Fortgeschrittene Techniken der funktionalen Programmierung
Wintersemester 2005/2006

FGL/Haskell - A Functional Graph Library for Haskell

von Christoph Stoike

Betreuer:
Bernd Braßel

Literatur:

- Marting Erwig. *Inductive Graphs and Functional Graph Algorithms.* Journal of Functional Programming, Vol. 11, No. 5, 467-492, 2001
- http://web.engr.oregonstate.edu/~erwig/fgl/haskell/
- Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press

Inhaltsverzeichnis

1 Einleitung **2**

2 Induktive Graphen **2**
2.1 Induktive Graphdefinition 2
2.2 Pattern-Matching auf Graphen 5
2.2.1 Pattern-Matching 5
2.2.2 Termrepräsentationen und active patterns 8
2.3 Implementierung und Komplexität 10
2.3.1 Graph-Repräsentationen und Persistenz 10
2.3.2 Repräsentation durch binäre Suchbäume 11

3 Funktionale Graphalgorithmen **12**
3.1 Tiefensuche 12
3.2 Breitensuche 17
3.3 Dijkstra-Algorithmus 20
3.4 Maximal unabhängige Knotenmengen 23

4 Zusammenfassung **24**

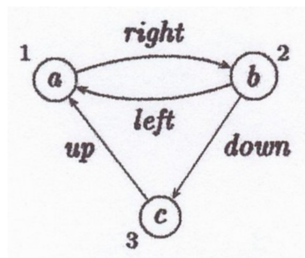
1 Einleitung

Graphen und Graphalgorithmen spielen in der Informatik eine wichtige Rolle. Ihre Umsetzung erfolgt meist mit Hilfe imperativer Sprachkonzepte. Dennoch mag es nahe liegen, dass dieses auch funktional gelingen kann. Die eigentliche Frage ist aber, ob dies so zu schaffen ist, dass die für funktionale Programme so typische Eleganz und Einfachheit auf der einen und die asymptotische Komplexität der imperativen Variante auf der anderen Seite nicht verloren gehen. Mit der Lösung dieser Frage beschäftigt sich diese Seminararbeit.
Dabei wird zunächst die Definition einer Datenstruktur für Graphen in Haskell vorgestellt. Anschließend soll demonstriert werden, wie man aufbauend auf dieser Definition verschiedene Graphalgorithmen effizient implementieren kann.
Die hier angegebenen Definitionen von Datenstrukturen und Funktionen basieren auf Martin Erwigs Arbeit *Inductive Graphs and Functional Graph Algorithms* und sind Bestandteil der darauf aufbauenden *Functional Graph Library (FGL)*, einer Haskell-Bibliothek für Graphen und Graphalgorithmen.

2 Induktive Graphen

2.1 Induktive Graphdefinition

Üblicherweise wird ein Graph als ein Paar $G = (V, E)$ mit einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$ definiert. Hierbei unterscheidet man gerichtete und ungerichtete Graphen und kann sowohl Knoten als auch Kanten mit einer Beschriftung, einem Label, versehen. Wir wollen uns im Folgenden mit dem allgemeinsten Fall beschäftigen, also einem gerichteten, mit Knoten- und Kantenlabeln versehenen Multigraphen (Mehrfachkanten sind also möglich). Dann sind zum Beispiel ungerichtete Graphen als symmetrische gerichtete Graphen darstellbar. Labellose Graphen entsprechen dann einfach Graphen mit leerem Label. Ferner nehmen wir an, dass Knoten durch ganze Zahlen repräsentiert werden.
Nachfolgend ist nun ein Graph angegeben, wie wir ihn behandeln wollen:



Betrachten wir nun Graphen aus der induktiven Sichtweise. Dann ist ein solcher entweder der leere Graph oder ein Graph, der durch einen Knoten und Kanten, die diesen mit dem bisherigen Graphen verbinden, erweitert wird.
Ein Graph G ist also:

- der leere Graph
- ein Graph G', erweitert durch:
  - einen Knoten v mit einem Label
  - Kanten, die von Knoten aus G' zu v führen, mit einem Label
  - Kanten, die von v zu Knoten in G' führen, mit einem Label

Wir wollen diese Definition nun in Haskell überführen. Dazu definieren wir zunächst drei neue Typen:

1. Knoten entsprechen Integer-Werten:

   $$\textbf{type}\ Node\ =\ Int$$

2. Die Vorgänger bzw. Nachfolger eines Knotens v werden durch Adjazenzlisten repräsentiert, in denen die einzelnen Kanten als Tupel repräsentiert werden, die ein Kantenlabel des Typs b und die Knotennummer des Vorgängers bzw. Nachfolgers von v enthalten:

   $$\textbf{type}\ Adj\ b\ =\ [(b, Node)]$$

3. Nun müssen den einzelnen Knoten noch ihre jeweiligen Vorgänger- und Nachfolgerlisten zugeordnet werden, was in folgendem Typ geschieht, der als Tupel (Vorgängerliste, Knotennummer, Knotenlabel vom Typ a, Nachfolgerliste) definiert ist:

   $$\textbf{type}\ Context\ a\ b\ =\ (Adj\ b,\ Node,\ a,\ Adj\ b)$$

Der Typ *Context* enthält also gerade die Information, die benötigt wird, um einen Graphen um einen Knoten samt Kanten zu erweitern. Dies führt uns zur Definition einer Datenstruktur für Graphen. Der Graphtyp selbst ist dabei als abstrakter Datentyp definiert.
Konstruiert wird ein Graph mittels *Empty*, wodurch der leere Graph dargestellt wird, und der Funktion &, die für die Erweiterung eines bestehenden Graphen durch einen *Context*, also durch einen Knoten und dessen Kanten, sorgt:

$$\textbf{data}\ Graph\ a\ b\ =\ Empty\ |\ Context\ a\ b\ \&\ Graph\ a\ b$$

Bei dieser Implementierung von Graphen als abstrakten Datentypen tritt jedoch ein Problem auf. Denn die Tatsache, dass & eine Funktion und kein Konstruktor ist, sorgt dafür, dass & in Patterns nicht genutzt werden kann. Als Lösung definiert FGL neben dem Prädikat *isEmpty* folgende Funktion, die aus einem Graphen einen beliebigen Context herauszieht:

$$matchAny \;::\; Graph\ a\ b \rightarrow (Context\ a\ b,\ Graph\ a\ b)$$

Anzumerken ist hierbei, dass *matchAny* eine Fehlermeldung hervorruft, wenn es auf einen leeren Graphen angewendet wird.
Der Einfachheit halber wollen wir aber bei der &-Schreibweise bleiben.
Nun ist es uns möglich, den oben vorgestellten Beispielgraphen mittels eines Terms darzustellen. Der induktive Aufbau erfolgt folgendermaßen:
Beginnend mit dem leeren Graphen *Empty* wird ein Knoten, sagen wir die Nummer 3 mit dem Label *c*, hinzugefügt, d.h.:

$$([\,],\ 3,\ 'c',\ [\,])\ \&\ Empty$$

Als nächstes wird Knoten 2 und eine Kante von 2 nach 3 mit Label *down* eingefügt, also muss die Nachfolgerliste des Knotens 2 die 3 enthalten:

$$([\,],\ 2,\ 'b',\ [(``down",\ 3)])\ \&\ ([\,],\ 3,\ 'c',\ [\,])\ \&\ Empty$$

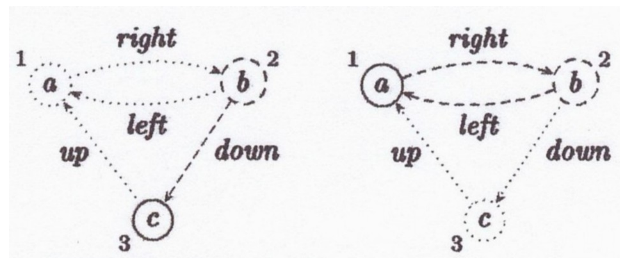
Der gesamte Graph sieht dann wie folgt aus:

$$([(``left",\ 2),(``up",\ 3)],\ 1,\ 'a',\ [(``right",\ 2)])$$
$$\&\ ([\,],\ 2,\ 'b',\ [(``down",\ 3)])\ \&\ ([\,],\ 3,\ 'c',\ [\,])\ \&\ Empty$$

Dies ist jedoch nicht die einzige Möglichkeit den Graphen darzustellen. Man kann ihn zum Beispiel in umgekehrter Knotenreihenfolge aufbauen:

$$([(``down",\ 2)],\ 3,\ 'c',\ [``up",\ 1)])$$
$$\&\ ([(``right",\ 1)],\ 2,\ 'b',\ [(``left",\ 1)])\ \&\ ([\,],\ 1,\ 'a',\ [\,])\ \&\ Empty$$

Die beiden Wege sind hier auch graphisch noch einmal veranschaulicht, links der erste, rechts der zweite:



Es ist sogar so, dass für den induktiven Aufbau eines Graphen die Reihenfolge, in der die Knoten eingefügt werden, völlig beliebig ist.
Dies führt zu zwei wichtigen Feststellungen:

- **Fakt 1 (Vollständigkeit)**
  Zu jedem Multigraphen mit Labeln gibt es einen Graphterm, durch den dieser repräsentiert werden kann

- **Fakt 2 (Repräsentationsmöglichkeiten)**
  Zu jedem Graphen g und jedem Knoten v von g mit Label l existieren p, s und g' so, dass (p, v, l, s) & g' gerade g entspricht.

