Kapitel 7
Grafen

7.1 Grafen und ihre Speicherung

Zur Beantwortung der Frage „was ist ein Graf?“ führt man sich am Besten einige praktische Anwendungsbeispiele vor Augen. Das wohl jedem bekannte Beispiel für den Einsatz von Grafen ist das Finden eines Weges zwischen verschiedenen Orten mit Hilfe eines Navigationssystems. Auf einer gedachten Karte stellen die Orte einzelne Punkte dar, die durch Straßen miteinander verbunden sind. Das Navigationssystem hat nun die Aufgabe, eine geeignete Strecke zwischen zwei Punkten unter Berücksichtigung einer Reihe von Nebenbedingungen zu ermitteln. Das Ganze – Orte, Straßen und Eigenschaften von Orten und Straßen – nennt man einen Grafen.

Mathematisch betrachtet bestehen Grafen aus einer Menge von Ecken (Knoten) $\boldsymbol{E}$, einer Menge von jeweils zwei Ecken verbindenden Kanten $\boldsymbol{K}$, ggf. ergänzt durch eine Bewertung von Ecken $\boldsymbol{W_E}$ und einer oder mehreren Bewertungen von Kanten $\boldsymbol{W_K}$, den oben angesprochenen Eigenschaften. Kanten werden als Eckenpaare $a_{jk} = (e_j,e_k)$ angegeben, wobei e_i die Ausgangsecke, e_k die Endecke genannt wird. Eine Kante kann nur von der Ausgangsecke zur Endecke durchlaufen werden.[1] Eine Bewertung ist die Zuordnung einer Zahl zu einer Ecke oder Kante.
Sind für einen Grapfen n Ecken gegeben, so kann man die Kanten in einer Matrix speichern. Für diese gilt:

$$A = (a_{ik}),a_{ik} = \begin{cases} B & \text{Kante mit Bewertung } B \text{ von Ecke } i \text{ zu Ecke } k \\ 0 & \text{Ecken sind nicht verbunden} \end{cases}$$

Besitzen die Kanten keine Bewertung, wird $B = 1$ gesetzt. In der Regel sind von den n^2 möglichen Kanten aber nur wenige tatsächlich vorhanden, so dass sich die Verwendung der bereits definierten schwach besetzten Matrizen empfiehlt.

[1] Vergleiche jedoch die anwendungstechnische Unterscheidung zwischen gerichteten und ungerichteten Grafen. Bei ungerichteten Grafen sind die beiden gegenläufig verlaufenden Kanten zu einer verschmolzen.

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_7, © Springer-Verlag Berlin Heidelberg 2010

Um etwas mehr Funktionalität einzubauen, definieren wir den Datentyp `GrafType`. Anwendungstechnisch kann zwischen verschiedenen Graftypen unterschieden werden:[2]

- **ungerichtet-unbewertet**. Eine Kante in Grafen dieses Typs hat stets die Bewertung Eins und kann in beide Richtungen durchlaufen werden. Kanten werden vereinbarungsgemäß in der Form $a_{ik}, i < k$ gespeichert, d.h. die Speicherstruktur enthält jede Kante nur einmal (Halbierung des Speicherplatzbedarfs). Um alle in eine Ecke *ecke* laufende Kanten zu ermitteln, muss über `init_2nd(ecke,k)` und `init_1st(k,ecke)` durch die Kantenmatrix iteriert werden.
- **gerichtet-unbewertet**. In diesen Grafen wird zwischen eingehenden und ausgehenden Kanten unterschieden. Jede Kante wird einzeln gespeichert, d.h. bidirektionale Kanten sind jeweils als eingehende und ausgehende Kante vorhanden. Um alle von einer Ecke *ecke* ausgehenden Kanten zu ermitteln, muss über `init_2nd(ecke,k)` iteriert werden, die eingehenden Kanten erhält man durch die Iteration `init_1st(k,ecke)`.
- **ungerichtet-bewertet**. Wie ungerichtet-unbewertet, nur dass nun von Eins verschiedene Bewertungen zulässig sind.
- **gerichtet-bewertet**. Wie gerichtet-unbewertet, nur dass nun von Eins verschiedene Bewertungen zulässig sind.

```
enum GTypes {    udir_uval=0,
              dir_uval =1,
              udir_val =2,
              dir_val =3,
              dir_mask =1,
              val_mask =2 };
```

