Modelle, Ideen, Programme
- Gedanken zur Datenstruktur *Graph* -

Gernot Lorenz

Version 1.1, Februar 2009

Zusammenfassung

Mittlerweile hat sich „Modellieren" als didaktischer Schlüsselbegriff der Informatik etabliert, ähnlich wie bei den anderen mathematisch-naturwissenschaftlichen Disziplinen. Insbesondere beim Programmieren fördert das Entwickeln geeigneter Modelle das Abstraktionsvermögen weitaus mehr als das anschließende Codieren.
Die im mathematisch-naturwissenschaftlichen Bereich bekannte Struktur des *Graphen* ist selbst ein Modell für viele reale Situationen und soll im folgenden informatisch modelliert werden: für die Datenmodellierung liegt der Typ *Liste* nahe, für die Verarbeitung soll das funktionale Paradigma eingeübt werden: Es bietet sich die Sprache *Scheme* mit der Lernsprachenumgebung *DrScheme* an.

1 Gemeinsame Strukturen & Abstraktion

Das Erkennen einer gemeinsame Struktur in unterschiedlichen Situationen setzt u.a. die Fähigkeit zur *Abstraktion* voraus. Damit wollen wir beginnen.

1.1 Reale Beispiele

Wir untersuchen und vergleichen drei Situationen (*Abb. 1, Abb. 2 und Abb. 3*):

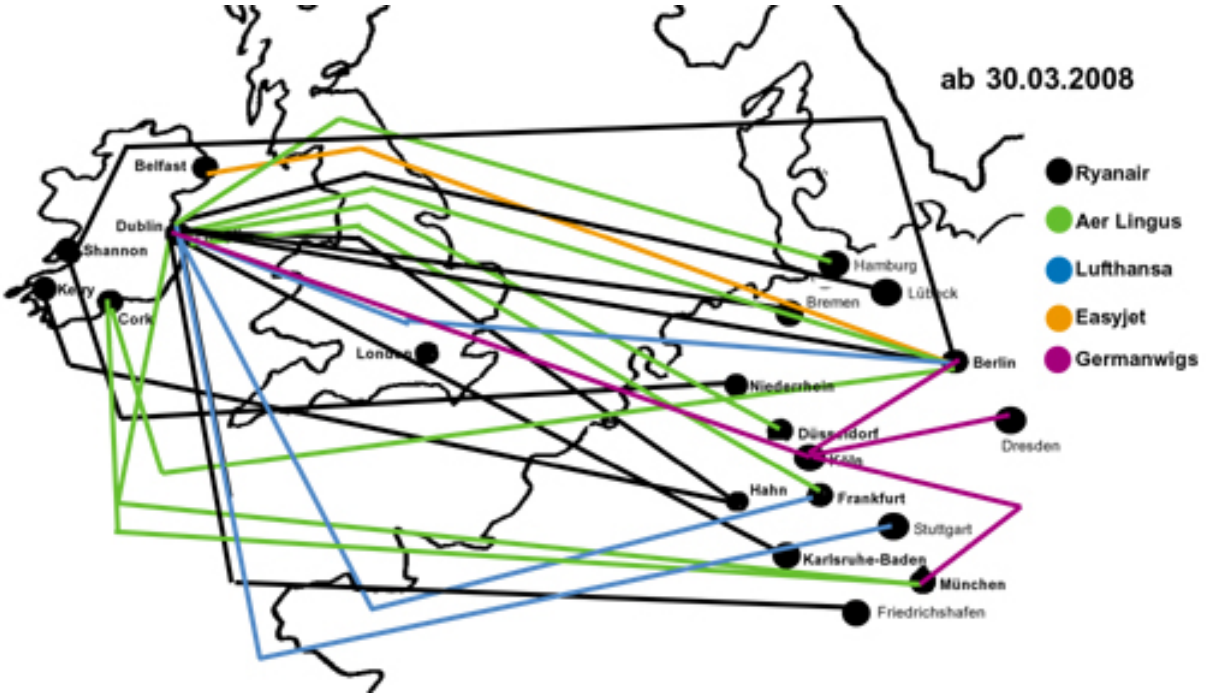

Abb. 1: Flugverbindungen Deutschland - Irland

Abb. 2: Irrgarten

Abb. 3: S-Bahn-Plan (Ausschnitt) von Stuttgart

Wir können jetzt zu den drei Situationen jeweils unterschiedliche Fragen stellen, etwa zu

Situation 1:	Wozu braucht man sowas ? Warum schaut man diesen Plan an ? Wie komme ich von Belfast nach Hannover bzw. Dresden ?
Situation 2:	.... ?
Situation 3:	Wozu braucht man sowas ? Warum schaut man diesen Plan an ? Wie komme ich von Marbach nach Vaihingen ?