Nun bleibt noch zu klären, ob auch eine Konsistenzüberprüfung der Graphkonstruktion möglich ist. Da & aber als eine Funktion definiert wurde, können Konsistenzchecks integriert werden, sodass ein Fehler gemeldet wird, wenn bereits vorhandene Knoten eingefügt werden sollen oder ein in einer Vorgänger- oder Nachfolgerliste vorhandener Knoten im Graphen selbst fehlt.

2.2 Pattern-Matching auf Graphen

2.2.1 Pattern-Matching

Wir haben gesehen, dass bei der induktiven Graphkonstruktion die Reihenfolge des Einfügens von Knoten egal ist. Damit ist die theoretische Grundlage für ein effektives Pattern-Matching auf Graphen geschaffen, das in diesem Abschnitt an Hand von einigen Beispielen vorgestellt wird.
Wir beginnen mit einer einfachen Funktion, die lediglich testet, ob ein Graph leer ist oder nicht:

$$
\begin{aligned}
&isEmpty \;::\; Graph\ a\ b \rightarrow Bool \\
&isEmpty\ Empty \;=\; True \\
&isEmpty\ _ \;=\; False
\end{aligned}
$$

Eine etwas interessantere Operation ist *gmap*, eine *map*-Funktion auf Graphen, die als Eingabe eine Funktion f und einen Graphen hat. Die Funktion f wiederum arbeitet auf einem Context:

$$
\begin{aligned}
&gmap \;::\; (Context\ a\ b \rightarrow Context\ c\ d) \rightarrow Graph\ a\ b \rightarrow Graph\ c\ d \\
&gmap\ f\ Empty \;=\; Empty \\
&gmap\ f\ (c\ \&\ g) \;=\; f\ c\ \&\ gmap\ f\ g
\end{aligned}
$$

Mit der vorne eingeführten Graphdefinition über *matchAny* ließe sich *gmap* dann auch so definieren:

$$
\begin{aligned}
gmap\ f\ g\ &|\ isEmpty\ g \;=\; g \\
&|\ otherwise \;=\; f\ c\ \&\ (gmap\ f\ g') \\
&\textbf{where}\ (c,\ g') \;=\; matchAny\ g
\end{aligned}
$$

Man erkennt, dass *gmap* die Struktur der Knoten erhält, aber die der Kanten nicht unbedingt. Dies führt uns zu einer ersten Anwendung von *gmap*, bei der die Pfeile eines Graphen umgedreht werden. Realisiert wird dies durch Vertauschen der Vorgänger- und Nachfolgerlisten in jedem Context:

$$\begin{aligned} &grev \ :: \ Graph\ a\ b \ \rightarrow Graph\ a\ b \\ &grev \ = \ gmap\ swap\ \textbf{where}\ swap\ (p, v, l, s) \ = \ (s, v, l, p) \end{aligned}$$

Als nächsten Schritt wollen wir anhand zweier Eigenschaften obiger Funktionen demonstrieren, wie einfach Beweise durchzuführen sein können.
Die Hintereinanderausführung zweier *gmap*-Operation kann zu einer Operation verschmolzen werden:
Behauptung: *(Fusion von gmap)*

$$gmap\ f\ .\ gmap\ f' = gmap\ (f\ .\ f')$$

Beweis: per Induktion über die Struktur des Graphen g
Induktionsanfang:
Sei $g = Empty$. Dann gilt:

$$gmap\ f\ (gmap\ f'\ Empty) = gmap\ f\ Empty = Empty = gmap\ (f.f')\ Empty$$

Induktionsschluss:
Sei $g = c\ \&\ g'$ und die Behauptung für g' gezeigt (I.V.). Dann gilt:

$$\begin{aligned} gmap\ f\ (gmap\ f'\ g) &= gmap\ f\ (gmap\ f'\ (c\ \&\ g')) \\ &= gmap\ f\ (f'\ c\ \&\ (gmap\ f'\ g')) \\ &= f\ (f'\ c)\ \&\ gmap\ f\ (gmap\ f'\ g') \\ &= (f\ .\ f')\ c\ \&\ gmap\ (f\ .\ f')\ g' \\ &= gmap\ (f\ .\ f')\ (c\ \&\ g') \\ &= gmap\ (f\ .\ f')\ g \end{aligned}$$

Als Zweites soll gezeigt werden, dass die doppelte Umkehrung eines Graphen mittels *grev* wieder der Graph selbst ist.
Behauptung: *(Inversion von grev)*

$$grev\ .\ grev = id$$

Beweis:
Zum Beweis benötigen wir folgende zwei offensichtliche Feststellungen:

- Idempotenz von *swap*: $swap\ .\ swap = id$
- Anwendung von *gmap* auf die Identität: $gmap\ id = id$

Damit gilt unter Verwendung der Fusionsregel von *gmap*:

$$\begin{aligned} grev\ .\ grev &= gmap\ swap\ .\ gmap\ swap \\ &= gmap\ (swap\ .\ swap) \\ &= gmap\ id \\ &= id \end{aligned}$$

Man erkennt schnell, wie elegant und einfach dieser Beweis ist, wenn man vergleichsweise den Beweis derselben Aussage in der imperativen Variante betrachtet. Dort erfolgt dieser durch Iterieren über alle Adjazenzlisten, was einen doch deutlich höheren Aufwand darstellt.

Eine weitere nützliche Funktion auf Graphen ist *ufold*, die von den Fold-Operationen für Listen abgeleitet und wie folgt definiert ist:

$$\begin{aligned} &ufold\ ::\ (Context\ a\ b \rightarrow c \rightarrow c) \rightarrow c \rightarrow Graph\ a\ b \rightarrow c \\ &ufold\ f\ u\ Empty\ =\ u \\ &ufold\ f\ u\ (c\ \&\ g)\ =\ f\ c\ (ufold\ f\ u\ g) \end{aligned}$$

Die Funktionsweise von *ufold* ist ähnlich wie die der *map*-Operation. Auf jedes *Context*-Element einer Graphen wird die Funktion f angewendet. Der Unterschied ist aber, dass f nun zwei Argumente hat, von denen das eine zwar weiterhin das entsprechende *Context*-Element ist, das andere jedoch, beginnend mit einem Startelement, in jedem Iterationsschritt mit übergeben wird und bei der Auswertung eine Art Zwischenwert darstellt.
Zur Verdeutlichung eine Anwendung:
Die Funktion *nodes* soll die Knoten eines Graphen als Liste ausgeben:

$$\begin{aligned} &nodes\ ::\ Graph\ a\ b\ \rightarrow [Node] \\ &nodes\ =\ ufold\ (\backslash(p, v, l, s) \rightarrow (v\ :))\ [\ ] \end{aligned}$$

Wenden wir *nodes* nun auf unseren Beispielgraphen aus Abschnitt 2.1 bei Weglassen der Labels an, so erhalten wir:

$$\begin{aligned} &nodes\ (([2, 3], 1, [2])\ \&\ ([\ ], 2, [3])\ \&\ ([\ ], 3, [\ ])\ \&\ Empty) \\ &= 1 :\ nodes\ (([\ ], 2, [3])\ \&\ ([\ ], 3, [\ ])\ \&\ Empty) \\ &= 1 :\ (2 :\ nodes\ (([\ ], 3, [\ ])\ \&\ Empty) \\ &= 1 :\ (2 :\ (3 :\ nodesEmpty)) \\ &= 1 :\ (2 :\ (3 :\ [\ ])) \\ &= [1, 2, 3] \end{aligned}$$

Mit Hilfe von *nodes* lässt sich eine Funktion *newnodes* definieren, welche i Knoten zurückgibt, die im Argumentgraphen noch nicht vorhanden sind:

$$newNodes :: Int \rightarrow Graph\ a\ b \rightarrow [Node]$$
$$newNodes\ i\ g = [n{+}1..n{+}i]\ \textbf{where}\ n = foldr\ max\ 0\ (nodes\ g)$$

