Fakultät Informatik, Mathematik und Naturwissenschaften

# Implementierung der Tiefensuche in KIV

eingereicht als

Prüfungsrelevante Studienleistung

im Studiengang Informatik (Master)

von

Ricardo Hofmann
Matthias Neubert
Stefan Veit

Leipzig, 26. Februar 2011

Betreuender Professor: Prof. Dr. Uwe Petermann

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Aufgabenstellung

Innerhalb der Lehrveranstaltung Programmverifikation, ist es die Aufgabe, für eine selbstgewählte Problemstellung, ein Verifikationsprojekt mit Hilfe der Software *KIV* (Karlsruhe Interactive Verifier) zu erstellen. Für die vorliegende Arbeit wurde diesbezüglich das Problem der Tiefensuche auf Graphen betrachtet. Dabei handelt es sich um einen Algorithmus, der mittels der Strategie der „Tiefen-Zuerst-Suche“, die Existenz von Pfaden - von einem gegebenen Startknoten zu einer Zielknotenmenge - prüft respektive in einer modifizierten Variante den dabei resultierenden Pfad zurückgibt.

Eine genaue Beschreibung des Algorithmus sowie der Überblick über die konkrete Umsetzung des Projektes, wird Gegenstand der folgenden Kapitel sein.

## 1.2 Motivation

Die Graphentheorie ist ein wichtiges Gebiet innerhalb der Informatik. Für viele Problemstellungen dient die Datenstruktur Graph der Abstraktion und Abbildung verschie-

denster Fragestellungen.

Speziell für das Verifikationswerkzeug KIV fehlen jedoch diesbezüglich noch entsprechende Bibliotheken. Ein erster Ansatz wurde durch die Umsetzung der Graphenbibliothek durch das studentische Projekt von Marc Drobek und David Redlich realisiert. Aufbauend auf dieser bereits vorhandenen Spezifikation/Implementierung, soll dieses Projekt, die Bibliothek um die algorithmische Herangehensweise der sogenannten Tiefensuche anreichern. Die im Weiteren beschriebenen Umsetzungen und Maßnahmen zur Spezifikation, Implementierung bis hin zur Verifikation sollen dies veranschaulichen.

## 1.3 Gliederung der Arbeit

Die folgenden Kapitel beschäftigen sich mit der konkreten Beschreibung der betrachteten Problemstellung sowie der Erläuterung der eingeführten Spezifikationen.

Kapitel 2 wird dabei insbesondere auf die Tiefensuche, deren Implementierung sowie die Umsetzung in KIV eingehen.

Ein detaillierteren Einblick in die Realisierung innerhalb des Karlsruhe Interactive Verifiers bietet Kapitel 3, welches auf die einzelnen Spezifikationen, die implementierten Module sowie verwendete Hilfssätze eingeht.

Den Abschluss wird Kapitel 4 bilden, wo die wichtigsten Kernpunkte der Arbeit nochmals zusammengefasst sind.

# 2
# Problembeschreibung

## 2.1 Tiefensuche

Die Tiefensuche ist ein Verfahren zum Traversieren von Graphen. Das Ziel besteht darin einen Knoten zu finden, der zu einer gegebenen Zielknotenmenge gehört. Die Suche ist genau dann erfolgreich, wenn ein Pfad von einem gegebenen Startknoten zu einem Zielknoten aus der Zielknotenmenge existiert.

Es werden zunächst vom Startknoten alle direkt erreichbaren Knoten ermittelt, d.h. Knoten, die durch eine Kante mit dem Startknoten verbunden sind. Diese Knoten werden in eine open-Liste eingetragen. Auf der open-Liste befinden sich alle Knoten, die von einem expandierten Knoten erreichbar sind, selbst aber noch nicht expandiert wurden. Unter „expandieren“ ist das Ermitteln der Nachbarknoten zu verstehen. Knoten, die bereits expandiert wurden, werden in die close-Liste eingetragen, um sie nicht nochmals zu untersuchen. Bei der Expansion eines Knotens werden nur Knoten berücksichtigt, die sich noch nicht auf der close-Liste befinden. Alle entdeckten Nachfolgerknoten werden vorn an die open-Liste angehangen. Daher kann die open-Liste als Stack betrachtet werden (LIFO-Prinzip). Durch die Verwendung dieser Datenstruktur unterscheidet sich die Tiefen- von der Breitensuche.

Von der open-Liste wird nun immer der erste Knoten entnommen, auf die close-Liste gesetzt und expandiert. Die erhaltenen Nachfolgerknoten werden jeweils der open-Liste hinzugefügt.

Die Suche endet, sobald ein Knoten aus der Zielknotenmenge erreicht wurde oder die open-Liste leer ist. Im letztgenannten Fall existiert kein Pfad vom Startknoten zu einem Zielknoten.

Die Funktionsweise wird anhand des in Abb. 2.1 gezeigten Beispielgraphen verdeutlicht. Dabei ist a der Startknoten und e der einzige Knoten in der Zielknotenmenge. In Tabelle 2.1 sind alle Bearbeitungsschritte der Tiefensuche aufgeführt. Es wird jeweils angegeben, welcher Knoten bearbeitet wird und welche Elemente sich nach der Bearbeitung dieses Knotens auf der open- und close-Liste befinden.

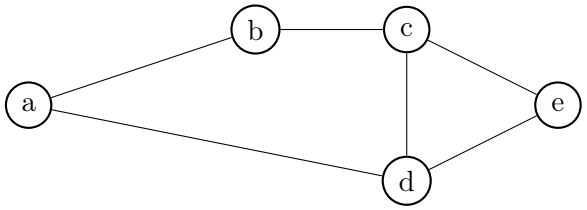


Abbildung 2.1: Beispielgraph

| bearb. Knoten | open | close |
|---|---|---|
| - | [a] | [] |
| a | [b, d] | [a] |
| b | [c, d] | [a, b] |
| c | [e, d] | [a, b, c] |
| e | Abbruch mit Rückgabewert `true` | |

Tabelle 2.1: Abarbeitungschritte für Beispielgraphen

## 2.2 Implementierungen

### Rekursive Implementierung

Die Tiefensuche kann rekursiv wie in Listing 2.1 als Pseudocode dargestellt implementiert werden. Die Funktion wird mit den Paramentern `open = Startknoten`, `close = leere Liste`, `ziel = Zielknotenmenge` und `graph = zu untersuchender Graph` aufgerufen.