Offensichtlich haben *Situation 1* und *Situation 3* große Gemeinsamkeiten, während *Situation 2* aus dem Rahmen fällt.
Vermutlich kommt man mit der allgemeinen Frage

Worum geht es in allen drei Fällen ?

weiter: auch in *Situation 2* will man nach dem Eintritt in den Irrgarten wieder herauskommen, d.h. ähnlich wie in den beiden anderen Situationen, und wir erhalten als Antwort die allgemeine Frage *Wie komme ich vom Eingang zum Ausgang ?*

1.2 Gemeinsame Fragestellung

Daraus folgt die

Abstraktion 1. *Wie komme ich von A nach B ?*

Oder genauer:

- ▷ Gibt es überhaupt einen Weg ?
- ▷ Wenn ja, gibt es mehrere ?
- ▷ Welcher ist der kürzeste ?
- ▷ ...

☞ Mit **genau diesen Fragen** werden wir uns im Folgenden auseinandersetzen, um sie mit Hilfe eines Computerprogramms beantworten zu können

2 Graph als Modell

2.1 Definition

Dass die oben gestellten Fragen berechtigt sind, liegt offenbar am Weg bzw. an den Wegen von A nach B, genauer an der *Struktur* der Wege: Bei allen Situation bestehen sie aus

- *Abschnitten* - von jetzt ab **Kanten** genannt -, an deren
- *Schnittstellen* - von jetzt ab **Knoten** genannt - auch Verzweigungen möglich sind

Da diese beiden Merkmale die in den Beispielen erkannte gemeinsame Struktur ausmachen, hat man eine eigene Begriffsbildung geschaffen:

Defnition 1. *Eine Struktur aus* ***Kanten*** *und* ***Knoten*** *heißt* ***Graph***

An den o.a. Beispielen erkennt man den Nutzen dieser Struktur:

Abstraktion 2. *Ein Graph ist für viele reale Situationen ein passendes* ***Modell***

Machen wir uns klar:

- ▷ die Wahl des Modells hängt von der Fragestellung ab
- ▷ das Modell vernachlässigt alle für die Fragestellung bzw. Problemlösung bedeutungslosen Aspekte

Dies können wir am folgenden Beispiel sehen:

⇒

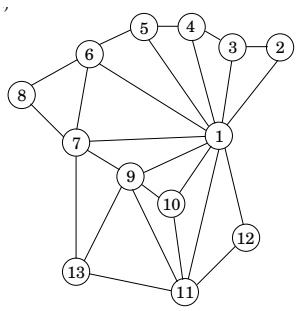

Abb. 4: Südamerika: Graph als Modell für mögliche länderverbindende Flugrouten

Offensichtlich hilft dieses Modell bei Fragen wie

- ▷ Von welchem Land nach welchem anderen Land gibt es eine direkte Flugverbindung ?
- ▷ Wie komme ich von Uruguay nach Venezuela ?
- ▷ Wo muss ich umsteigen ?
- ▷ ...

Dazu braucht man nur die Ländernamen (hier durch Nummern vereinfacht) als Knoten, und die Kanten geben die Verbindungen an, sofern es eine gibt.
Für einen Biologen oder Meterologen dagegen wäre dieses Modell von Südamerika völlig uninteressant.

2.2 Typen

Man kann Graphen (orthogonal) einteilen in verschiedene Typen:

- *gerichtet*, wenn die Kanten zwischen zwei Knoten nur in eine Richtung durchlaufen werden können
- *gewichtet*, wenn die Länge der Kanten eine Rolle spielt
- *zusammenhängend* (selbsterklärend)

- *zyklisch*, wenn es für einen Knoten (in einem gerichteten Baum) einen Weg über andere Knoten zu sich zurück gibt
- ...