Möchte man einen gerichteten in einen ungerichteten Graphen durch einfaches Weglassen der Pfeilspitzen umwandeln, so kann dies erreicht werden, indem man zur Vorgängerliste alle Nachfolger hinzufügt und anders herum. So erhält man einen symmetrischen gerichteten Graphen, der einem ungerichteten entspricht. Hierbei ist darauf zu achten, dass doppelte Elemente, die bei der Vereinigung entstanden sind, wieder entfernt werden:

$$nub :: [Int] \rightarrow [Int]$$
$$nub\ [\,] = [\,]$$
$$nub\ (x : xs) = x : (nub\ (filter(/= x)\ xs)$$

$$undir :: Eq\ b \Rightarrow Graph\ a\ b \rightarrow Graph\ a\ b$$
$$undir = gmap\ (\backslash(p, v, l, s) \rightarrow \textbf{let}\ ps = nub\ (p{+\!+}s)\ \textbf{in}\ (ps, v, l, ps))$$

2.2.2 Termrepräsentationen und active patterns

Wie bereits erwähnt, gibt es für einen Graphen mehrere mögliche Formen, ihn in einem Term darzustellen. Dies führt dazu, dass bei der Korrektheitsuntersuchung eines Algorithmus dieser robust im Hinblick auf verschiedene Termrepräsentationen sein muss. Die Frage nach der Korrektheit ist somit für jede Algorithmusdefinition einzeln zu beantworten.
Betrachten wir zum Beispiel unsere Definition von *grev*. Hier wird weder die Reihenfolge der *Contexts* verändert, noch wird durch das Vertauschen der Vorgänger- und Nachfolgerlisten die Bedingung verletzt, dass die darin befindlichen Knoten bereits im Graphen enthalten sein müssen. *grev* arbeitet also korrekt.
Problematisch wird es aber bei denjenigen Funktionen, die den *Context* auf mehr oder weniger beliebige Art und Weise verändern. *gmap* ist ein Beispiel dafür, da die oben schon geschilderte Anforderung an Nachfolger- und Vorgängerknoten, bereits im Graphen zu sein, unter Umständen verletzt werden könnte.
Weiterhin erfordern es manche Algorithmen, dass ein Graph in einer ganz speziellen Reihenfolge aufgebaut wurde.
Zur Lösung dieser Problematik führt Erwig *active Patterns* ein. Hierbei handelt es sich um eine Erweiterung von Patterns durch eine Funktionskomponente, die auf den Argumentwert angewendet wird, bevor dieser gegen das Pattern gematcht wird. Dabei kann diese Funktion dazu verwendet werden, das Argument in eine andere Form zu transformieren, also in unserem Fall einen Graphen in eine andere Termdarstellung zu bringen.

Wir haben bereits festgestellt (Fakt 2), dass es für jeden Knoten v eines Graphen eine Termrepräsentation der Art (p, v, l, s) & g mit geeignetem p, l, s und g gibt.
Mit diesem Wissen kann nun ein *active Graph-Pattern* definiert werden:
Das active Pattern $(c \ \&^v \ g)$ wird gegen einen Graphen g' gematcht. Dabei wird in g' nach dem Knoten v gesucht und bei erfolgreicher Suche die Termrepräsentation von g', zumindest konzeptionell, so umgewandelt, dass v als letzter Knoten in g' eingefügt wurde. Damit steht der Context von v an äußerster Stelle.
Ist v im Graphen, so wird vs Context an c gebunden und der Restgraph an g. Falls nicht, wird wie beim normalen Pattern-Matching fortgefahren: Es werden keine Bindungen produziert, die Termrepräsentation nicht verändert. Anzumerken ist noch, dass das v in $\&^v$ ein Ausdruck und kein Pattern ist, d. h., falls es eine Variable ist, muss es bei der Auswertung des active Graph-Patterns bereits an einen Wert gebunden sein. Dies geschieht in der Regel, indem v als Parameter vor dem eigentlichen Graphpattern genutzt wird.
Nachfolgend sind nun einige Beispiele angegeben, die den Gebrauch von active Graph-Patterns demonstrieren sollen:

- Bestimmung der Nachfolger eines Knotens:

$$\begin{aligned} &gsuc :: Node \to Graph\ a\ b \to [Int] \\ &gsuc\ v\ ((_,\ _,\ _,\ s)\ \&^v\ g)\ =\ map\ snd\ s \end{aligned}$$

- Berechnung des Grades eines Knotens:

$$\begin{aligned} ° :: Node \to Graph\ a\ b \to Int \\ °\ v\ ((p,\ _,\ _,\ s)\ \&^v\ g)\ =\ length\ p\ +\ length\ s \end{aligned}$$

- Löschen eines Knotens aus einem Graphen:

$$\begin{aligned} &del :: Node \to Graph\ a\ b \to Graph\ a\ b \\ &del\ v\ (_\ \&^v\ g)\ =\ g \end{aligned}$$

Der Gebrauch von active Patterns bringt jedoch ein Problem mit sich: Sie sind in Haskell nicht verfügbar. Daher sind obige Funktionen in FGL auch anders implementiert. Hier wird die Nutzung von active Patterns durch einen expliziten Aufruf der Funktion *match* ersetzt. *match* sucht in einem Graphen nach dem Context eines gegebenen Knotens und gibt diesen Context, falls gefunden, zusammen mit dem Restgraphen zurück:

$$match :: Node \to Graph\ a\ b \to (Maybe\ (Context\ a\ b),\ Graph\ a\ b)$$

Dann kann eine Funktion f, die active Patterns verwendet, so umgeschrieben werden:

$$
\begin{array}{lll}
f\ p\ (c\ \&^v\ g) = e & & f\ p\ g' = \textbf{case}\ match\ v\ g'\ \textbf{of} \\
f\ p\ g \qquad\quad = e' & \Rightarrow & \qquad (Just\ c,\ g) \quad \rightarrow e \\
 & & \qquad (Nothing,\ g) \rightarrow e'
\end{array}
$$

Die Funktion *gsuc* könnte damit dann auch so implementiert werden:

$$
\begin{array}{lll}
gsuc\ v\ g' & = & \textbf{case}\ match\ v\ g'\ \textbf{of} \\
 & & \qquad (Just\ (_,\ _,\ _,\ s),\ g) \rightarrow map\ snd\ s
\end{array}
$$

Später werden wir auch eine Funktion benötigen, die die Nachfolger aus einem bereits bekannten Context extrahiert. Diese wird jetzt schon einmal definiert:

$$
\begin{array}{l}
suc\ ::\ Context\ a\ b \rightarrow [Node] \\
suc(_,\ _,\ _,\ s)\ =\ map\ snd\ s
\end{array}
$$

Zum Abschluss diese Abschnitts fassen wir die grundlegenden Graphoperationen noch einmal zusammen:

- **Konstruktion**
  - Leerer Graph (*Empty*)
  - Context einfügen (&)
- **Dekomposition**
  - Test auf leeren Graph (*Empty*-match)
  - Beliebigen Context extrahieren (&-match)
  - Bestimmten Context extrahieren ($\&^v$-match)

2.3 Implementierung und Komplexität

2.3.1 Graph-Repräsentationen und Persistenz

Bislang haben wir Graphen nur als abstrakten Datentypen angegeben. Nun beschäftigen wir uns mit der genauen Implementierung. Diese muss die oben zusammengefassten Operationen zur Konstruktion und Dekomposition von Graphen umfassen sowie die Persistenz von Graphen garantieren, das heißt Graph-Updates müssen alte Versionen intakt lassen.
Die erste Idee wäre es, eine einfache Termrepräsentation zu wählen, wodurch Persistenz automatisch erfüllt ist. Jedoch ist dabei eine Implementierung der &- und vor allem der $\&^v$-Operation extrem ineffizient, da es notwendig ist, die Existenz von Vorgängern und Nachfolgern sicher zu stellen, genauso wie die Nichtexistenz von neu einzufügenden Knoten, was bezogen auf die Graphgröße lineare Zeit benötigt.

Auf der Suche nach Alternativen schauen wir uns zunächst einmal an, wie dies in der Regel imperativ gelöst wird. Dort herrschen zwei wesentliche Repräsentationsformen vor: Adjazenzlisten und Inzidenzmatrizen. Im Allgemeinen zieht man Adjazenzlisten vor, denn:

1. sie brauchen weniger Speicherplatz (außer bei hohem Füllgrad)

2. sie bieten eine Zugriffszeit von $O(1)$ auf die Nachfolger eines beliebigen Knotens im Gegensatz zu $\Omega(n)$ bei einer Inzidenzmatrix mit n Spalten

Aus diesen Gründen wollen wir uns auf Adjazenzlisten konzentrieren. Im Folgenden wird nun eine Möglichkeit aufgezeigt, wie das Problem der Persistenz bei Adjazenzlisten gelöst werden kann.