Zu Beginn wird geprüft, ob sich der aktuell betrachtete Knoten in der Zielknoten befindet. Ist dies der Fall wird `true` als Ergebnis zurückgegeben. Ansonsten wird der

Knoten der close-Liste hinzugefügt und expandiert. Auf die Implementierung der Funktion `expand` wird nicht näher eingegangen. Sie bestimmt die Nachbarn eines Knotens unter Ausschluss derjenigen, die sich bereits auf der close-Liste befinden.

Ist die open-Liste nach diesem Bearbeitungsschritt nicht leer, so erfolgt ein rekursiver Aufruf der Funktion.

```
1  dfs-rek(open, close, ziel, graph)
2  {
3    knoten = open.pop;
4    if (knoten in ziel)
5    {
6      return true;
7    }
8    else
9    {
10     close := knoten + close;
11     open := expand (knoten, close) + open;
12
13     if(open.isEmpty == false)
14     {
15       dfs-rek(open, close, ziel, graph);
16     }
17     else
18     {
19         return false;
20     }
21   }
22 }
```

Listing 2.1: Pseudocode rekursive Tiefensuche

### Iterative Implementierung

Eine iterative Variante der Tiefensuche ist in Listing 2.2 als Pseudocode dargestellt. Die Aufrufparameter der Funktion sind der Startknoten (`startknoten`), die Zielknotenmenge (`ziel`) und der zu untersuchende Graph (`graph`).

Nach der Initialisierung der open- und close-Liste wird eine while-Schleife solange durchlaufen, bis die open-Liste leer ist oder vorzeitig mit der Rückgabe von `true` abgebrochen wird. In der Schleife wird der erste Knoten der open-Liste entnommen und geprüft, ob dieser zur Zielknotenmenge gehört. Ist dies der Fall war die Suche erfolgreich und es wird

abgebrochen. Andernfalls wird der Knoten der close-Liste hinzugefügt und expandiert.

Ist die open-Liste nach diesem Schritt leer, wird die Schleife verlassen und `false` zurückgegeben, da die Suche erfolglos war. Ansonsten erfolgt der nächste Schleifendurchlauf.

```
1  dfs-it(startknoten, ziel, graph)
2  {
3    open := startknoten;
4    close := [];
5    while(open != empty)
6    {
7      knoten := open.pop;
8      if(knoten in ziel)
9      {
10       return true;
11     }
12     else
13     {
14       close := knoten + close;
15       open := expand(knoten, close) + open;
16     }
17   }
18   return false;
19 }
```

Listing 2.2: Pseudocode iterative Tiefensuche

## 2.3 Umsetzung in KIV

Für die Umsetzung und Verifikation der Tiefensuche in KIV wurde die Graphenbibliothek von M. Drobek und D. Redlich verwendet (vgl. [DR08]). Darauf aufbauend wurden zwei Spezifikationen entwickelt.

`digraph-dfs-proc` spezifiziert die Tiefensuche, wie in Abschnitt 2.1 beschrieben. Das bedeutet der Rückgabewert ist entweder `true` oder `false`, abhängig davon, ob ein Knoten aus der Zielknotenmenge erreicht werden kann oder nicht. Eine genauere Beschreibung der Spezifikation erfolgt im Abschnitt 3.1.

Dieser Spezifikation genügen die Module `dfs-it-1` als iterative Implementierung sowie `dfs-rek-1` als rekursive Implementierung. Details zu diesen Modulen können den Abschnitten 3.2 bzw. 3.3 entnommen werden.

Die zweite Spezifikation (`digraph-dfs-func`, vgl. Abs. 3.5.1) definiert das Verhalten einer Variante der Tiefensuche, die zusätzlich den Pfad vom Startknoten zu einem gefundenen Knoten aus der Zielknotenmenge liefert, sofern dieser existiert. Für diese Spezifikation wurden ebenfalls eine rekursive (`dfs-rek-func`) und eine iterative Implementierung (`dfs-it-func-1`) umgesetzt. Genauere Beschreibungen zu diesen Modulen sind in den Abschnitten 3.5.2 und 3.5.3 zu finden.

Zusätzlich wurde die Spezifikation `digraph-path` mit der zugehörigen Implementierung `dfs-temp` erstellt. Diese stellt einen Hilfssatz dar, der die Anzahl der erreichbaren Knoten von einem gegebenen Knoten (z.B. dem Startknoten) spezifiziert. Damit konnte eine Vereinfachung der Terminationsbeweise in den Modulen der Tiefensuche erreicht werden. Details dazu werden im Abschnitt 3.4 erläutert.

Zusammenfassend ist der komplette Projektgraph in Abb. 2.2 dargestellt.