Für die Eckenbewertung wird zusätzlich eine ganzzahlige Eckenliste definiert.[3] Mit einigen sinnvollen Typdefinitionen wird GrafType zu

```
class GrafType {
public:
   typedef pair<unsigned int, unsigned int>  kante;
   typedef list<kante>kantenliste;
   typedef vector<kante>gradliste;
   ...
```

---

[2] Je nach Anwendung kann man natürlich weitere Typen definieren, etwa solche mit mehreren Bewertungen pro Kante oder Ecke oder mit Bewertungsfunktionen. Wir beschränken uns zunächst auf das, was mit einer einzelnen schwach besetzten Matrix als Basis realisiert werden kann.

[3] Ecken mit Selbstbezug, d.h. Kanten des Typs (ecke,ecke), sind nicht vorgesehen bzw. müssen mittels der Eckenbewertung in Algorithmen berücksichtigt werden. Mehrfachbewertungen von Kanten nach unterschiedlichen Kriterien können durch Mehrfachinstanziierung eines Grafen mit unterschiedlichen Bewertungen, Nutzen der Kantenbewertung als Schlüssel in einer externen Liste oder durch Änderung des Template-Parameters des SBMatrix-Attributs realisiert werden. Im letzten Fall ist allerdings auch eine weitgehende Umprogammierung der Grafenklasse notwendig.

```
protected:
    GTypes typ;
    vector<int>edges;
    SBMatrix<int>adjazenz;
    friend class GrafTypeIterator;
};//ennd class
```

Je nach Algorithmus kann es sinnvoll sein, einen Grafen eines bestimmten Typs in einen einfacheren Grafen zu überführen. Bei Konversionen zwischen den Typen gilt

- **ungerichtet->gerichtet**. Die Kantenanzahl wird verdoppelt, d.h. $a_{ki} = a_{ik}$, $i < k$ gesetzt.
- **gerichtet->ungerichtet.** Alle Kanten werden auf beide Richtungen eingerichtet, d.h. ist $a_{ik}, i < k$ vorhanden, wird a_{ki} gelöscht, sonst vorher $a_{ik} = a_{ki}$, $i < k$ gesetzt. Sind bewertete Kanten in beiden Richtungen vorhanden, wird die Übernahme des Mittelwerts in den ungerichteten Grafen vereinbart.
- **bewertet->unbewertet.** Die Bewertungen werden auf Eins zurückgesetzt.
- **unbewertet->bewertet.** Änderungen der Kantenelemente sind nicht notwendig.

Der Code für sämtliche Umwandlungen beginnt mit einer Feststellung der verschiedenen Umwandlungseigenschaften wie Richtung und Bewertung von altem und neuen Graf.

```
bool GrafType::set_type(GTypes tp){
  unsigned int i,j;
  bool loop, ndir,adir,nval,aval;
  kantenliste kl; kantenliste::iterator kt;
  if(tp < udir_uval || dir_val<tp)
    return false;
  ndir=((int)tp&(int)dir_mask)!=0;
  adir=((int)typ&(int)dir_mask)!=0;
  nval=((int)tp&(int)val_mask)!=0;
  aval=((int)typ&(int)val_mask)!=0;
```