2.3.2 Repräsentation durch binäre Suchbäume

In dem hier vorgestellten werden die Vorgänger- und Nachfolgerlisten eines Knotens in einem balancierten, binären Suchbaum gespeichert.
Ein binärer Suchbaum ist ein binärer Baum, bei dem die Knoten des linken Teilbaums eines Knotens v stets kleiner sind als v selbst, die des rechten entsprechend immer größer. Die Komplexität der Suchoperation ist zwar im schlimmsten Fall bei n Knoten $O(n)$, bei einem balancierten Baum allerdings $O(log\ n)$.
In unserer Variante wird ein Graph durch ein Paar (t, m) repräsentiert, bei dem t ein Suchbaum von Tupeln der Art (Knoten, (Vorgänger, Label, Nachfolger)) und m der größte in t vorkommende Knoten ist.
Betrachten wir die Komplexität der Suchbaumlösung. Die direkte Bekanntheit des größten Knotens bringt den Vorteil mit sich, dass die Schaffung neuer Knoten in $O(1)$ möglich ist.
Beim Einfügen eines neuen Knotens v benötigt das Einfügen von dessen Context (p, v, l, s) $O(log\ n)$ Schritte, was der Komplexität der Suchoperation in Suchbäumen entspricht. Ferner muss aber auch noch v als Nachfolger bzw. Vorgänger für jeden Knoten in p bzw. s eingefügt werden, was eine Zeit von $O(c\ log\ n)$ mit $c = length\ p + length\ s$ erfordert. Insgesamt benötigt also das Einfügen eine Zeit von $O(c\ log\ n)$, was bei vollen Graphen bis zu $O(n\ log\ n)$ werden kann.
Das Löschen eines Knotens v ist ein wenig aufwändiger. Um die Eintragung von v in den Nachfolger- bzw. Vorgängerlisten der Elemente von p bzw. s zu löschen, muss nämlich zunächst einmal die genaue Speicherstelle von v in den entsprechenden Listen gefunden werden, was einen weiteren Faktor c verursacht. Die Komplexität ist damit $O(c^2\ log\ n)$, kann durch Abspeichern der Nachfolger und Vorgänger wiederum als Suchbäume aber noch zu

$O(c\ (log\ c\ log\ n))$ verbessert werden. Bei Graphen mit hohem Füllgrad führt dies zu einer Komplexität von $O(n^2\ log\ n)$ bzw. $O(n\ log^2\ n)$.

3 Funktionale Graphalgorithmen

Nachdem nun die Grundlagen zur induktiven Repräsentation von Graphen vorgestellt wurden, behandelt dieses Kapitel verschiedene Graphalgorithmen und deren Komplexität. Dazu wird angenommen, dass & und $\&^v$ in $O(1)$ liegen, auch wenn dies für einige Implementationen nicht ganz korrekt sein mag.

3.1 Tiefensuche

Die Tiefensuche (DFS) ist ein Suchalgorithmus, der einen Graphen Knoten für Knoten erkundet und dabei zuerst in die Tiefe vorgeht, d. h. er besucht zunächst nur einen Nachfolger des Ausgangsknotens und dann einen Nachfolger dieses Knotens usw.. Wenn er zu einem Knoten kommt, dessen Nachfolger er alle schon besucht hat, dann geht er zu dem davor entdeckten Knoten zurück und betrachtet einen weiteren seiner Nachfolger. Es wird also während des Durchlaufs jeder Knoten genau einmal besucht.
Eine Tiefensuche liefert wichtige Informationen über die innere Struktur des Graphen und wird daher auch zur Implementierung anderer Graphalgorithmen verwendet.
Die Parameter einer Tiefensuche sind zum einen der Graph selbst, aber auch eine Liste von Knoten, die darüber Auskunft gibt, welche Knoten bereits besucht wurden und welche nicht. Dies ist wichtig, falls der Graph nicht zusammenhängend ist. Als Ergebnis kann beispielsweise eine Liste der Knoten in der Reihenfolge, in der sie besucht wurden, ausgegeben werden oder ein Wald von Bäumen, der genau diejenigen Kanten des Ausgangsgraphen enthält, über die die Knoten bei der Tiefensuche erreicht wurden.

Wir beginnen mit einem Algorithmus, der die erste der oben erwähnten Ausgabevarianten umsetzt:

$$\begin{array}{lll} dfs :: [Node] \rightarrow Graph\ a\ b \rightarrow [Node] & & \\ dfs\ [\,]\quad g & = & [\,] \\ dfs\ vs\quad Empty & = & [\,] \\ dfs\ (v : vs)\ (c\ \&^v\ g) & = & v :\ dfs\ (suc\ c{+\!\!+}vs)\ g \\ dfs\ (v : vs)\ g & = & dfs\ vs\ g \end{array}$$

Zur Funktionsweise: Der erste Fall beschreibt die Situation, in der keine Knoten mehr zu besuchen sind, im zweiten Fall ist der Graph leer. Dann stoppt

die Tiefensuche und gibt die leere Liste zurück. Andernfalls versucht *dfs* den Context des Knotens v im Graphen zu finden. Im Erfolgsfall (dritte Gleichung) wird v als nächster Knoten in die Resultatsliste eingefügt und die Suche auf dem Restgraphen beginnend mit v's Nachfolgern fortgesetzt. Im Misserfolgsfall, d. h. v war im Ausgangsgraphen nicht enthalten, fährt *dfs* einfach mit den verbleibenden Knoten von vs fort.
Es ist interessant zu sehen, wie elegant die für die Tiefensuche so typische Eigenschaft, dass jeder Knoten nur einmal besucht wird, sichergestellt wurde. Während bei imperativen Implementierungen die Knoten mit verschiedenen Marken versehen werden und bei den meisten funktionalen mit einer zusätzlichen Mengenstruktur gearbeitet wird, um sich bereits besuchte Knoten zu merken, löst unsere Definition dies durch active Patterns: Wird ein Knoten v besucht und damit in der zweiten Gleichung gematcht, so arbeitet der Algorithmus anschließend auf einem Restgraphen g weiter, der v überhaupt nicht mehr enthält. Es besteht also gar keine Notwendigkeit, sich besuchte Knoten explizit zu merken.
Um noch einmal deutlich zu machen, dass die hier verwendeten, in Haskell aber nicht umsetzbaren, active Patterns mit Hilfe der *match*-Funktion ersetzt werden können, ist hier der *dfs*-Algorithmus noch einmal alternativ implementiert:

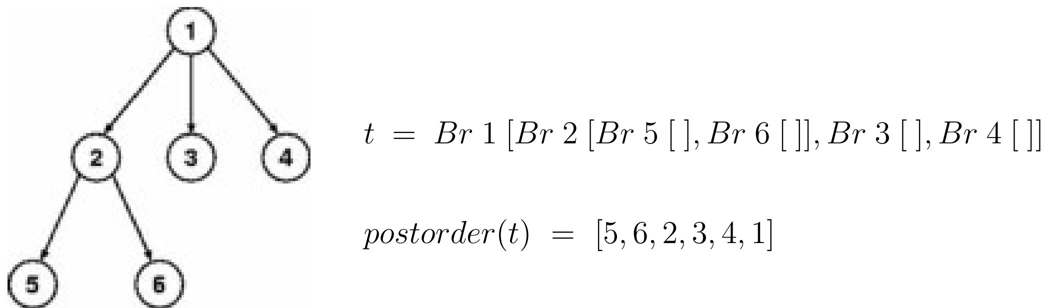
$$
\begin{array}{lcl}
dfs\ [\,]\ \ g' & = & [\,] \\
dfs\ (v:vs)\ g' & = & \textbf{case}\ match\ v\ g'\ of \\
 & & (Just\ c,\ g) \quad \rightarrow v : dfs\ (suc\ c{+}{+}vs)\ g \\
 & & (Nothing,\ g) \rightarrow dfs\ vs\ g
\end{array}
$$