✶ Suche und diskutiere reale Beispiele für die verschiedenen Typen

☞ Im Folgenden werden nur *gerichtete, nicht gewichtete, zusammenhängende* Graphen untersucht

2.3 Darstellung/Repäsentation

2.3.1 Diagramm

Die gängige *symbolische* Darstellung ist das *Diagramm:*

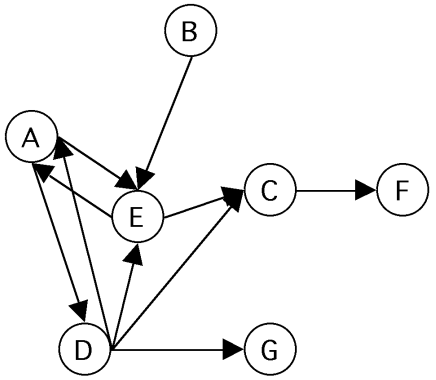

Abb. 5: Muster-Graph G_0: *gerichtet, nicht gewichtet, zusammenhängend*

Dabei werden

- die *Knoten* durch *Kreise mit Namen* repräsentiert
- die *Kanten* durch *Pfeile* zwischen den Kanten repräsentiert

✶ Weshalb braucht man den Kanten im Gegensatz zu den Knoten keine Namen zu geben ?

2.3.2 Formal

Da nach Definition ein Graph G aus einer Mengen von Knoten, im folgenden mit V bezeichnet, und aus einer Mengen von Kanten, im folgenden mit E bzeichnet, besteht, können bei unserem Mustergraph G_0 (*Abb. 5*) sagen:

$$\begin{aligned} G_0 &= (V, E) \text{ mit} \\ V &= \{A, B, C, D, E, F, G\} \text{ und} \\ E &= \{(A,D), (A,E), (B,E), (C,F), (D,A), (D,C), (D,E), (D,G), (E,A), (E,C)\} \end{aligned}$$

✶ Weshalb braucht man bei einem zusammenhängendem Graphen nur E anzugeben ?

3 Informatische Modellierung - funktional

3.1 Datenmodellierung

In der formalen Definition ist ein Graph ein Paar, bestehend aus der Knotenmenge V und der Kantenmenge E, also

```
(define-struct graph (knoten kanten))
```

Bei Mengen liegt eine Modellierung als *Listen* nahe, und es ergibt sich für den den Mustergraphen G_0

```
(define g0
  (make-graph
   (list ... <Knoten> ...)
   (list ... <Kanten> ...))
```

Während die Knoten als Symbole `'A 'B ...` repräsentiert werden können, handelt sich bei den Kanten um geordnete *Paare*. Demnach bietet sich für eine Kante der Datentyp `record` an, d.h. wir können

```
(define-struct kante (ak ek))
```

definieren.
Somit ergibt sich für unseren Mustergraphen G_0:

```
(define g0
  (make-graph
   (list 'A 'B 'C 'D 'E 'F 'G)
   (list
    (make-kante 'A 'D) (make-kante 'A 'E) (make-kante 'B 'E)
    (make-kante 'C 'F) (make-kante 'D 'A) (make-kante 'D 'C)
    (make-kante 'D 'E) (make-kante 'D 'G) (make-kante 'E 'A)
    (make-kante 'E 'C))))
```

Alternativ kann man z.B. auch Einzellisten mit den Knoten und ihren Nachfolgern bilden.

3.2 Spielereien zum Warmlaufen – Übungen

1. Vergleiche

   ```
   (define g0-alternativ
     (make-graph
      (list 'A 'B 'C 'D 'E 'F 'G)
      (list (list 'A 'D) (list 'A 'E) (list 'B 'E) (list 'C 'F) (list 'D 'A)
            (list 'D 'C) (list 'D 'E) (list 'D 'G) (list 'E 'A) (list 'E 'C))))
   ```

   mit `g0`.

   (a) Entwickle eine Funktion

   ```
   ;wandle-graf: graph (Kanten als Verbunde) --> graf (Kanten als Listen)
   ```

   die `g0` in `g0-alternativ` umwandelt.

   (b) Entwickle die Umkehrfunktion zu `mache-graf`

2. Finde weitere Datenmodelle

3. Überlege, wovon die Wahl des Datenmodells abhängt

4. Bei einem zusammenhängenden Graph enthält E alle Informationen. Um zusätzlich eine Liste aller Knoten V zu erhalten, ist eine Funktion

   ```
   ;hole-knoten: graph -->  liste
   ```

   zu entwickeln.