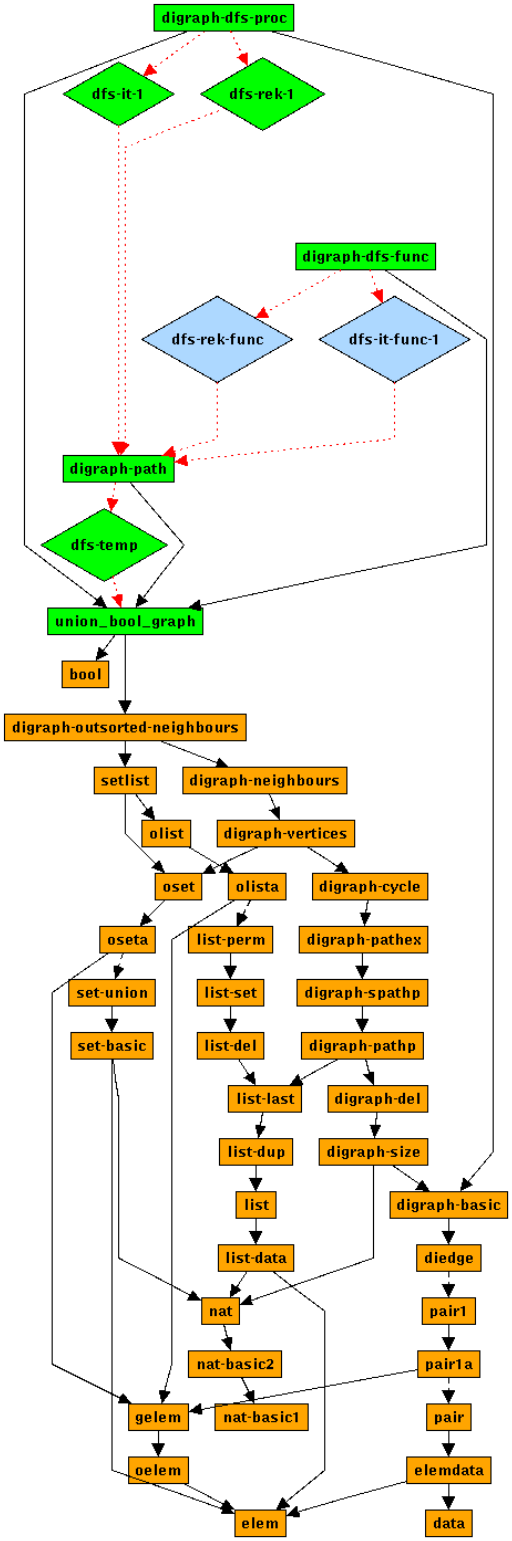


Abbildung 2.2: Projektgraph KIV

# 3
# Spezifikation und Module

## 3.1 Spezifikation `digraph-dfs-proc`

Ausgehend von den Überlegungen in Kapitel 2 soll zunächst eine Tiefensuche als Prozedur umgesetzt werden. Dies bedeutet, dass die Rückgabe des Programmes ein Wahrheitswert ist. Enthält der Pfad G einen Weg zum Startknoten zur Zielknotenmenge, so wird TRUE zurück geliefert, anderfalls FALSE. Im folgenden Absatz sieht man die Signatur der Spezifikation:

digraph-dfs-proc =
**enrich** digraph-basic, union_bool_graph **with**
**predicates** deptSearchP : elem × list × digraph;

**axioms**

$\vdash$ z = [] $\wedge$ a $\in$_v g $\vee$ a ' = [] $\rightarrow$ (deptSearchP(a, z, g) $\leftrightarrow$ false);

$\vdash$ pathp(x, g) $\wedge$ x.first = a $\wedge$ x.last = b $\rightarrow$ (deptSearchP(a, b ', g) $\leftrightarrow$ true);

$\vdash$ a –> b $\in$_e g $\wedge$ b $\in$ z $\rightarrow$ (deptSearchP(a, z, g) $\leftrightarrow$ true);

$\vdash$ a –> b $\in$_e g $\wedge$ b –> c $\in$_e g $\wedge$ c $\in$ z $\rightarrow$ (deptSearchP(a, z, g) $\leftrightarrow$ true);

**end enrich**

Die Spezifikation erweitert hierbei die vorhandenen Spezifikationen digraph-basic sowie union_bool_graph, welche Hilfssätze enthalten und im Abschnitt 3.4 erläutert werden. Die in der Spezifikation definierte Prozedur deptSearchP bildet dabei drei Eingabeparameter elem × list × digraph auf einen Wahrheitswert ab. Dabei bezeichnet „elem“ den Startknoten, „list“ die Menge der Zielknoten und „digraph“ den Graphen. Des Weiteren werden in der Spezifikation verschiedene Axiome deklariert:

- Das erste Axiom beschreibt den Fall, dass unzulässige Parameter beim Prozeduraufruf übergeben werden. Dies ist der Fall, wenn keine Zielknoten angegeben werden oder a ein leerer Knoten ist. In diesem Fall existiert kein Pfad im Graph und somit soll die Prozedur FALSE zurückgeben.
- Das zweite Axiom nimmt Bezug auf die Signatur pathp, welche in der Graphenbibliothek implementiert wurde und überprüft, ob es einen Pfad x im Graphen g gibt. Ist dies der Fall und der Anfangsknoten des Pfades stimmt mit dem Startknoten a über ein und der letzte Knoten des Pfades ist ein Element der Zielknotenmenge, dann liefert die Prozedur deptSearchP TRUE zurück.
- Im dritten Axiom wird der Bezug zum Begriff der Kante im Graphen gebildet. Wenn es eine Kante von Knoten a nach Knoten b im Graphen g gibt und b in der Zielknotenmenge liegt, dann muss die Methode deptSearchP diesen Weg finden. Dieses Axiom ist ein Spezialfall des vierten Axioms.
- Im vierten Axiom wird der Begriff der Kante um die transitive Eigenschaft erweitert. Unsere Prozedur muss also auch dann einen Pfad finden bzw. TRUE zurückliefern, wenn es von Knoten a über einen anderen Knoten b einen Weg in die Zielknotenmenge gibt. Setzt man in diesem Fall b = c, hat man den Spezialfall, welcher in Axiom 3 beschrieben wurde.

## 3.2 Modul `dfs-it-1`

Das Modul „dfs-it-1“ implementiert die Spezifikation „digraph-dfs-proc“, also die Tiefensuche als Prozedur mit der Rückgabe eines Wahrheitswertes. Die genaue Implementierung ist unten aufgezeigt. Allgemein kann gesagt werden, dass zunächst die fehlerhaften Eingaben abgefangen werden, bevor der eigentliche Algorithmus zum Einsatz kommt. Dabei wird in einer Schleife so lange ein neuer Knoten expandiert, bis entweder ein Pfad gefunden wurde, oder alle erreichbaren Knoten abgearbeitet wurden. Für letzteren Fall wurde die Hilfsfunktion „getM“ eingeführt, welche im Abschnitt 3.4 noch genauer erläutert wird. In der while-Schleife wird wie erwähnt ein Knoten von der open-Liste

untersucht und geschaut, ob von ihm erreichbare Knoten noch nicht auf der close-Liste sind. Dazu wird die Funktion „outsortSuccs“ verwendet, welche aus der Graphenbibliothek stammt.
Unterhalb der Deklaration findet man noch einmal die Axiome. Hierbei wird aufgelistet, in wie weit die Axiome mit der Implementierung bewiesen werden konnten und wie die Abhängigkeiten untereinander sind.