Eine zweite Möglichkeit, das Resultat der Tiefensuche auszugeben, ist die Rückgabe als spannenden Wald, also einem Wald von Bäumen, der alle Knoten des Ausgangsgraphen enthält.
Bei der Implementierung muss berücksichtigt werden, dass Nachfolger und Geschwister eines Knotens v gesondert behandelt werden müssen. In *dfs* war dies nicht nötig, da alle Knoten lediglich in eine Resultatsliste eingereiht wurden. Nun aber muss die Struktur von Bäumen gespeichert werden, d. h. Nachfolger bilden Unterbäume von v, während Geschwister gemeinsam mit v Wurzeln von Unterbäumen von v's Vorgänger werden.
Bevor wir mit dem eigentlichen Algorithmus starten können, benötigen wir eine Datenstruktur für Bäume und eine Funktion, die einen Baum in Postorder ausliest, d. h. eine Knotenliste erstellt, in der die Knoten von Unterbäumen vor der Wurzel stehen:

$$
\textbf{data}\ Tree\ a\ =\ Br\ a\ [Tree\ a]
$$

$$
\begin{aligned}
&postorder \; :: \; Tree\ a \rightarrow [a] \\
&postorder\ (Br\ v\ ts) \; = \; concatMap\ postorder\ ts{+}{+}[v] \\[1em]
&concatMap \; :: \; (a \rightarrow [b]) \rightarrow [a] \rightarrow [b] \\
&concatMap\ f \; = \; concat\ .\ map\ f
\end{aligned}
$$

Ein Baum besteht also aus einer Wurzel des Typs a und einer Liste von Unterbäumen. *postorder* benutzt die im Haskell-Prelude definierte Funktion *concatMap*, die nach Anwendung einer Funktion über *map* die Resultatslisten konkateniert.
Hier noch ein Beispiel eines einfachen Baumes t:



$$t \; = \; Br\ 1\ [Br\ 2\ [Br\ 5\ [\,], Br\ 6\ [\,]], Br\ 3\ [\,], Br\ 4\ [\,]]$$

$$postorder(t) \; = \; [5, 6, 2, 3, 4, 1]$$

Nun können wir einen Tiefensuchalgorithmus definieren, der einen Wald als Rückgabewert liefert:

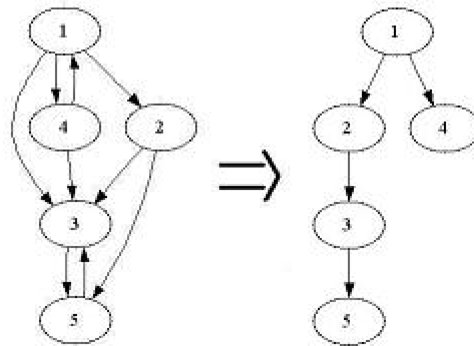
$$
\begin{aligned}
&df \; :: \; [Node] \rightarrow Graph\ a\ b \rightarrow ([Tree\ Node],\ Graph\ a\ b) \\
&df\ [\,] \quad\quad g \quad\quad\quad = \; ([\,],\ g) \\
&df\ vs \quad\; Empty \quad = \; ([\,],\ Empty) \\
&df\ (v : vs)\ (c\ \&^v\ g) \; = \; (Br\ v\ f : f',\ g_2)\ \textbf{where} \\
&\qquad\qquad\qquad\qquad\qquad\qquad (f,\ g_1) \; = \; df\ (suc\ c)\ g \\
&\qquad\qquad\qquad\qquad\qquad\qquad (f',\ g_2) \; = \; df\ vs\ g_1 \\
&df\ (v : vs)\ g \quad\quad = \; df\ vs\ g
\end{aligned}
$$

Der wesentliche Unterschied zu *dfs* besteht in der dritten Gleichung. Wird ein Knoten v im Graphen gefunden, so sind zwei unabhängige, spannende Wälder zu erstellen. Zunächst wird eine Tiefensuche auf den Nachfolgern von v im Restgraphen g durchgeführt, wobei als Ergebnis der Tiefensuchenwald f und ein Teilgraph g_1 von g zurückgegeben werden. g_1 stellt dabei den Teil von g dar, der beim Tiefensuchendurchlauf auf v's Nachfolgern nicht behandelt wurde. Dadurch, dass bei der Erstellung des Tiefensuchenwaldes f' nur auf g_1 gearbeitet wird, wird sichergestellt, dass jeder Knoten nur einmal besucht wird. f' ist nun der Resultatswald der Tiefensuche auf den Knoten, die bisher noch nicht behandelt wurden, weshalb bei der Ausgabe von *df* auch der im

zweiten Parameter übergebene Graph von großer Bedeutung ist, da hier die noch ungenutzten Teile des Ausgangsgraphen gespeichert sind.
Da dieser Graph aber nur intern genutzt wird, definieren wir eine zusätzliche Funktion *dff*, die lediglich den Tiefensuchenwald zurückgibt:

$$\begin{aligned} dff &:: [Node] \rightarrow Graph\ a\ b \rightarrow [Tree\ Node] \\ dff\ vs\ g &= fst\ (df\ vs\ g) \end{aligned}$$

Wir wollen nun zu einem Beispielgraphen g_0 dessen Tiefensuchenbaum mit Hilfe des eben vorgestellten Algorithmus bestimmen. Es folgen Abbildungen von g_0 und dem erwarteten Resultatsbaums zur Wurzel 1:



g_0 lässt sich unter Vernachlässigung von Labels folgendermaßen ausdrücken:

$$\begin{aligned} g_0 = &\ ([4], 1, [2,3,4]) \\ &\ \&\ ([\ ], 2, [3,5]) \\ &\ \&\ ([4,5], 3, [5]) \\ &\ \&\ ([\ ], 4, [\ ]) \\ &\ \&\ ([\ ], 5, [\ ]) \\ &\ \&\ Empty \end{aligned}$$

Die Berechnung des Tiefensuchenbaums zur Wurzel 1 beginnt mit:

$$dff\ [1]\ g_0 = fst\ (df\ [1]\ g_0)$$

Wir werden einige während der Tiefensuche entstehende Teilgraphen von g_0 benötigen, die hier zur besseren Lesbarkeit kurz definiert werden:

$$\begin{aligned} g' &= ([\ ], 2, [3,5])\ \&\ ([4,5], 3, [5])\ \&\ ([\ ], 4, [\ ])\ \&\ ([\ ], 5, [\ ])\ \&\ Empty \\ g'' &= ([4,5], 3, [5])\ \&\ ([\ ], 4, [\ ])\ \&\ ([\ ], 5, [\ ])\ \&\ Empty \\ g''' &= ([\ ], 4, [\ ])\ \&\ ([\ ], 5, [\ ])\ \&\ Empty \\ g'''' &= ([\ ], 4, [\ ])\ \&\ Empty \end{aligned}$$

Es folgt die rekursive Anwendung der Funktion *df*, wobei die Endergebnisse jedes Ableitungszweiges in blau dargestellt sind.

$df\ [1]\ g_0 = (Br\ 1\ f_1 : f_2, g_2) = ([Br\ 1\ [Br\ 2\ [Br\ 3\ [Br\ 5\ [\,]]], Br\ 4\ [\,]]]], Empty)$
$\quad (f_1, g_1) = df\ [2,3,4]\ g' = ([Br\ 2\ [Br\ 3\ [Br\ 5\ [\,]]]], Empty)$
$\quad\quad df [2,3,4]\ g' = (Br\ 2\ f_3 : f_4, g_4) = ((Br\ 2\ [Br\ 3\ [Br\ 5\ [\,]]]) : [\,], Empty)$
$\quad\quad\quad (f_3, g_3) = df\ [3,5]\ g'' = ([Br\ 3\ [Br\ 5\ [\,]]], g'''')$
$\quad\quad\quad\quad df [3,5]\ g'' = (Br\ 3\ f_5 : f_6, g_6) = ((Br\ 3\ [Br\ 5\ [\,]]) : [\,], g'''')$
$\quad\quad\quad\quad\quad (f_5, g_5) = df\ [5]\ g''' = ([Br\ 5\ [\,]], g'''')$
$\quad\quad\quad\quad\quad\quad df [5]\ g''' = (Br\ 5\ f_7 : f_8, g_8) = ((Br\ 5\ [\,]) : [\,], g'''')$
$\quad\quad\quad\quad\quad\quad\quad (f_7, g_7) = df\ [\,]\ g'''' = ([\,], g'''')$
$\quad\quad\quad\quad\quad\quad\quad (f_8, g_8) = df\ [\,]\ g'''' = ([\,], g'''')$
$\quad\quad\quad\quad\quad (f_6, g_6) = df\ [5]\ g'''' = ([\,], g'''')$
$\quad\quad\quad\quad\quad\quad df\ [5]\ g'''' = [\,]\ g'''' = ([\,], g'''')$
$\quad\quad\quad (f_4, g_4) = df\ [3,4]\ g'''' = ([Br\ 4\ [\,]], Empty)$
$\quad\quad\quad\quad df [3,4]\ g'''' = df\ [4]\ g'''' = ([Br\ 4\ [\,]], Empty)$
$\quad\quad\quad\quad\quad df [4]\ g'''' = (Br\ 4\ f_9 : f_{10}, g_{10}) = ((Br\ 4\ [\,]) : [\,], Empty)$
$\quad\quad\quad\quad\quad\quad (f_9\ , g_9\ ) = df\ [\,]\ Empty = ([\,], Empty)$
$\quad\quad\quad\quad\quad\quad (f_{10}, g_{10}) = df\ [\,]\ Empty = ([\,], Empty)$
$\quad (f_2, g_2) = df\ [\,]\ Empty = ([\,], Empty)$