3.3 Rand und Erreichbarkeit

Bevor wir uns dem zentralen Problem

Wie komme ich von A nach B ?

widmen, gehen wir der grundsätzlichen Frage

Kann man vom Knoten A überhaupt Knoten B erreichen ?

nach.
Also suchen wir eine Funktion

```
;erreichbar?: anfangsknoten zielknoten graph --> BOOLEsch
```

und erhalten als Gerüst

```
(define (erreichbar? anfangsknoten zielknoten graph)
        ...)
```

Für `g0` erwarten wir z.B.

```
(erreichbar? 'A 'B g0) --> false
(erreichbar? 'A 'E g0) --> true
(erreichbar? 'A 'F g0) --> true
```

Dafür soll im Folgenden anhand des Mustergraphen G_0 eine Lösung erarbeitet werden.

Neben dem trivialen Fall, wenn der Zielknoten der Anfangsknoten ist `((equal? k z) true)` haben wir den einfachen Fall, dass der Zielknoten B ein direkter Nachfolger des Anfangsknotens ist, also z.B. ist E direkter Nachfolger von A, d.h. über eine einzige Kante zu erreichen. Dann sind wir schnell fertig, wenn nicht, müßten wir uns die Nachfolger der Nachfolger anschauen, usw.
Deshalb liegt es nahe, sich die Nachfolger anzuschauen. Dafür führen wir folgenden Begriff ein:

Defnition 2. *Die Menge aller Nachfolger eines Knotens $k \in V$, also alle Knoten, die über eine Kante direkt von k erreichbar sind, heißt* ***Rand*** *des Knotens k*

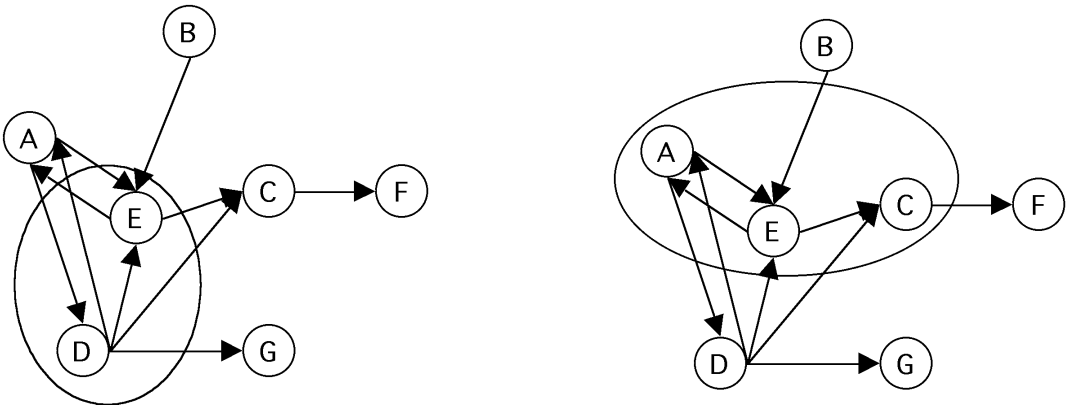

Abb. 6: Rand von A (links) und Rand von E (rechts) in G_0

Wir benötigen eine Funktion

```
;rand: knoten graph --> Liste (der Nachfolger)
```

die uns in einem Graphen alle Nachfolger zu einem gegebenen Knoten liefert, etwa in Listenform, also:

```
(define (rand knoten graph)
  ...)
```

und erwarten z.B. für G_0:

```
> (rand 'A g0)  -->   (list 'E 'D)
> (rand 'B g0)  -->   (list 'E)
> (rand 'C g0)  -->   (list 'F)
> (rand 'D g0)  -->   (list 'A 'C 'E 'G)
> (rand 'E g0)  -->   (list 'A 'C)
> (rand 'F g0)  -->   empty
> (rand 'G g0)  -->   empty
```

Dazu muss `rand` die Liste der Kanten des Graphen durchlaufen, die Nachfolger des Anfangsknotens finden und in eine Liste schreiben.
Da die Funktion lediglich die Kantenliste `(graph-kanten g)` des Graphen konsumieren muss, lassen wir das eine Hilfsfunktion

```
;rand-hilfe: knoten liste --> liste
```

erledigen und erhalten

```
(define (rand knoten graph)
  (rand-hilfe knoten (graph-kanten graph)))
```