### decl

deptSearchP#(a, z, g; **var** bbool)
{
**let** open = [] + a, close = [], $n_0 = 0$, $a_0$ = a **in**
**{**
  bbool := false;
  $n_0$ := getM(a, g) + 1;
  **if** ¬ (a ' = [] ∨ z = []) **then**
    **if** a ∈ z **then** bbool := true **else**
    **{**
      **while** $n_0 > 0$ ∧ (bbool ↔ false) ∧ (open ≠ []) **do**
        **{**
          $a_0$ := open.first;
          **if** $a_0$ ∈ z **then** bbool := true **else**
          **{**
            $n_0$ := $n_0$ - 1;
            close := $a_0$ ' + close;
            open := set2list(outsortSuccs($a_0$, list2set(close), g)) + open.rest
          **}**
        **};**
      **if** $n_0 = 0$ **then** bbool := false
    **}**
**}**
}

### Term-deptsearchpjsiz

⊢ ⟨deptSearchP#⟩true

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : decl

- used by : Exp-02, Exp-03, Exp-04
- interactions : 47
- proof steps : 47

### Exp-01

$\vdash z = [] \wedge a \in\_v\ g \vee a\ ' = [] \rightarrow (\langle deptSearchP\#\rangle bbool \leftrightarrow false)$

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : decl
- interactions : 0
- proof steps : 8

### Exp-02

$\vdash pathp(x, g) \wedge x.first = a \wedge x.last = b \rightarrow (\langle deptSearchP\#\rangle bbool \leftrightarrow true)$

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchpjsiz
- interactions : 1
- proof steps : 2

### Exp-03

$\vdash a -> b \in\_e\ g \wedge b \in z \rightarrow (\langle deptSearchP\#\rangle bbool \leftrightarrow true)$

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchpjsiz
- interactions : 3

- proof steps : 4

### Exp-04

$\vdash$ a –> b $\in$_e g $\wedge$ b –> c $\in$_e g $\wedge$ c $\in$ z $\rightarrow$ ($\langle$deptSearchP#$\rangle$bbool $\leftrightarrow$ true)

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchpjsiz
- interactions : 1
- proof steps : 2

Wie zu sehen ist, konnten alle Axiome bewiesen werden. Dabei wurde bei den Axiomen 2 bis 4 auf den Terminationsbeweis zurück gegriffen. In folgendem Abschnitt sind noch einmal die bei den Beweisen verwendeten Beweisschritte aufgeführt.

Statistic for each theorem:

| | proof steps | interactions | automation | induction |
|---|---|---|---|---|
| Term-deptsearchpjsiz | 47 | 47 | 0.0 % | nat |
| Exp-01 | 8 | 0 | 100.0 % | — |
| Exp-03 | 4 | 3 | 25.0 % | — |
| Exp-04 | 2 | 1 | 50.0 % | — |
| Exp-02 | 2 | 1 | 50.0 % | — |

Heuristic applications:

| heuristic | total applications | percentage |
|---|---|---|
| Interactive | 52 | 82.5 % |
| symbolic execution | 5 | 7.9 % |
| System | 4 | 6.3 % |
| simplifier | 1 | 1.5 % |
| weak unfold | 1 | 1.5 % |

Rule applications:

| rule | total applications | percentage |
|---|---|---|
| assign right | 13 | 20.6 % |
| simplifier | 8 | 12.6 % |

| | | |
|---|---|---|
| if right | 6 | 9.5 % |
| normalize | 4 | 6.3 % |
| insert lemma | 4 | 6.3 % |
| skip right | 4 | 6.3 % |
| assign left | 3 | 4.7 % |
| while exit right | 3 | 4.7 % |
| insert elim lemma | 2 | 3.1 % |
| split left | 2 | 3.1 % |
| while right | 1 | 1.5 % |
| case distinction | 1 | 1.5 % |
| cut formula | 1 | 1.5 % |
| skip left | 1 | 1.5 % |
| apply induction | 1 | 1.5 % |
| if negative left | 1 | 1.5 % |
| induction | 1 | 1.5 % |
| call left | 1 | 1.5 % |
| weakening | 1 | 1.5 % |
| vardecls left | 1 | 1.5 % |
| split right | 1 | 1.5 % |
| call right | 1 | 1.5 % |
| vardecls right | 1 | 1.5 % |
| execute while | 1 | 1.5 % |

Lemma applications:

| theorem | total applications | percentage |
|---|---|---|
| Term-deptsearchpjsiz | 4 | 66.6 % |
| decl | 2 | 33.3 % |

## 3.3 Modul `dfs-rek-1`

### decl-01

deptSearchP#(a, z, g; **var** bbool)
{
**let** n = 0 **in** {n := getM(a, g); deptSearchProc#}
}

- Type: an axiom
- no proof exists

- used by : Term-deptsearchpjsiz, Exp-01

### decl-03

deptSearchProc#(open, close, z, g, n; **var** bbool)
{
bbool := false;
**if** ¬ (open = [] ∨ z = [] ∨ n = 0) **then**
  **let** a = open.first **in**
  **if** a ∈ z **then** bbool := true **else**
  **{**
    close := a ' + close;
    open := set2list(outsortSuccs(a, list2set(close), g)) + open.rest;
    n := n - 1;
    **if** n ≠ 0 **then** deptSearchProc#
  **}**
}

- Type: an axiom
- no proof exists
- used by : Exp-01, Term-deptsearchpjsiz-01

### Term-deptsearchpjsiz

⊢ ⟨deptSearchP#⟩true

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchpjsiz-01, decl-01
- used by : Exp-02, Exp-03, Exp-04
- interactions : 5
- proof steps : 5

### Exp-01

⊢ z = [] ∧ a ∈_v g ∨ a ' = [] → (⟨deptSearchP#⟩bbool ↔ false)

- Type: a proof obligation

- must be proved
- a complete proof exists
- used lemmas : decl-03, decl-01
- interactions : 1
- proof steps : 10

### Exp-02

$\vdash$ pathp(x, g) $\wedge$ x.first = a $\wedge$ x.last = b $\rightarrow$ ($\langle$deptSearchP#$\rangle$bbool $\leftrightarrow$ true)

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchpjsiz
- interactions : 1
- proof steps : 2

### Exp-03

$\vdash$ a –> b $\in$_e g $\wedge$ b $\in$ z $\rightarrow$ ($\langle$deptSearchP#$\rangle$bbool $\leftrightarrow$ true)

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchpjsiz
- interactions : 3
- proof steps : 4

### Exp-04

$\vdash$ a –> b $\in$_e g $\wedge$ b –> c $\in$_e g $\wedge$ c $\in$ z $\rightarrow$ ($\langle$deptSearchP#$\rangle$bbool $\leftrightarrow$ true)