Das so erhaltene Ergebnis $Br\ 1\ [Br\ 2\ [Br\ 3\ [Br\ 5\ [\,]]], Br\ 4\ [\,]]$ entspricht genau dem erwarteten Tiefensuchenbaum.
Mit Hilfe der Tiefensuche können weitere Graphalgorithmen einfach realisiert werden, von denen hier zwei vorgestellt werden sollen.
Ein erster ist die topologische Sortierung eines Graphen, bei der die Knoten in einer Liste derart angeordnet werden, dass alle Nachfolger eines Knotens hinter diesem stehen. Wenden wir unsere *postorder*-Funktion auf den Tiefensuchenwald des Graphen an, so stehen alle Nachfolger eines Knotens vor diesem. Damit muss diese Liste nur noch umgedreht werden und wir erhalten eine topologische Sortierung:

$$\begin{aligned} topsort &:: Graph\ a\ b \rightarrow [Node] \\ topsort &= reverse\ .\ concatMap\ postorder\ .\ dff \end{aligned}$$

Man beachte, dass *postorder* auf Bäumen definiert ist. Da wir aber mit einem ganzen Wald operieren, bedarf es der Anwendung von *postorder* auf jeden einzelnen Baum dieses Waldes, was mit *concatMap* realisiert wird.
Darauf aufbauend lassen sich die starken Zusammenhangskomponenten eines Graphen G berechnen. Diese sind Teilgraphen von G, in denen jeder Knoten von jedem anderen über einen Pfad in dieser Komponente erreichbar ist. Berechnet werden können solche starken Zusammenhangskomponenten, indem

man einen Tiefensuchenwald des umgekehrten Graphen ausgehend von einer topologisch sortierten Knotenliste ermittelt:

$$\begin{aligned} &scc :: Graph\ a\ b \rightarrow [TreeNode] \\ &scc\ g = dff\ (topsort\ g)\ (grev\ g) \end{aligned}$$

3.2 Breitensuche

Ein zweiter grundlegender Graphalgorithmus ist die Breitensuche, die sich zur Tiefensuche in nur einem wesentlichen Punkt unterscheidet. Beim Durchsuchen aller Knoten eines Graphen werden nun zuerst die Geschwister eines Knotens besucht und erst dann dessen Nachfolger, es wird also in die Breite vorgegangen.

$$\begin{array}{lll} bfs :: [Node] \rightarrow Graph\ a\ b \rightarrow [Node] & & \\ bfs\ [\,]\quad g & = & [\,] \\ bfs\ vs\quad Empty & = & [\,] \\ bfs\ (v : vs)\ (c\ \&^{v}\ g) & = & v : bfs\ (vs {+\!\!+} suc\ c)\ g \\ bfs\ (v : vs)\ g & = & bfs\ vs\ g \end{array}$$

Der Unterschied zum *dfs*-Algorithmus besteht nur in der dritten Gleichung. Wurden dort die Nachfolger des Knotens v vor die Liste vs gesetzt, werden sie nun dahinter eingefügt. Es handelt sich nicht mehr um einen Stack, der bei *dfs* genutzt wurde, sondern um eine Schlange. Zwar ist die Implementation einer Schlange als Liste nicht sehr effizient, soll uns aber der Einfachheit halber genügen. Da bessere Implementierungen Operationen auf der Schlange in konstanter Zeit ermöglichen, wollen wir im Folgenden auch mit dieser Komplexität arbeiten. bfs ist damit ein linearer Algorithmus.
Wir können nun zu dem bereits bei der Tiefensuche verwendeten Graphen g_0 eine Liste von Knoten in der Reihenfolge berechnen, wie sie bei der Breitensuche beginnend beim Knoten 1 besucht wurden. Wir definieren wieder benötigte Teilgraphen von g_0:

$$\begin{array}{lll} g_0 & = & ([4], 1, [2, 3, 4])\ \&\ ([\,], 2, [3, 5])\ \&\ ([4, 5], 3, [5]) \\ & & \ \&\ ([\,], 4, [\,])\ \&\ ([\,], 5, [\,])\ \&\ Empty \\ g' & = & ([\,], 2, [3, 5])\ \&\ ([4, 5], 3, [5])\ \&\ ([\,], 4, [\,]) \\ & & \ \&\ ([\,], 5, [\,])\ \&\ Empty \\ g'' & = & ([4, 5], 3, [5])\ \&\ ([\,], 4, [\,])\ \&\ ([\,], 5, [\,])\ \&\ Empty \\ g''' & = & ([\,], 4, [\,])\ \&\ ([\,], 5, [\,])\ \&\ Empty \\ g'''' & = & ([\,], 5, [\,])\ \&\ Empty \end{array}$$

Jetzt kann bfs auf g_0 mit Startknoten 1 angewendet werden:

$$bfs\ [1]\ g_0 = 1\ :\ bfs\ [2,3,4]\ g' = [1,2,3,4,5]$$
$$bfs\ [2,3,4]\ g' = 2\ :\ bfs\ [3,4,3,5]\ g'' = [2,3,4,5]$$
$$bfs\ [3,4,3,5]\ g'' = 3\ :\ bfs\ [4,3,5,5]\ g''' = [3,4,5]$$
$$bfs\ [4,3,5,5]\ g''' = 4\ :\ bfs\ [3,5,5]\ g'''' = [4,5]$$
$$bfs\ [3,5,5]\ g'''' = bfs\ [5,5]\ g'''' = [5]$$
$$bfs\ [5,5]\ g'''' = 5\ :\ bfs\ [5]\ Empty = [5]$$
$$bfs\ [5]\ Empty = [\ ]$$

Wie auch bei der Tiefensuche ist es weitaus schwieriger eine Breitensuchwald zu erzeugen. Hinzu kommt noch, dass Ausdrücke, durch die die Bäume dargestellt werden, wie bei *df*, von unten nach oben aufgebaut werden müssen, d. h. Unterbäume werden an eine Wurzel angefügt. Jedoch liefert die Rekursion in bfs die Knoten so, dass sie stattdessen für eine Konstruktion von oben nach unten prädestiniert sind.
Dies wiederum kann bei der wichtigsten Anwendung von Breitensuchbäumen, der Bestimmung kürzester Wege von einer Wurzel zu einem anderen Knoten, ausgenutzt werden. Dort werden gerichtete Bäume verwendet, bei denen die Pfeile von Nachfolgern zu Vorgängern führen, um so einen Weg von diesem Knoten zur Wurzel leicht ermitteln zu können.
Damit kann die Suche eines kürzesten Weges vom Knoten s zum Knoten t mit Hilfe der Breitensuche wie folgt realisiert:

1. Berechne den Breitensuchenbaum mit Wurzel s.

2. Finde t darin.

3. Folge den Kanten von t bis zur Wurzel

Die Repräsentierung eines Baumes wie oben erwähnt erfolgt, indem man zu jedem Knoten dessen Vorgänger aufzeichnet. Da solch eine Aufzeichnung dynamisch erstellt wird, kann zur Implementierung kein Array genommen werden. Stattdessen benutzten wir einen binären Suchbaum. Dies führt zwar zu einem zusätzlichen logarithmischen Faktor bei jeder Operation zur Berechnung des Breitensuchenbaumes und der anschließenden Konstruktion des kürzesten Weges, doch kann zumindest letzterer berichtigt werden.
Zu den einzelnen Knoten werden nicht mehr nur ihre Vorgänger, sondern der gesamte Pfad bis zur Wurzel aufgezeichnet, den wir von nun an *Root-Path* oder *R-Path* nennen wollen. Auf die Komplexität hat dies keinen großen Einfluss. Um u als Vorgänger von v anstatt lediglich mit einem Verweis auf v in den Baum einzufügen, muss zunächst der bei u bereits gespeicherte R-Path ermittelt wegen. Diesen nennen wir p. Dann wird $u : p$ mit dem Verweis auf v in den Baum eingefügt. Auf diese Weise kann der kürzeste Weg von s nach t ermittelt werden, indem erst t in unserem Baum lokalisiert wird und dann

die Liste der dort gespeicherten Knoten umgedreht wird.
Die Komplexität zur Berechnung des Breitensuchbaumes ist ebenso wie die zur Ermittlung kürzester Wege nach wie vor $O(n\ log\ n\ +\ m)$ bei n Knoten und m Kanten.
Eine weitere Verbesserung kann erzielt werden, wenn der Breitensuchbaum durch eine Liste (an Stelle eines Baumes) von R-Paths von jedem Knoten zur Wurzel repräsentiert wird. So einen Baum nennen wir *Root-Path-Tree*.
Nach diesen Vorüberlegungen wollen wir nun eine Funktion bft definieren, die zu einem Knoten v einen Breitensuchbaum mit Wurzel v als eine Liste von R-Paths berechnet.
Hierzu werden zwei neue Typen eingeführt: Ein Root-Path *Path* entspricht einer Liste von Knoten und ein Root-Path-Tree *RTree* einer Liste von Root-Paths.

```
type Path  = [Node]
type RTree = [Path]

bft :: Node → Graph a b → RTree
bft v = bf [[v]]

bf :: [Path] → Graph a b → RTree
bf [ ]               g          = [ ]
bf vs                Empty      = [ ]
bf (p@(v : _) : ps)  (c &v g)   = p : bf (ps++map (: p) (suc c)) g
bf (p : ps)          g          = bf ps g
```