Wenn der Graph nicht leer ist, gibt es beim Vergleich des (Ausgangs-)knotens mit den Anfangsknoten `(kante-ak <kante>)` der einzelnen Kanten zwei Fälle: Entweder Gleichheit oder nicht, damit lautet das Gerüst

```
(define (rand-hilfe knoten liste)
  (cond
    ((empty? liste) empty)
    ((equal? knoten (kante-ak (first liste)))
     ...
    (else
    ...)))
```

Bei Gleichheit wird der Endknoten der Kante `(kante-ek (first liste))` in die Randliste aufgenommen und die Funktion ruft sich mit dem Rest Kantenliste selbst auf, bei Ungleichheit ruft sie sich lediglich selbst mit dem Rest auf, also insgesamt

```
(define (rand-hilfe knoten liste)
  (cond
    ((empty? liste) empty)
    ((equal? knoten (kante-ak (first liste)))
     (cons (kante-ek (first liste)) (rand-hilfe knoten (rest liste))))
    (else
     (rand-hilfe knoten (rest liste)))))
```

Damit können wir die Frage nach einer Strategie für `(erreichbar? 'A 'F g0)` , also für die Erreichbarkeit von F, ausgehend von Knoten A, mit Hilfe von Rändern beantworten: F liegt im Rand von B, B liegt im Rand von von C, C liegt im Rand von E, und E liegt im Rand von A.
Oder anders: Ist der Zielknoten nicht im Rand vom Anfangsknoten, dann fragen wir, ob der Zielknoten in den Rändern der Knoten des ersten Randes liegt, usw. Also müssen wir die Ränder von Rändern von Rändern ... untersuchen usw. usw., bis F auftaucht oder ein Rand leer ist.

Leider hat dieses Vorgehen einen Haken: Wie wir z.B. am Graph G_0, insbesondere an Abb. 6 sehen, liegt D im Rand von A, andererseits liegt A im Rand von E, und E liegt wiederum im Rand von D, d.h. es liegt ein *Zyklus* vor! Im diesem Fall gerät das o.a. Verfahren, die Ränder der Ränder usw. zu untersuchen, in eine Endlosschleife.

Was ist zu tun? Damit nicht bereits untersuchte Knoten erneut untersucht werden, müssen sie ausgesondert werden: dies kann mit einem Parameter `schon-besucht` umgesetzt werden, den wir der Hilfsfunktion

```
;erreichbar?-hilfe: anfangsknoten zielknoten graph schon-besucht
```

zuordnen.
Da am Anfang noch kein Knoten besucht ist, wird die Hilfsfunktion mit dem Wert `empty` für `schon-besucht` aufgerufen:

```
(define (erreichbar? anfangsknoten zielknoten graph)
  (erreichbar?-hilfe anfangsknoten zielknoten graph empty))
```

Die Hilfsfunktion muss drei Fälle unterscheiden:

1. wenn ein Knoten schon besucht ist, dann Ende
2. wenn Anfangsknoten = Zielknoten, dann Ende
3. sonst muss der Rand des Knoten näher untersucht werden

Wir erhalten als Gerüst:

```
(define (erreichbar?-hilfe anfangsknoten zielknoten graph schon-besucht)
  (cond
    (...<anfangsknoten schon-besucht> --> false)
    (...<anfangsknoten = zielknoten> --> true)
    (else
     ...<Untersuchung des Randes des Anfangsknotens>...)))
```

Die beiden ersten Fälle sind schnell erledigt; für den dritten Fall muss zunächst der Rand des Knotens ermittelt werden und als Zwischenergebnis gemerkt werden. Dafür machen wir die lokale Definition

```
(local ((define rand-aktuell (rand k g)))
        ...)
```

Bei der Untersuchung von `rand-aktuell` muss wieder eine Fallunterscheidung gemacht werden:

1. der Zielknoten liegt in diesem Rand, dann Ende
2. wenn nicht, dann werden alle Knoten des Randes (eine Liste) untersucht und der aktuelle Knoten ausgesondert, indem er der Liste `schon-besucht` zugefügt wird, wofür wir eine neue Hilfsfunktion benötigen

Die neue Hilfsfunktion unterscheidet sich von `erreichbar?-hilfe` dadurch, dass sie als ersten Parameter nicht einen einzelnen (Anfangs-)Knoten, sondern eine Liste von Knoten konsumiert:

```
;erreichbar?-von-liste: liste knoten graph schon-besucht --> BOOLEsch
```