- Type: a proof obligation
- must be proved
- a complete proof exists

- used lemmas : Term-deptsearchpjsiz
- interactions : 1
- proof steps : 2

### Term-deptsearchpjsiz-01

$\vdash \langle$deptSearchProc#$\rangle$true

- Type: created by user
- a complete proof exists
- used lemmas : decl-03
- used by : Term-deptsearchpjsiz
- interactions : 14
- proof steps : 14

Statistic for each theorem:

| | proof steps | interactions | automation | induction |
|---|---|---|---|---|
| Term-deptsearchpjsiz-01 | 14 | 14 | 0.0 % | nat |
| Exp-01 | 10 | 1 | 90.0 % | — |
| Term-deptsearchpjsiz | 5 | 5 | 0.0 % | — |
| Exp-03 | 4 | 3 | 25.0 % | — |
| Exp-04 | 2 | 1 | 50.0 % | — |
| Exp-02 | 2 | 1 | 50.0 % | — |

Heuristic applications:

| heuristic | total applications | percentage |
|---|---|---|
| Interactive | 25 | 67.5 % |
| symbolic execution | 5 | 13.5 % |
| System | 4 | 10.8 % |
| simplifier | 2 | 5.4 % |
| weak unfold | 1 | 2.7 % |

Rule applications:

| rule | total applications | percentage |
|---|---|---|
| assign right | 6 | 16.2 % |
| insert lemma | 5 | 13.5 % |

| | | |
|---|---|---|
| normalize | 4 | 10.8 % |
| if right | 3 | 8.1 % |
| call left | 2 | 5.4 % |
| skip right | 2 | 5.4 % |
| simplifier | 2 | 5.4 % |
| call right | 2 | 5.4 % |
| assign left | 2 | 5.4 % |
| vardecls right | 2 | 5.4 % |
| weakening | 1 | 2.7 % |
| induction | 1 | 2.7 % |
| if negative left | 1 | 2.7 % |
| apply induction | 1 | 2.7 % |
| vardecls left | 1 | 2.7 % |
| case distinction | 1 | 2.7 % |
| skip left | 1 | 2.7 % |

Lemma applications:

| theorem | total applications | percentage |
|---|---|---|
| Term-deptsearchpjsiz | 4 | 44.4 % |
| decl-03 | 2 | 22.2 % |
| decl-01 | 2 | 22.2 % |
| Term-deptsearchpjsiz-01 | 1 | 11.1 % |

## 3.4 „Hilfssätze“ - Spezifikation und Module

Bei der Bearbeitung des Projektes und der Beweisführung mussten verschiedene Ansätze aus der Graphenbibliothek zusammengeführt sowie Teilbeweise ausgelagert werden, da sie in mehreren Beweisen notwendig waren.
Zunächst werde die Spezifikation „union_bool_graph“ betrachtet, welche die notwendigen Pakete der Graphenbibliothek zusammenfasst. Folgenden die verwendete Spezifikation:

union_bool_graph = digraph-outsorted-neighbours + bool

Eine weitere Spezifikation ist „digraph-path“, in welcher die in Kapitel 3.1 erwähnte Signatur „getM“ definiert wird.

digraph-path =
**enr¯ich** union_bool_graph **with**
**functions** getM : elem × digraph → nat ; **axioms**

getM(a, @g) = 0;

getM(a, @g +v a) = 1;

¬ b ∈_v g ∧ a ∈_v g → getM(a, g +e a –> b) = getM(a, g) + 1;

**end enrich**

Die Methode „getM“ bekommt sowohl einen Knoten a sowie einen Graphen g übergeben und gibt eine Zahl m zurück. Diese bezeichnet hierbei die Anzahl der von dem Knoten a im Graphen g erreichbaren Knoten, wobei sowohl direkt erreichbare als auch über transitive Kanten erreichbare berücksichtigt werden. Auch hier werden wieder drei Axiome definiert:

- Axiom 1 gibt dann, dass in einem leeren Graphen kein Knoten von a erreicht werden kann.
- In Axiom 2 wird der leere Graph um einen Knoten a erweitert. Da Anzahl der von a erreichbaren Knoten ist in diesem Fall 1.
- Im dritten Axiom wird ein Knoten b, welche noch nicht im Graphen lag sowie eine Kante vom Knoten an zu Knoten b hinzugefügt. Die Anzahl der erreichbaren Knoten entspricht nun der Anzahl der erreichbaren Knoten ohne den Knoten b plus 1.

Die Spezifikation wird durch das Modul „dfs-temp“ implementiert. Folgend ist die Implementierung der Funktion „getM“ zu sehen.

```
getM#(a, g; var n)
{
let s20 = ∅, b99 = a in
{
   n := 0;
   s20 := g.vs;
   while s20 ≠ ∅ do
      {b99 := s20.max; s20 := s20.butmax; if path∃(a, b99, g) then n := n + 1}
}
}
```

Für einen gegebenen Knoten werden in einer Schleife alle anderen Knoten des Graphen betrachtet. Mit der Funktion „path∃“ aus der Graphenbibliothek wird dann überprüft, ob es einen Pfad zwischen den beiden Knoten gibt. Wenn ja, wird die Zählvariable erhöht.
Die Beweise der Axiome gelangen alle, wie sich in folgendem Abschnitt sehen lässt.

Term-getmjsiz: ⊢ ⟨getM#⟩ true

uses: (decl)
used by: (Exp-01 Exp-03)
Proof steps: 50, interactions: 50

Exp-01:

$\vdash$

$\neg$ b $\in$_v g $\wedge$ a $\in$_v g $\rightarrow$ $\langle$getM#$\rangle\langle$getM#$\rangle$m $= m_0 + 1$

uses: (Term-getmjsiz)
Proof steps: 2, interactions: 1

Exp-02: $\vdash$ $\langle$getM#$\rangle$ m $= 0$

uses: (decl)
Proof steps: 6, interactions: 6

Exp-03: $\vdash$ $\langle$getM#$\rangle$ m $= 1$

uses: (Term-getmjsiz)
Proof steps: 1, interactions: 1

Auch hier sei wieder auf die verwendeten Beweisschritte hingewiesen, welche in den folgenden Tabellen aufgeführt sind:

Statistic for each theorem:

| | proof steps | interactions | automation | induction |
|---|---|---|---|---|
| Term-getmjsiz | 50 | 50 | 0.0 % | nat |
| Exp-02 | 6 | 6 | 0.0 % | — |
| Exp-01 | 2 | 1 | 50.0 % | — |
| Exp-03 | 1 | 1 | 0.0 % | — |