Bei Löschen der Kantenrichtungen wird dafür gesorgt, dass im Ergebnisgrafen nur Matrixelemente mit $a_{ki} = a_{ik}$, $i < k$ vorhanden sind. Von gegenläufigen Kanten im gerichteten Grafen wird eine gelöscht, bei falscher Größenreihenfolge der Indizes erfolgt ein Umkopieren:

```
typ=tp;
if(adir && !ndir){
  kl=get_kantenliste();
  for(kt=kl.begin();kt!=kl.end();kt++){
    if(kt->first>kt->second){
      if(adjazenz(kt->second,kt->first)!=0)
        adjazenz.set(kt->second,kt->first,
```

```
              (adjazenz(kt->first,kt->second)+
               adjazenz(kt->second,kt->first))/2);
      else
        adjazenz.set(kt->second,kt->first,
              adjazenz(kt->first,kt->second));
        adjazenz.set(kt->first,kt->second,0);
      }//endif
  }//endfor
}//endif
```

Bei Einfügen der Richtungskennung wird die Anzahl der Kanten verdoppelt.

```
if(ndir && !adir){
  kl=get_kantenliste();
  for(kt=kl.begin();kt!=kl.end();kt++){
    adjazenz.set(kt->second,kt->first,
                 adjazenz(kt->first,kt->second));
  }//endfor
}//endif
```

Abschließend werden überzählige Kantenbewertungen gelöscht

```
  if(aval && !nval){
    for(loop=adjazenz.init(i,j);loop;
        loop=adjazenz.next(i,j))
        adjazenz.set(j,i,1);
  }//endif
  return true;
}//end function
```

Die Abfrage und Änderung von Typ, Eckenbewertung und Kanten erfolgt durch entsprechende Methodensätze `get_xxx`, `set_xxx` und `del_xxx`. Wobei `xxx` stellvertretend für Ecke oder Kante steht. Die Vergrößerung eines Grafen ist durch Anfügen weiterer Kanten zwischen vorhandenen Ecken bzw. durch Hinzufügen weiterer Ecken an das Ende der Eckenliste möglich.[4] Die Implementation dieser relativ einfachen Funktionen sei Ihnen überlassen.

Das Löschen einer Ecke kann natürlich auch inmitten der Eckenliste erfolgen. Da sich hierbei die Eckenliste verkürzt und die auf die gelöschte Ecke folgenden Ecken eine andere Indexnummer erhalten, ist die Kantenliste anzupassen:

- Alle Kanten a_{Ki} , a_{iK} sind zu entfernen.
- Alle Kanten a_{ik} mit $(i > K) \vee (k > K)$ sind zu Kopieren und zu Löschen:

$$a_{i'k'} = a_{ik} \text{ , } x' = \begin{cases} x - 1 \text{ wenn } x > K \text{ , } a_{ik} = 0 \\ x \text{ sonst} \end{cases}$$

[4] Einfügen einer Ecke in der Mitte der vorhandenen muss nicht vorgesehen werden.

Dies führt zu der Implementation

```
void GrafType::del_ecke(unsigned int ecke){
   SBMatrix<int>tmp; unsigned int i,j,i1,j1;
   bool loop;

   if(ecke>=edges.size()) return;
   for(i=0;i<edges.size();i++){
      if(i==ecke) continue;
      i1=i-(int)(i>ecke);
      for(loop=adjazenz.init_2nd(i,j);loop;
          loop=adjazenz.next(i,j)){
        if(j==ecke) continue;
        j1=j-(int)(j>ecke);
        tmp.set(i1,j1,adjazenz(i,j));
     }//endfor
  }//endfor
  adjazenz=tmp;
  edges.erase(edges.begin()+ecke);
}//end function
```

Zusätzlich erlauben die Methoden

```
eckenliste  grad_ecken() const;
kantenliste get_outbound_kliste(unsigned int) const;
kantenliste get_inbound_kliste(unsigned int) const;
kantenliste get_kantenliste() const;
```

die Ermittlung von Listen der ein- und ausgehenden Kanten sowie der Anzahlen der ein- und ausgehenden Kanten an einer Ecke in der Datenform

```
typedef pair<unsigned int, unsigned int>  kante;
typedef list<kante>                       kantenliste;
typedef vector<kante>                     eckenliste;
```

Die Kantenlisten werden hierbei als verkettete Listen definiert, da sie sequentiell bearbeitet *(Kanten in der Reihenfolge, in der sie eine Weg definieren)* und in verschiedenen Algorithmen aus mehreren Listen zusammengesetzt werden.