Somit ergibt sich

```
(define (erreichbar?-hilfe k z g schon-besucht)
  (cond
    ((member k schon-besucht) false)
    ((equal? k z) true)
    (else
     (local ((define rand-aktuell (rand k g)))
       (cond
         ((member z rand-aktuell) true)
         (else
          (erreichbar?-von-liste rand-aktuell z g (cons k schon-besucht))))))))
```

Die Hilfsfunktion `erreichbar?-von-liste` wird daher mit `rand-aktuell` für den Parameter `liste` aufgerufen und `schon-besucht` wird durch `(cons k schon-besucht)` ersetzt.
Da die Funktion eine Liste (von Knoten) verbraucht, liegt eine strukturelle Rekursion vor, und wir haben das Gerüst:

```
(define (erreichbar?-von-liste liste z g schon-besucht)
  (cond
    ((empty? liste) false)
    (........)
    (else
     ........)))
```

In der ersten Ellipse muss geprüft werden, ob der Zielknoten vom ersten Knoten des Randes, also vom ersten Element der Liste, aus erreichbar ist. Das kann mit der schon definierten Funktion `erreichbar?-hilfe` geschehen: `((erreichbar?-hilfe (first liste) z g schon-besucht) true)` .
In der zweiten Ellipse, also der Alternative, ruft sich die Funktion mit dem restlichen Rand auf und sondert den aktuellen wieder aus in `schon-besucht` .
Wir fügen alles zusammen:

```
(define (erreichbar?-von-liste liste z g schon-besucht)
  (cond
    ((empty? liste) false)
    ((erreichbar?-hilfe (first liste) z g schon-besucht) true)
    (else
     (erreichbar?-von-liste (rest liste) z g (cons (first liste) schon-besucht)))))
```

Anmerkung
Es fällt auf, dass im Rumpf der Funktion `erreichbar?-hilfe` die Funktion `erreichbar?-von-liste` aufgerufen wird und umgekehrt: Wir haben zwei Funktionen, die wechselseitig aufrufen, man spricht auch von *mutueller* Rekursion.

3.4 Wege zum Ziel

Nachdem die Frage nach der Erreichbarkeit in einem Graphen geklärt ist, kehren wir zur Ausgangsfrage

Wie kommt man von A nach B in einem Graphen?

zurück, d.h. man benötigt zur Lösung dieser Frage keine BOOLEsche Funktion, sondern eine Funktion, die eine der Liste der möglichen Wege liefert. Es liegt nahe, einen einzelnen Weg selbst auch als Liste, und zwar der besuchten Knoten, zu modellieren:

```
;alle-wege: start ziel graph --> wegliste
```

Für unseren Mustergraphen

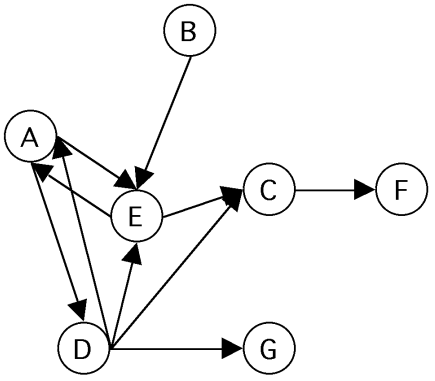

Abb. 7: Muster-Graph G_0

erwarten wir z.B.:

```
(alle-wege  'A 'B g0) --> empty
(alle-wege  'A 'E g0) --> (list (list 'A 'D 'E) (list 'A 'E))
(alle-wege  'A 'F g0) --> (list (list 'A 'D 'E 'C 'F) (list 'A 'E 'C 'F) (list 'A 'D 'C 'F))
```

Für den Sonderfall *start* = *ziel* genügt z.B. die Ausgabe `(alle-wege 'A 'A g0) --> (list 'A)`.
Die Vorgehensweise ähnelt der von `erreichbar?` insofern, dass zunächst der Rand des Anfangsknotens durchsucht wird. Für den Fall, dass ein Randknoten=Zielknoten ist, also ein Weg gefunden wurde, muss sich die Funktion diesen Weg als Liste von Knoten merken, was sinnvoller Weise durch einen zusätzlichen Parameter, etwa durch `zielwege` geschehen kann. Die restlichen Randknoten bilden mit dem Anfangsknoten in Form von Kanten (Teil-)Wege, die noch weiter untersucht werden müssen. Auch diese muss sich die Funktion merken, etwa in Form des Parameters `wegliste`. Für diese Aufgabe eignet sich eine Funktion

```
;alle-wege-hilfe: ziel wegliste graph zielwege --> allewege
```

die von `alle-wege` aufgerufen wird, wobei `zielwege` den Anfangswert `empty` und, da jeder Weg den Anfangsknoten `start` enthält, hat `wegliste` den Anfangswert `(list (list start))`:

```
(define (alle-wege start ziel graph)
  (alle-wege-hilfe ziel (list (list start)) graph empty))
```