Heuristic applications:

| heuristic | total applications | percentage |
|---|---|---|
| Interactive | 57 | 98.2 % |
| System | 1 | 1.7 % |

Rule applications:

| rule | total applications | percentage |
|---|---|---|
| insert rewrite lemma | 14 | 24.1 % |

| | | |
|---|---|---|
| cut formula | 9 | 15.5 % |
| assign right | 8 | 13.7 % |
| simplifier | 6 | 10.3 % |
| weakening | 3 | 5.1 % |
| call right | 2 | 3.4 % |
| split right | 2 | 3.4 % |
| vardecls right | 2 | 3.4 % |
| case distinction | 2 | 3.4 % |
| if right | 1 | 1.7 % |
| while exit right | 1 | 1.7 % |
| all right | 1 | 1.7 % |
| normalize | 1 | 1.7 % |
| invariant right | 1 | 1.7 % |
| insert lemma | 1 | 1.7 % |
| skip right | 1 | 1.7 % |
| exists left | 1 | 1.7 % |
| induction | 1 | 1.7 % |
| all left | 1 | 1.7 % |

Lemma applications:

| theorem | total applications | percentage |
|---|---|---|
| decl | 2 | 66.6 % |
| Term-getmjsiz | 1 | 33.3 % |

## 3.5 Tiefensuche als Funktion

### 3.5.1 Spezifikation `digraph-dfs-func`

Im Gegensatz zu den vorangegangenen Kapiteln, bei dem die Tiefensuche als Prozedur spezifiziert und implementiert wurde, wird nun der Algorithmus als Funktion umgesetzt. Das heißt, dass sowohl die iterative als auch die rekursive Variante einen Rückgabeparameter besitzen, der einen Pfad vom Startknoten a in die Zielknotenmenge z enthält, wenn dieser im Graphen G existiert. Andernfalls wird die leere Liste zurückgegeben.

Der folgende Absatz zeigt die Signatur der Spezifikation:

digraph-dfs-func =
**enrich** union_bool_graph **with**
    **functions** deptSearchF : elem × list × digraph → list ;

**axioms**

z = [] ∧ a ∈_v g ∨ a ' = [] → deptSearchF(a, z, g) = [];

pathp(x, g) ∧ x.first = a ∧ x.last = b → deptSearchF(a, b ', g) = x;

a –> b ∈_e g ∧ b ∈ z → deptSearchF(a, z, g).first = a;

a –> b ∈_e g ∧ b ∈ z → deptSearchF(a, z, g).last = b;

a –> b ∈_e g ∧ b –> c ∈_e g ∧ c ∈ z → deptSearchF(a, z, g).first = a;

a –> b ∈_e g ∧ b –> c ∈_e g ∧ c ∈ z → deptSearchF(a, z, g).last = c;

**end enrich**

Die Spezifikation erweitert die Spezifikation *union_ bool_ graph*, welche in Abschnitt 3.4 erläutert wird. Innerhalb von *digraph-dfs-func* wird die Funktion *deptSearchF* beschrieben, welche eine Abbildung von *elem* × *list* × *digraph* → *list* ist. Die definierten Axiome werden im Folgenden entsprechend der obigen Ordnung erklärt:

- Das erste Axiom besagt, dass wenn der Funktion entweder keine Zielknotenmenge oder kein Startelement übergeben wird, diese einen leeren Pfad zurückgibt, sprich die leere Liste.
- Das zweite Axiom stellt den Fall dar, dass es im Graphen eine Kante gibt, deren Startknoten a und deren Endknoten b ist. Wird die Funktion in der Form aufgerufen, dass a der Startknoten und b der Zielknoten ist, so ist der Rückgabepfad die Kante x.
- Axiom drei und vier erfüllen dieselben Voraussetzungen wie zwei, mit dem Unterschied, dass die Aussage von Axiom 3 ist, dass der Start des Rückgabepfades a ist und in Axiom vier das Ende Pfades Knoten b.
- Axiom fünf bildet die Transitivität ab, wobei gesagt wird, dass wenn eine Kante von a nach b und von b nach c existiert, wobei c Zielknoten ist, der Knoten a das erste Element auf dem Rückgabepfad ist.
- Analog zu Axiom fünf, ist die Aussage das auf dem transitiven Pfad das letzte Element der Zielknoten c ist.

### 3.5.2 Modul `dfs-rek-func`

Das Modul *dfs-rek-func* beinhaltet die rekursive Implementierung der Tiefensuchefunktion. Dabei setzt sich das Programm aus einer Startmethode und der eigentlichen rekursiven Funktion zusammen. Zunächst wird (analog zur Prozedur - siehe Abschnitt 3.3) in der Startmethode die Anzahl der erreichbaren Knoten durch die Hilfsfunktion

*getM( . , . )* errechnet. Anschließend kommt es zur eigentlichen Ausführung der rekursiven Funktion. Nach den Initialisierungen und den Tests der übergebenen Parameter auf Korrektheit (alle Variablen belegt, keine Null-Werte), wird in jedem Durchlauf für den aktuell betrachteten Knoten, der Graph expandiert und die neu gefundenen Knoten einzeln weiterverfolgt. Besonderheit bei dieser Variante als Funktion ist, dass mittels einer Schleife ein Backtracking realisiert wird, welches die Exploration des Graphen auf einfache Weise ermöglicht. Dabei wird die Gegebenheit ausgenutzt, dass durch das rekursive Verhalten, korrekte Teilstücke des bereits gefundenen Weges stets wiederverwendet werden können und bei Verzweigungen innerhalb der Schleife alle Wege durchprobiert werden.

