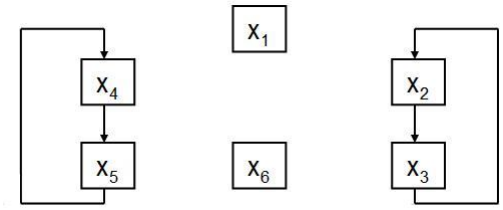
Kante von einem der Constraints aus dem SCC zu einem anderen Constraint aus einem anderen SCC existiert.

3.3.1 Beispiel 7

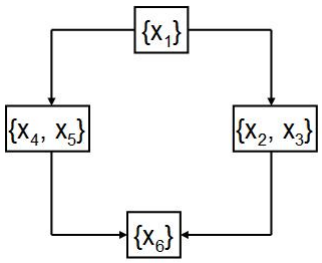
Dieses Beispiel basiert wieder auf dem Gleichungssystem aus Beispiel 1 und der daraus entstandenen Graph-Struktur aus Beispiel 4:



Daraus ergibt sich folgende Darstellung für die SCC:



Aus den SCC lässt sich nun der reduzierte Graph bilden:



Hieraus lässt sich nun die Reihenfolge der SCC ablesen. Dabei wird festgestellt, dass sich in diesem Beispiel zwei mögliche Reihenfolgen für die SCC ergeben:

- $\{x_1\}, \{x_2, x_3\}, \{x_4, x_5\}, \{x_6\}$
- $\{x_1\}, \{x_4, x_5\}, \{x_2, x_3\}, \{x_6\}$

3.3.2 Algorithmus

Der Algorithmus dazu unterteilt die Worklist in drei Listen.
In der ersten Liste befinden sich alle SCC in ihrer festgelegten Reihenfolge. Liste 2 beinhaltet die abzuarbeitenden Constraints des aktuellen SCC. Und in Liste 3 befinden sich die aktuell abzuarbeitenden Constraints aus Liste 2 in Reverse Postorder.
Im ersten Schritt wird nun der aus der Reihenfolge resultierende kleinste SCC aus Liste 1 genommen und dessen Constraints in Liste 2 geschrieben. Danach werden im 2. Schritt die Constraints in Liste 2 nach Reverse Postorder sortiert und in Liste 3 geschrieben. Nun werden im letzten Schritt des Algorithmus die Constraints aus Liste 3 der Reihe nach ausgewertet. Wenn dabei wiederum ein anderer Constraints beeinflusst wird, der sich nicht mehr in Liste 1 befindet, wird dieser Liste 1 hinzugefügt.
Wenn Liste 3 nun abgearbeitet ist, wird der Vorgang wiederholt und wieder der kleinste SCC aus Liste 1 entfernt, in Liste 2 sortiert und in Liste 3 geschrieben und dort abgearbeitet.
Da sich die Constraints von einem SCC immer gegenseitig beeinflussen, wird bei der 1. Abarbeitung der Constraints des SCC immer der jeweils andere Constraint Liste 1 hinzugefügt. Somit ist der nächste SCC, der aus Liste 1 entfernt wird, wiederum dieser SCC, da er ja der aktuell kleinste ist und beide Constraints wieder in Liste 1 enthalten sind. Die Prozedur wiederholt sich so lange, bis sich bei der Auswertung der Cosntraitns keine neuen Werte mehr ergeben und der jeweils andere somit nicht mehr Liste 1 hinzugefügt werden muss.
Dieser Vorgang wird so oft wiederholt, bis alle SCC aus Liste 1 entfernt und ausgewertet wurden. Damit terminiert der Algorithmus.
Der geänderte 2. Schritt des abstrakten Algorithmus sieht nun so aus:

```
if W.c = nil then
scc:=min{fst(srPostorder[x>=t]) | (x>=t) of W.p};
W_scc:= {(x>=t) of W.p | fst(srPostorder[x>=t]) =scc};
W.c:= sort_srPostorder(W_scc);
W.p:= W.p\W_scc;
return (head(W.c), (tail(W.c), W.p))
```

W_{scc} ist dabei die Liste mit den Constraints des aktuellen SCC. $W.c$ ist die sortierte Liste der Constraints des aktuellen SCC und $W.p$ ist die Liste mit den restlichen, zukünftig abzuarbeitenden Constraints.