Das Gerüst von `alle-wege-hilfe` ergibt sich aus folgender Überlegung: Ist ein Randknoten=Zielknoten, dann

- wird der Weg bis zu diesem Knoten der Liste `zielwege` hinzugefügt
- die restlichen Randknoten, also die restlichen (Teil-)Wege werden untersucht

Ist Randknoten $\neq$ Zielknoten, dann

- wird eine neue `wegliste` gebildet, die sich aus den restlichen, noch nicht untersuchten Wegen und den neuen, längeren Wegen zusammensetzt, die sich daraus ergeben, dass man dem aktuellen Weg jeweils die Kanten zu den Nachfolgern hinzufügt

```
(define (alle-wege-hilfe ziel wegliste graph zielwege)
( ......
  (cond
    ((equal? einknoten ziel)
     (alle-wege-hilfe ziel
                      (rest wegliste)
                      graph
                      <... füge weg zielwege hinzu>...)))
  (else
   (alle-wege-hilfe ziel
                    <...erstelle neue wegliste...>
                    graph
                    zielwege))))
```

Die Terminierung ist dadurch sichergestellt, dass die Wegliste irgendwann leer ist, und zwar deshalb, weil der erste rekursive Aufruf stets mit dem Rest der Wegliste erfolgt und beim zweiten Aufruf irgendwann keine neuen Teilwege mehr hinzugefügt werden können.
Der Zugriff auf den ersten Knoten erfolgt dadurch, dass aus der Wegliste der erste Weg `einweg` und daraus der erste Knoten `einknoten` lokal definiert werden. Somit ergibt sich für die erste Ellipse im Gerüst:

```
........
  (cond
    ((empty? wegliste) zielwege)
    (else
     (local
         ((define einweg (first wegliste))
          (define einknoten (first einweg)))
  ........
```

Das Hinzufügen eines Zielwegs zur Liste der Zielwege geschieht einfach durch `(cons (reverse einweg) zielwege))`, wobei die Liste aus optischen Gründen umgedreht wird.
Das Erstellen der neuen Wegliste beim zweiten Selbstaufruf (dritte Ellipse im Gerüst) gestaltet sich etwas komplexer, da sie sich aus zwei Teilen zusammensetzt: Zum einen aus den noch nicht untersuchten (Teil-)Wegen, also `(rest wegliste)`, zum anderen aus den neuen (Teil-)Wegen, die mit Hilfe der Funktion

```
;neue-wege: einweg einknoten graph --> Liste neuer Wege
```

erhält. Damit lautet die dritte Ellipse

```
(append (rest wegliste) (neue-wege einweg einknoten graph))
```

Die neuen Wege ergeben sich daraus, dass man einen Weg dadurch zu neuen Wegen verlängert, indem jeweils die Kante zu einem Nachfolgerknoten hinzufügt, also benötigt man die schon definierte Funktion `rand` mit dem Aufruf `(rand knoten graph)`.
An dieser Stelle ist Vorsicht geboten: Es muss vermieden werden, dass dabei Knoten berücksichtigt werden, die bereits in einem anderen Weg vorkommen! Das kann durch eine Funktion

```
;nicht-in-liste: liste1 liste2 --> liste
```

sicher gestellt werden, die diejenigen Elemente von `liste1` auflistet, die nicht in `liste2` enthalten sind. Da diese Funktion eine Liste durchläuft, ergibt sich sofort das Gerüst:

```
(define (nicht-in-liste liste1 liste2)
  (cond
    ((empty? liste1) empty)
    ((...<ist nicht in liste2>...)
     (...<füge hinzu>... (nicht-in-liste (rest liste1) liste2)))
    (else
     (nicht-in-liste (rest liste1) liste2))))
```