Die genaue Deklaration zeigt folgender Absatz, der außerdem die Axiome und ihre Abhängigkeiten auflistet:

### decl-01

deptSearchF#(a, z, g; **var** y)
{
**let** path = [], n = 0, $n_0$ = 0 **in**
**{**
  n := getM(a, g);
  $n_0$ := n;
  deptSearchProcF#;
  y := path
**}**
}

- Type: an axiom
- no proof exists
- used by : Term-deptsearchfjsiz

### decl-03

deptSearchProcF#(a, close, z, y, g, n; **var** path, $n_0$)
{
path := [];
$n_0$ := n;
**let** open = a ' **in**
**if** ¬ (open = [] ∨ z = [] ∨ n = 0) **then**
  **{**
    y := y + a ';

```
      if a ∈ z then path := y else
      {
        close := a ' + close;
        open := set2list(outsortSuccs(a, list2set(close), g)) + open.rest;
        n := n - 1;
        n_0 := n;
        while ¬ (n = 0 ∨ open = [] ∨ path ≠ []) do
          {
            deptSearchProcF#;
            n := n_0;
            open := open.rest
          }
      }
  }
}
```

- Type: an axiom
- no proof exists
- used by : Term-deptsearchfjsiz-01

### Term-deptsearchfjsiz

$\vdash \langle$deptSearchF#$\rangle$true

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz-01, decl-01
- used by : Exp-01, Exp-02, Exp-03, Exp-04, Exp-05, Exp-06
- interactions : 5
- proof steps : 5

### Exp-01

$\vdash$ z = [] $\wedge$ a $\in$_v g $\vee$ a ' = [] $\rightarrow \langle$deptSearchF#$\rangle$open = []

- Type: a proof obligation
- must be proved

- a complete proof exists
- used lemmas : Term-deptsearchfjsiz
- interactions : 1
- proof steps : 3

### Exp-02

$\vdash$ pathp(x, g) $\wedge$ x.first = a $\wedge$ x.last = b $\rightarrow$ $\langle$deptSearchF#$\rangle$open = x

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz
- interactions : 2
- proof steps : 3

### Exp-03

$\vdash$ a –> b $\in$_e g $\wedge$ b $\in$ z $\rightarrow$ $\langle$deptSearchF#$\rangle$open.first = a

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz
- interactions : 2
- proof steps : 3

### Exp-04

$\vdash$ a –> b $\in$_e g $\wedge$ b $\in$ z $\rightarrow$ $\langle$deptSearchF#$\rangle$open.last = b

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz

- interactions : 2
- proof steps : 3

### Exp-05

$\vdash$ a –> b $\in$_e g $\wedge$ b –> c $\in$_e g $\wedge$ c $\in$ z $\rightarrow$ $\langle$deptSearchF#$\rangle$open.first = a

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz
- interactions : 2
- proof steps : 3

### Exp-06

$\vdash$ a –> b $\in$_e g $\wedge$ b –> c $\in$_e g $\wedge$ c $\in$ z $\rightarrow$ $\langle$deptSearchF#$\rangle$open.last = c

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz
- interactions : 2
- proof steps : 3

### Term-deptsearchfjsiz-01

$\vdash$ $\langle$deptSearchProcF#$\rangle\langle$y := $x_0\rangle$true

- Type: created by user
- a partial proof exists
- used lemmas : decl-03
- used by : Term-deptsearchfjsiz
- interactions : 23
- proof steps : 24

In diesem Fall konnten bis auf einen Teil des Terminationsbeweises alle Axiome bewiesen werden. Die einzelnen Statistiken zu den Beweisschritten werden nachfolgend aufgelistet:

Statistic for each theorem:

| | proof steps | interactions | automation | induction |
|---|---|---|---|---|
| Term-deptsearchfjsiz-01 | 24 | 23 | 4.1 % | — |
| Term-deptsearchfjsiz | 5 | 5 | 0.0 % | — |
| Exp-06 | 3 | 2 | 33.3 % | — |
| Exp-01 | 3 | 1 | 66.6 % | — |
| Exp-05 | 3 | 2 | 33.3 % | — |
| Exp-02 | 3 | 2 | 33.3 % | — |
| Exp-04 | 3 | 2 | 33.3 % | — |
| Exp-03 | 3 | 2 | 33.3 % | — |

Heuristic applications:

| heuristic | total applications | percentage |
|---|---|---|
| Interactive | 16 | 69.5 % |
| System | 6 | 26.0 % |
| simplifier | 1 | 4.3 % |

Rule applications:

| rule | total applications | percentage |
|---|---|---|
| insert lemma | 7 | 30.4 % |
| normalize | 6 | 26.0 % |
| simplifier | 6 | 26.0 % |
| assign right | 2 | 8.6 % |
| vardecls right | 1 | 4.3 % |
| call right | 1 | 4.3 % |

Lemma applications:

| theorem | total applications | percentage |
|---|---|---|
| Term-deptsearchfjsiz | 6 | 75.0 % |
| decl-01 | 1 | 12.5 % |
| Term-deptsearchfjsiz-01 | 1 | 12.5 % |

### 3.5.3 Modul `dfs-it-func-1`

Im Modul *dfs-it-func-1* wird die Tiefensuchefunktion iterativ implementiert. Wesentlich für die Verarbeitung ist hierbei, dass die maximale Anzahl der erreichbaren Knoten in der Variablen n0 gespeichert wird. Um dies zu bestimmen, wird die Hilfsfunktion *getM( . , . )* benutzt. Außerdem werden zu Beginn weitere Initialisierungen und Korrektheitsprüfungen der übergebenen Parameter durchgeführt. Die Kernfunktionalität leistet die äußere Schleife, die nach maximal n0-Schritten abbricht und den Graphen durchläuft. Besonders für die Funktion ist nun, dass um am Ende einen Rückgabewert zu liefern, ein weiterer Graph erzeugt wird, der zunächst leer ist, im Verlaufe der Abarbeitung jedoch jeweils um die expandierten Knoten erweitert wird. Der Graph dient der Speicherung der Vorgängerbeziehung zwischen den explorierten Knoten, so dass für den Fall, dass der Zielknoten gefunden wird, der Algorithmus nachverfolgen kann, wie der Weg vom Startknoten bis dahin möglich war. Die Rückverfolgung sowie die Erzeugung dieses Hilfsgraphen, wird im Körper der inneren Schleifen durchgeführt.