Aufgabe. Implementieren Sie die Klasse `GrafTyp` vollständig. Vergessen Sie auch nicht geeignete `to_string` und `from_string`-Methoden.

7.2 Arten des Eckenverbundes

Wir untersuchen zunächst einige grundlegende Eigenschaften eines Grafen. Meist ist man über eine Eigenschaftsaussage hinaus auch daran interessiert, wie denn diese Eigenschaft genau aussieht. Diese Aussage geben die ersten Algorithmen noch nicht, sind aber wichtige Vorstufen zum Gewinnen der Aussagen.

7.2.1 Distanzlisten

Eine erste Frage, die man an einen Grafen richten kann, ist diese:

> *Ist eine Ecke von einer gegebenen Anfangsecke aus erreichbar, und wenn ja, wie viele Kanten müssen durchlaufen werden?*

Für die Antwort implementieren eine Methode, die als Rückgabewert einen Vektor mit ganzzahligen Komponenten ausgibt, deren Zahlenwert die Kantenanzahl zum Erreichen der jeweiligen Ecke angibt (also die Frage für sämtliche Ecken beantwortet). Ist eine Ecke nicht erreichbar, wird -1 ausgegeben.

Der Algorithmus ist denkbar einfach:

(a) Die Eckenliste wird mit –1 initialisiert, die Startecke erhält den Wert Null, der Zähler ebenfalls.
(b) Es wird in der Eckenliste eine Ecke ausgewählt, deren Zählwert dem Zähler entspricht (im ersten Durchgang ist das nur die Startecke). Zu dieser Ecke wird die Liste der ausgehenden Kanten ermittelt und jede Endecke einer Kante mit (Zähler+1) markiert, sofern die Markierung der Ecke -1 ist (Ecken, die einen von -1 verschiedenen Wert aufweisen, sind bereits auf einem kürzeren Weg erreichbar und werden nicht erneut markiert).
(c) Das Verfahren wird bei (b) fortgesetzt, wenn weitere Ecken mit einer dem Zähler entsprechenden Markierung existieren.
(d) Ist keine Ecke mit Zählermarkierung vorhanden, wird der Zähler um eine Einheit erhöht, und das Verfahren bei (b) fortgesetzt. Wird dabei gar keine Ecke gefunden, ist der Algorithmus beendet.

Da dieser Algorithmus, von der Startecke ausgehend, den Grafen über alle vorhandenen nächsten Kanten gewissermaßen flutet, heißt dieser Algorithmus „Breitensuchalgorithmus“. Er bestimmt zwar den Abstand von Ecken, gibt jedoch zunächst keine Auskunft darüber, welche Kanten durchlaufen werden müssen, um von einer Ecke zu einer anderen zu kommen.

Die Methode löst noch einige weitere Aufgaben, auf die wir erst später zu sprechen kommen, aber im Code bereits berücksichtigen. Für die erste Aufgabe relevant sind die ersten drei Übergabeparameter. Überlesen Sie also im ersten Durchlauf den zu den anderen gehörenden Code.

```
bool Breitensuche(unsigned int ecke,
                  GrafType const& graf,
                  Eckenliste& elist,
                  GrafType::kantenliste* klist=0,
                  bool find_cycle=false){
  int count;
  bool loop,udir;
  GrafType::kantenliste kl;
  GrafType::kantenliste::iterator ki;
```

```
  Eckenliste::iterator it;
  elist.clear();
  elist.resize(graf.anzahl_ecken(),-1);
  if(klist) klist->clear();
```