Für die erste Ellipse kann die vordefinierte Funktion `member` benutzt werden, für die zweite Ellipse `(cons ...)`.
Insgesamt ergibt sich:

```
(define (nicht-in-liste liste1 liste2)
  (cond
    ((empty? liste1) empty)
    ((not (member (first liste1) liste2))
     (cons (first liste1) (nicht-in-liste (rest liste1) liste2)))
    (else
     (nicht-in-liste (rest liste1) liste2))))
```

Im vorliegenden Fall müssen also die Knoten des neuen Randes ausgesondert werden, die bereits im vorliegenden Weg enthalten sind, also durch den Aufruf `(nicht-in-liste (rand knoten graph) weg)`, wobei man die so erhaltene Liste der Übersichtlichkeit halber einer lokalen Variablen zuweisen kann:

```
(define echte-nachbarn (nicht-in-liste (rand knoten graph) weg))
```

Die oben beschriebene Erzeugung der neuen Weg kann man elegant durch

```
(map (lambda (n) (cons n weg)) echte-nachbarn)
```

erreichen und erhält zusammengefasst:

```
(define (neue-wege weg knoten graph)
  (local
      ((define echte-nachbarn (nicht-in-liste (rand knoten graph) weg)))
    (map (lambda (n) (cons n weg)) echte-nachbarn)))
```

Damit sind alle Teile der Funktion `alle-wege` besprochen.

An dem Beispiel-Aufruf

```
(alle-wege 'D 'F g0) --> (list (list 'D 'A 'E 'C 'F) (list 'D 'E 'C 'F) (list 'D 'C 'F))
```

kann man an der Reihenfolge der Wege deutlich die Breitensuche erkennen: Da mittels `(cons (reverse einweg) zielwege))` ein neuer (Ziel-)Weg stets am Anfang der Zielweg-Liste eingefügt wird, werden die Wege mit wachsender Länge gefunden.

3.5 Kürzeste Wege

Wenn es Wege von A nach B gibt, ist es klar, dass es mindestens einen kürzesten Weg gibt. Spielt die Wahl es kürzesten Weges keine Rolle, kann man sich auf die Suche nach *einem* kürzesten Weg beschränken.

Von der gesuchten Funktion

```
;min-weg: start ziel graph --> weg
```

erwarten wir z.B.

```
(min-weg  'A 'B g0) --> empty
(min-weg  'A 'E g0) --> (list 'A 'E)
(min-weg  'A 'F g0) --> (list 'A 'D 'C 'F)
```

Ein Vergleich mit `;alle-wege` hilft uns bei der Konstruktion der gesuchten Funktion:

1. Sobald in `;alle-wege-hilfe` durch die Abfrage `(equal? einknoten ziel)` der Fall `start = ziel` eintritt, müssen weder der Rest der Wegliste durchlaufen werden noch dieser Zielweg der Liste `zielwege` hinzugefügt werden; statt dessen wird dieser Weg als Funktionswert ausgegeben und die Funktion terminiert. Somit liefert die Funktion keine Wegliste, also eine Liste von Listen, sondern einen Weg, also eine einfache Liste von Knoten.
2. Dass im Falle 1. ein kürzester Weg gefunden wurde, ist dadurch sichergestellt, dass der zweite Selbstruf nicht mehr stattfindet, der ja ohnehin aufgrund der Funktionsweise von `;neue-wege` nur längere Wege liefert.

Insgesamt genügen folgende Änderungen am Quellcode:

- `;alle-wege` wird in `;min-weg` umbenannt
- `;alle-wege-hilfe` wird in `;min-weg-hilfe` umbenannt
- der Parameter `zielwege` entfällt
- der Selbstaufruf nach der Abfrage `(equal? einknoten ziel)` wird durch die Ausgabe des kürzesten Weges `(reverse einweg)` (in der richtigen Reihenfolge der Knoten) ersetzt

Im Überblick:

```
;min-weg: ziel wegliste graph --> weg (als Liste)
(define (min-weg-hilfe ziel wegliste graph)
  (cond
    ((empty? wegliste) empty)
    (else
     (local
         ((define einweg (first wegliste))
          (define einknoten (first einweg)))
       (cond
         ((equal? einknoten ziel)
          (reverse einweg))
         (else
          (min-weg-hilfe ziel
                         (append (rest wegliste)
                                 (neue-wege einweg einknoten graph))
                         graph)))))))

;min-weg: knoten knoten graph --> weg (als Liste)
(define (min-weg start ziel graph)
  (minweg-hilfe ziel (list (list start)) graph))
```