Die Funktion bft startet die Breitensuche in einem Graphen g zur Wurzel v. Dabei wird eine bfs sehr ähnliche Funktion bf aufgerufen, die eine Schlange von R-Paths als erstes Argument hat. Bei der Initialisierung ist dies nur der R-Path von v, also $[v]$.
In bf erfolgt eine Unterscheidung nach 4 Fällen. Ist die Liste von R-Paths oder der Graph selbst leer, so ist nichts zu tun. Ansonsten betrachten wir den ersten Knoten in der Schlange. Dieses v wird durch seinen R-Path repräsentiert. Ist v im betrachteten Graphen enthalten, so wird p an seinen R-Path gebunden und ins Resultat eingefügt. Die Breitensuche wird nun rekursiv auf den Restgraphen g angewendet. Hierfür wird p aus der Schlange entfernt, anschließend werden v's Nachfolger in diese eingefügt, indem ihre noch zu erstellenden R-Paths an ps angehängt werden. Diese entstehen wiederum durch Erweitern des R-Paths von v um den jeweiligen Nachfolger von v. Der vierte Fall befasst sich mit der Situation, dass der zu dem am Schlangenanfang stehenden R-Path p gehörende Knoten im Restgraphen nicht mehr enthalten ist. Er wurde also bereits bearbeitet. Hier kann der Algorithmus

direkt mit der Restschlange ps fortfahren.
Das Resultat ist nun eine Liste von R-Paths, aus denen der Breitensuchbaum leicht zu erstellen ist, indem man gleiche Teile dieser Pfade miteinander identifiziert
Nun können wir einen Algorithmus zur Bestimmung eines kürzesten Weges von s nach t definieren. Dieser ermittelt zuerst den Breitensuchbaum zur Wurzel s, welcher durch eine Liste von Root-Paths repräsentiert wird. Der erste von diesen, der t als erstes Element hat, wird ausgewählt, umgedreht und wir haben mit ihm den kürzesten Weg gefunden. Hier ist der Algorithmus:

$$
\begin{aligned}
&first \ :: \ (a \rightarrow Bool) \rightarrow [a] \rightarrow a \\
&first\ p \ = \ head \ . \ filter\ p \\
\\
&esp \ :: \ Node \rightarrow Node \rightarrow Graph\ a\ b \rightarrow Path \\
&esp\ s\ t \ = \ reverse \ . \ first\ (\backslash(v : _) \rightarrow v \ == \ t) \ . \ bft\ s
\end{aligned}
$$

Unter der Annahme, dass wieder die üblichen Optimierungen wie zuvor durchgeführt wurden und Graphoperationen in $O(1)$ liegen, hat *esp* lineare Laufzeit.

3.3 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus ist ein weiterer Algorithmus zur Bestimmung kürzester Wege. Im Unterschied zu *esp* operiert Dijkstra auf gewichteten Graphen mit positiven Kantenlabeln. Es gibt somit auch eine andere Definition der Länge eines Pfades, die nun die Summe der Kantenlabels ist.
Das Ziel ist es, in einem Graphen G die kürzesten Wege eines Knotens v zu allen von ihm aus erreichbaren Knoten in einem Baum mit v als Wurzel zu speichern. Dafür beginnt man mit einem Baum t, der zunächst nur die Wurzel v enthält. Nun werden die Knoten von G betrachtet, die nicht in t liegen, aber Nachfolger eines Knotens von t sind. Derjenige mit der geringsten Entfernung zur Wurzel v wird mit der entsprechenden Kante in t übernommen. Dieser Schritt wird wiederholt, bis alle von v aus erreichbaren Knoten in t enthalten sind. t ist damit der kürzeste-Wege-Baum zu v.
Dijkstra's Algorithmus kann ähnlich wie bfs realisiert werden, nur dass Root-Paths in einer Halde statt in einer Schlange gespeichert werden.
Eine Halde ist ein höhenbalancierter Baum, bei dem die Labels eines Knotens größer oder gleich der seiner Nachfolger ist.
Wir benutzen die Haldenimplementierung von Okasaki (1998), die drei wesentliche Operationen enthält, die hier nur informell beschrieben werden:

- *unitHeap*: erstellt eine 1-elementige Halde
- *mergeAll*: kombiniert mehrere Halden zu einer einzigen
- *splitMin*: wird auf eine nicht-leere Halde angewendet und gibt ein Paar zurück, welches das Minimum der Halde und den Rest ohne dieses Minimum enthält

Wir werden später ein active Pattern $x \prec h$ benutzen, dass auf *splitMin* basiert. Es matcht jede nicht-leere Halde h' und bindet das Minimum von h' an x, h' ohne x an h.
Wie schon angedeutet werden wir wie bei *bfs* Root-Paths verwenden. Wir müssen unsere Definition jedoch erweitern, da wir es jetzt mit gewichteten Graphen zu tun haben. Ein Root-Path mit Labeln ist eine Liste von Paaren (Knoten, Label), die einen Pfad von einem Knoten zur Wurzel repräsentiert. Das Label stellt dabei die Länge des Pfades von diesem Knoten zur Wurzel dar. Somit zeigt das Label des ersten Knotens von jedem r-Path die Länge des gesamten Pfades an. Das des letzten, also der Wurzel selbst, ist dann natürlich immer 0.

$$\textbf{type}\ LNode\ a\ =\ (Node,\ a)$$
$$\textbf{type}\ LPath\ a\ =\ [LNode\ a]$$
$$\textbf{type}\ LRTree\ a\ =\ [LPath\ a]$$

So wie zuvor Breitensuchbäume durch Listen von R-Paths repräsentiert wurden, werden jetzt kürzeste-Wege-Bäume mit Listen von gelabelten R-Paths dargestellt. Um die gelabelten Pfade später in einer Halde speichern zu können, werden diese als Instanzen von *Ord* und *Eq* definiert:

$$\textbf{instance}\ Eq\ a \Rightarrow Eq\ (LPath\ a)\ \textbf{where}$$
$$((_, x) : _) == ((_, y) : _)\ =\ x == y$$

$$\textbf{instance}\ Ord\ a \Rightarrow Ord\ (LPath\ a)\ \textbf{where}$$
$$((_, x) : _) < ((_, y) : _)\ =\ x < y$$

Die Funktion *getpath* extrahiert einen Pfad zu einem bestimmten Knoten aus einem gelabelten Root-Tree. Labels werden dabei weggelassen:

$$getPath\ ::\ Node \rightarrow LRTree\ a \rightarrow Path$$
$$getPath\ v\ =\ reverse\ .\ map\ fst\ .\ first(\backslash((w, _) : _) \rightarrow w\ ==\ v)$$

getPath liefert uns also den kürzesten Pfad von der Wurzel zu v, natürlich erst wenn der entsprechende Root-Tree zuvor erstellt wurde. Letzteres soll nun angegangen werden.