Die Deklaration und die benutzten Axiome werden im folgenden Absatz aufgeführt:

**decl**

```
deptSearchF#(a, z, g; var y)
{
let open = [] + a,
    close = [],
    n_0 = 0,
    vorgaenger = @g,
    open_2 = [],
    start = a,
    n = 0,
    n_1 = 0,
    n_2 = 0
in
{
   y := [];
   n_0 := getM(a, g);
   n := n_0;
   n_1 := n_0;
   n_2 := n_0;
   if ¬ (a ' = [] ∨ z = []) then
      if a ∈ z then y := y + a else
      while n_0 > 0 ∧ open ≠ [] do
          {
```

```
        a := open.first;
        n0 := n0 - 1;
        if a ∈ z then
          {
            y := y + a;
            while a ≠ start ∧ n1 > 0 do
              {
                y := y + set2list(outsortSuccs(a, list2set(y), vorgaenger));
                a := y.last;
                n1 := n1 - 1
              }
          }
        else
        {
          close := a ' + close;
          open2 := set2list(outsortSuccs(a, list2set(close), g));
          open := open2 + open.rest;
          while open2 ≠ [] ∧ n2 > 0 do
            {
              vorgaenger := vorgaenger +e open2.first –> a;
              open2 := open2.rest;
              n2 := n2 - 1
            }
        }
      }
}
}
```

- Type: an axiom
- no proof exists
- used by : Term-deptsearchfjsiz, Exp-01

### Term-deptsearchfjsiz

$\vdash \langle \text{deptSearchF\#} \rangle \text{true}$

- Type: a proof obligation
- must be proved
- a partial proof exists

- used lemmas : decl
- used by : Exp-02, Exp-03, Exp-04, Exp-05, Exp-06
- interactions : 26
- proof steps : 39

### Exp-01

$\vdash$ z = [] $\wedge$ a $\in$_v g $\vee$ a ' = [] $\rightarrow$ $\langle$deptSearchF#$\rangle$open = []

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : decl
- interactions : 2
- proof steps : 13

### Exp-02

$\vdash$ pathp(x, g) $\wedge$ x.first = a $\wedge$ x.last = b $\rightarrow$ $\langle$deptSearchF#$\rangle$open = x

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz
- interactions : 1
- proof steps : 2

### Exp-03

$\vdash$ a –> b $\in$_e g $\wedge$ b $\in$ z $\rightarrow$ $\langle$deptSearchF#$\rangle$open.first = a

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz

- interactions : 1
- proof steps : 2

### Exp-04

$\vdash$ a –> b $\in$_e g $\wedge$ b $\in$ z $\rightarrow$ $\langle$deptSearchF#$\rangle$open.last = b

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz
- interactions : 1
- proof steps : 2

### Exp-05

$\vdash$ a –> b $\in$_e g $\wedge$ b –> c $\in$_e g $\wedge$ c $\in$ z $\rightarrow$ $\langle$deptSearchF#$\rangle$open.first = a

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz
- interactions : 1
- proof steps : 2

### Exp-06

$\vdash$ a –> b $\in$_e g $\wedge$ b –> c $\in$_e g $\wedge$ c $\in$ z $\rightarrow$ $\langle$deptSearchF#$\rangle$open.last = c

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchfjsiz
- interactions : 2
- proof steps : 3

Wie für den rekursiven Fall konnte der Terminationsbeweis nicht vollständig abgeschlossen werden. Die restlichen Axiome wurden allerdings bewiesen. Nachfolgend eine Auflistung zu den Beweisschritten:

Statistic for each theorem:

| | proof steps | interactions | automation | induction |
|---|---|---|---|---|
| Term-deptsearchfjsiz | 39 | 26 | 33.3 % | 2 |
| Exp-01 | 13 | 2 | 84.6 % | — |
| Exp-06 | 3 | 2 | 33.3 % | — |
| Exp-02 | 2 | 1 | 50.0 % | — |
| Exp-05 | 2 | 1 | 50.0 % | — |
| Exp-03 | 2 | 1 | 50.0 % | — |
| Exp-04 | 2 | 1 | 50.0 % | — |

Heuristic applications:

| heuristic | total applications | percentage |
|---|---|---|
| Interactive | 34 | 53.9 % |
| symbolic execution | 16 | 25.3 % |
| System | 6 | 9.5 % |
| conditional right split | 4 | 6.3 % |
| simplifier | 2 | 3.1 % |
| weak unfold | 1 | 1.5 % |

Rule applications:

| rule | total applications | percentage |
|---|---|---|
| assign right | 21 | 33.3 % |
| simplifier | 8 | 12.6 % |
| normalize | 6 | 9.5 % |
| insert lemma | 5 | 7.9 % |
| if right | 5 | 7.9 % |
| cut formula | 2 | 3.1 % |
| skip right | 2 | 3.1 % |
| induction | 2 | 3.1 % |
| call right | 2 | 3.1 % |
| while exit right | 2 | 3.1 % |
| vardecls right | 2 | 3.1 % |
| weakening | 2 | 3.1 % |

| | | |
|---|---|---|
| insert elim lemma | 2 | 3.1 % |
| split right | 1 | 1.5 % |
| while right | 1 | 1.5 % |

Lemma applications:

| theorem | total applications | percentage |
|---|---|---|
| Term-deptsearchfjsiz | 5 | 71.4 % |
| decl | 2 | 28.5 % |

# 4

# Zusammenfassung

Unser Ziel im Rahmen des Projektes eine sinnvolle Verwendung bzw. der Graphenbibliothek zu erreichen, konnte unserer Meinung nach voll umgesetzt werden. Mit der Implementierung und dem Beweis der Tiefensuche ist es uns gelungen, einen der gängigen Suchalgorithmen für Graphen zu verifizieren. Dabei ist es uns gelungen, verschiedene Varianten des Suchverfahrens umzusetzen. Die iterative Variante baut dabei vor allem auf die Abarbeitung von Schleifen auf, die rekursive Variante benötigt die Beweise von verschachtelten Funktionsaufrufen. Mit diesen verschiedenen Varianten ist es uns gelungen, einen großen Teil der Möglichkeiten von KIV zu erforschen. Dabei sei aber auch angemerkt, dass KIV ein sehr komplexes und mächtiges Werkzeug ist. Über das bloße Grundverständnis hinaus muss man sich für jeden Anwendungsfall spezifische Lösungsszenarien aneignen und diese geschickt kombinieren. Dies gelang und leider auch nicht in allen Fällen, so dass zumindest ein Beweis nicht abgeschlossen werden konnte. Leider machte uns an einigen Stelle auch KIV selbst die Arbeit schwer, da es in einigen Bereichen nicht so funktioniert, wie man es sich intuitiv erhofft. Alles in allem war das Projekt aber sehr förderlich, um unser Verständnis für die Verifikation von Programmen zu erweitern.

# Abbildungsverzeichnis

# Literaturverzeichnis

[DR08] Drobek, M. ; Redlich, D.: *Verifikation des Graphenbegriffs und Umsetzung kleinerer Graphenmodule.* Februar 2008