3.3.3 Beispiel 8

Zur Veranschaulichung das Algorithmus wird wiederum das Gleichungssystem aus Beispiel 1 und eine Reihenfolge der SCC aus Beispiel 7, nämlich $\{x_1\}, \{x_2, x_3\}, \{x_4, x_5\}, \{x_6\}$,

herangezogen.
In der ersten Spalte befindet sich die sortierte Liste der aktuell abzuarbeitenden Constraints des aktuellen SCC. Spalte 2 beinhaltet die später abzuarbeitenden Constraints anderer SCC. Die restlichen Spalten stellen wiederum die Werte in der Ergebnisfunktion dar.
Zuerst wird nun der kleinste SCC, nämlich $\{x_1\}$ aus Liste 1 entfernt. Dessen Constraint x_1 muss nicht sortiert werden, da es nur eines ist und kann so direkt in W.c geladen und anschließend ausgewertet werden.
Danach wird wiederum der kleinste SCC aus W.p entfernt, nämlich $\{x_2, x_3\}$. Dessen Constraints x_2 und x_3 werden nach Reverse Postorder geordnet und in W.c geschrieben. Zuerst wird x_2 ausgewertet. Die Auswertung beeinflusst x_3. x_3 ist nicht mehr in W.p enthalten, also wird es dort wieder eingefügt. Danach wird x_3 ausgewertet, was wiederum x_2 beeinflusst. Somit wird auch x_2 wieder W.p zugefügt.
Der nächste kleinste SCC in W.p ist wiederum $\{x_2, x_3\}$. Somit werden wieder x_2 und x_3 sortiert und in W.c eingetragne. Da sich bei der 2. Auswertung keine neuen Werte für die Constraints ergeben, werden die Constraints x_3 und x_2, die durch die Auswertungen von x_2 bzw. x_3 beeinflusst werden, nicht mehr in W.p eingetragen.
Somit ist das nächst kleinere SCC, der im nächsten Schritt aus Liste W.p entfernt wird $\{x_4, x_5\}$.
Dieser Vorgang wiederholt sich so lange, bis W.p leer ist und x_6 ausgewertet wurde.
Dabei benötigt dieser Algorithmus wiederum weniger Schritte als die Algorithmen davor und ist somit der effizienteste.

W.c	W.p	x_1	x_2	x_3	x_4	x_5	x_6
[]	$\{x_1, x_2, x_3, x_4, x_5, x_6\}$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
[]	$\{x_2, x_3, x_4, x_5, x_6\}$	$X_?$	-	-	-	-	-
$[x_3]$	$\{x_3, x_4, x_5, x_6\}$	-	$X_{3?}$	-	-	-	-
[]	$\{x_2, x_3, x_4, x_5, x_6\}$	-	-	$X_{3?}$	-	-	-
$[x_3]$	$\{x_4, x_5, x_6\}$	-	-	-	-	-	-
[]	$\{x_4, x_5, x_6\}$	-	-	-	-	-	-
$[x_5]$	$\{x_5, x_6\}$	-	-	-	$X_{5?}$	-	-
[]	$\{x_4, x_5, x_6\}$	-	-	-	-	$X_{5?}$	-
[]	$\{x_6\}$	-	-	-	-	-	-
[]	$\oslash$	-	-	-	-	-	$X_{35?}$

4 Zusammenfassung

Es kann also festgehalten werden, dass zumindest für dieses Beispiel, der SCC-Algorithmus der effizienteste ist. Reverse Postorder ist nicht signifikant schlechter, jedoch sind die zwei Listen kompliziert zu verwalten. Um dies zu vereinfachen, kann - jedoch auf Kosten der Effizienz - auf eine Implementierung des Round Robin Algorithmus zurückgegriffen werden. Die Implementierung mit LIFO ist am uneffizientesten, jedoch vom reinen Algorithmus her am einfachsten umzusetzen.