Die in der Halde gespeicherten Root-Paths repräsentieren den bislang entwickelten kürzesten Wege-Baum, wobei die ersten Knoten den Rand der Suche darstellen und die Labels die Entfernungskosten für diese Knoten. Alle anderen Knoten der Root-Paths sind Knoten, die bereits sicher zum kürzesten Wege-Baum gehören. Der Dijkstra-Algorithmus sucht sich nun den bislang kürzesten Pfad heraus, also den R-Path p, dessen erster Knoten das kleinste Label hat. Dieser Knoten v wird mit der Kante, durch die er erreicht wurde, dauerhaft in den Resultatsbaum eingefügt. Als nächstes werden neue R-Paths erstellt, indem je einer von v's Nachfolgern vor p eingereiht wird und als Label das von v plus die Kosten, die das Label der Kante zu v trägt, erhält. Dieser Vorgang wird von der Funktion *expand* erledigt.
Bei der Komplexitätsbetrachtung stellen wir zunächst fest, dass die Halde bis zu $O(m)$ Einträge enthalten kann bei m Kanten, da für einen Knoten verschiedene Pfade gespeichert werden, von denen am Ende jedoch nur der kostengünstigste gebraucht wird. Soll ein Minimum einer Halde gelöscht werden, so kann dies demnach bis zu $O(m)$ Mal passieren und hat damit eine Laufzeit von $O(m\ log\ m)$ $(= O(m\ log\ n))$. Insgesamt läuft unser Dijkstra-Algorithmus dann in $O(m\ +\ m\ log\ n)$, was asymptotisch schlechter ist als der imperative, der es in $O(m\ +\ n\ log\ n)$ schafft.
Doch nun zur Definition von unserem Algorithmus:

$$expand :: Real\ b \Rightarrow b \rightarrow LPath\ b \rightarrow Context\ a\ b \rightarrow [Heap\ (LPath\ b)]$$
$$expand\ d\ p\ (_,\ _,\ _,\ s)\ =\ map(\backslash(l,\ u) \rightarrow unitHeap\ ((u,\ l+d) : p))\ s$$

$$dijkstra :: Real\ b \Rightarrow Heap\ (LPath\ b) \rightarrow Graph\ a\ b \rightarrow LRTree\ b$$
$$dijkstra\ h\ g\ |\ isEmptyHeap\ h\ ||\ isEmpty\ g\ =\ [\ ]$$
$$dijkstra\ (p@((v,\ d) : _) \prec h)\ (c\ \&^v\ g)$$
$$=\ p : dijkstra\ (mergeAll\ (h : expand\ d\ p\ c))\ g$$
$$dijkstra\ (_ \prec h) g\ =\ dijkstra\ h\ g$$

Zur Funktionsweise: *dijkstra* liefert bei leerer Halde oder leerem Graphen natürlich auch einen leeren Baum zurück (1.Gleichung).
Die zweite Gleichung behandelt den Fall, dass das minimale Haldenelement, also der bislang kürzeste Pfad p, gefunden und sein erster Knoten v mit Label d lokalisiert wurde. Der Pfad p wird in die Lösung übernommen und *dijkstra* auf den Restgraphen g und die Halde angewendet, die entsteht, wenn h zusammen mit denjenigen Halden verschmolzen wird, die durch Expandieren über v's Nachfolger entstehen. Dies wiederum erfolgt über *expand*. *expand* bekommt dabei als Argumente die bisherige Länge d des Pfades zu v, den Pfad p, der zu v führt, und v's Context c, von dem nur die Nachfolgerliste s interessant ist, übergeben. Mit diesen bildet *expand* zu jedem dieser Nachfolger u mittels *unitHeap* eine Halde, die den Pfad von der Wurzel bis zu v enthält. Dazu wird u mit dem Kostenlabel, das der Summe der Kantengewichtung l und der bisherigen Pfadlänge d entspricht, vor den bisherigen

Pfad p angefügt.
Die dritte Gleichung behandelt den Fall, dass der letzte Knoten des kürzesten in der Halde gespeicherten Pfades im noch zu bearbeitenden Graphen nicht mehr enthalten ist. Es existiert also bereits ein Pfad zu ihm, der noch kürzer ist als der momentan ausgewählte. Dann wird der Algorithmus auf der Halde ohne diesen Pfad weitergeführt.
Zu beachten ist noch, dass durch die Einschränkung von b bzgl. des Typs *Real* alle numerischen Typen, mit Ausnahme des komplexen, erlaubt sind. Und da *Real* eine Unterklasse von *Ord* ist, sind diese auch miteinader vergleichbar.
Zum Schluss definieren wir noch eine Funktion *spt*, durch die die Initialisierung der Starthalde vom eigentlichen Algorithmus entkapselt wird. *spt* liefert als Rückgabewert den gesuchten kürzesten-Wege-Baum. Die Funktion *sp* berechnet schließlich den kürzesten Pfad von s nach t.

$$spt \ :: \ Real\ b \Rightarrow Node \rightarrow Graph\ a\ b \rightarrow LRTree\ b$$
$$spt\ v \ = \ dijkstra\ (unitHeap\ [(v, 0)])$$

$$sp \ :: \ Real\ b \Rightarrow Node \rightarrow Node \rightarrow Graph\ a\ b \rightarrow Path$$
$$sp\ s\ t \ = \ getPath\ t\ .\ spt\ s$$

3.4 Maximal unabhängige Knotenmengen

Die bisher vorgestellten Algorithmen haben einen Graphen lediglich in einem einzigen Thread benutzt. Da unser Datentyp für Graphen aber persistent ist, wollen wir nun einen Algorithmus zur Bestimmung maximal unabhängiger Knotenmengen definieren, der diese Eigenschaft ausnutzt, d. h. es werden verschiedene Versionen eines Graphen zur selben Zeit bearbeitet.
Eine unabhängige Knotenmenge ist eine Teilmenge der Knoten eines Graphen, in der keine zwei Knoten durch eine Kante verbunden sind. Eine maximal unabhängige Knotenmenge ist dann eine solche von maximaler Kardinalität.
Das Problem, diese Menge zu finden, ist NP-schwer. Es ist also nicht sehr wahrscheinlich, dass effiziente Algorithmen zur Lösung existieren. Dennoch gibt es bessere Wege, diese zu finden, als einfach alle möglichen Teilmengen durchzuprobieren.
Im Folgenden wird ein Algorithmus vorgestellt, bei dem rekursiv immer wieder folgende zwei Alternativen miteinander verglichen werden:

1. die maximal unabhängige Knotenmenge eines Graphen g, bei dem der Knoten v mit dem höchsten Grad entfernt wurde

2. die maximal unabhängige Knotenmenge von g, von dem v samt seinen Nachbarn entfernt wurde, vereinigt mit v

Dann ist die größere der beiden Mengen die maximal unabhängige Knotenmenge von g.

$$
\begin{array}{ll}
\mathit{indep} :: \mathit{Graph}\ a\ b \rightarrow [\mathit{Node}] & \\
\mathit{indep}\ \mathit{Empty} = [\,] & \\
\mathit{indep}\ g & = \textbf{if}\ \mathit{length}\ i_1 > \mathit{length}\ i_2\ \textbf{then}\ i_1\ \textbf{else}\ i_2 \\
& \quad \textbf{where}\ \mathit{vs} = \mathit{nodes}\ g \\
& \qquad\quad m = \mathit{maximum}\ (\mathit{map}\ (\mathit{flip}\ \mathit{deg}\ g)\ \mathit{vs}) \\
& \qquad\quad v = \mathit{first}\ (\backslash v \rightarrow \mathit{deg}\ v\ g == m)\ \mathit{vs} \\
& \qquad\quad c\ \&^v\ g' = g \\
& \qquad\quad i_1 = \mathit{indep}\ g' \\
& \qquad\quad i_2 = v : \mathit{indep}\ (\mathit{foldr}\ \mathit{del}\ g'\ (\mathit{pre}\ c \mathbin{+\!\!+} \mathit{suc}\ c))
\end{array}
$$

Der Algorithmus läuft wie folgt ab: Aus g's Knotenliste vs wird der erste Knoten v ausgewählt, der den maximalen Grad m hat. Es wird nun die maximal unabhängige Knotenmenge i_1 des Graphen g ohne v berechnet, und analog i_2 für g ohne v und ohne v's Nachfolger und Vorgänger. Letztere werden mit der zu *suc* analogen Funktion *pre* bestimmt. Zu i_2 wird v dann wieder hinzugefügt. Schließlich werden die beiden durch Listen repräsentierten Mengen hinsichtlich ihrer Länge miteinander verglichen. Die größere ist das gesuchte Ergebnis.

4 Zusammenfassung

In dieser Seminararbeit wurde vorgestellt, wie Graphen und Graphalgorithmen in der funktionalen Programmiersprache Haskell definiert werden können.
Auf Basis einer induktiven Graphdefinition wurden Implementierungen verschiedenster Graphalgorithmen erstellt, wie etwa Breiten- und Tiefensuche oder Dijkstra's Algorithmus zur Bestimmung kürzester Wege. Unter Verwendung von rekursiven Funktionen konnten diese auf elegante und klare Weise beschrieben werden, wobei bemerkenswert ist, dass unsere funktionalen Implementierungen hinsichtlich ihrer Effizienz den imperativen in nichts nach stehen.