Die Hauptschleife zählt die Entfernungen hoch und bricht ab, sobald in einem Durchlauf keine Knoten mit der passenden Markierung gefunden worden sind. Die logische Variable `loop` übernimmt die Feststellung in der inneren Schleife, die über alle Knoten mit der gesuchten Markierung verläuft. Wir machen hier Gebrauch von STL-Algorithmen, was den Kopf der `for`-Schleife auf drei Zeilen aufbläst, aber in Summe immer noch etwas übersichtlicher ist als eine Standardsuche.

```
elist[ecke]=0;
for(count=0, loop=true; loop ; count++ ){
  for(it=find(elist.begin(),elist.end(),count),
                                  loop=false ;
      it!=elist.end();
      it=find(++it,elist.end(),count),loop=true){
```

Die Verwendung von Iteratoren erfordert eine Umrechnung in ganzzahligen Indizes mittels der STL-Methode `distance`. Für die ermittelte Ecke wird zunächst eine Liste ausgehender Kanten erzeugt.

```
ecke=distance(elist.begin(),it);
kl=graf.get_outbound_kantenliste(ecke);
```

Soweit es den in diesem Teilkapitel diskutierten Algorithmus betrifft, wird in der Eckenliste die Markierung erzeugt, sofern noch keine vorhanden ist. Der `else`-Zweig ist hier zunächst uninteressant und wird auch nicht durchlaufen, da bei dieser Aufgabe die Methode mit `find_cycle=false` aufgerufen wird.

```
      for(ki=kl.begin();ki!=kl.end();ki++){
        if(elist[ki->second]==-1){
          elist[ki->second]=count+1;
          if(klist) klist->push_back(*ki);
        } else if(find_cycle &&
                    find(klist->begin(),
                         klist->end(),
                         GrafType::kante(
                            ki->second,ki->first))
                    ==klist->end()) {
          return true;
        }//endif
      }//endfor
    }//endwhile
  }//endfor
  return false;
}//end function
```

7.2.2 Verbundenheit von Grafen

Die Distanzliste gibt bei ungerichteten Grafen auch eine Auskunft darüber, ob die Grafen verbunden, d.h. alle Ecken untereinander erreichbar sind. In einem gerichteten Grafen erhält man diese Aussage nicht unmittelbar, da durchaus Ecke A von Ecke B aus nicht erreichbar sein kann, die Distanzliste also eine -1 enthält, während von A selbst ein Weg nach B führt. Wir müssen in gerichteten Grafen daher notfalls Distanzlisten für alle Ecken erstellen, um zu einer definitiven Aussage zu kommen. Unterschieden wird meist zwischen mehreren Typen der Verbundenheit

1. In **schwach verbundenen Grafen** sind alle Ecken sind durch Kanten miteinander verbunden. Ein Test kann einfach durchgeführt werden, indem ein gerichteter Graf in einen ungerichteten überführt wird. Nach einem Distanztest mit einer beliebigen Ecke müssen alle Listenelement von -1 verschieden sein. Funktion:

   ```
   bool connect_weak(GrafType const&);
   ```

2. In **stark verbundenen Grafen** muss jede Ecke von jeder anderen aus erreichbar sein. Ein Distanztest in einem gerichteten Grafen muss hierbei mit jeder Ecke ausgeführt werden und darf kein -1 − Ergebnis enthalten.[5] Funktion:

   ```
   bool connect_strong(GrafType const&);
   ```

3. In einem **verbundenen Grafen** genügt es, eine Ecke zu finden, von der aus alle anderen erreichbar sind. Diese Prüfung macht in der Regel den größten Aufwand.

Aufgabe. Implementieren Sie diese Algorithmen

7.2.3 Abspalten disjunkter Subgrafen

Stellt man fest, dass ein Graf nicht verbunden ist, kann es sinnvoll sein, den verbundenen Teil für folgende Algorithmen abzuspalten. Die Abspaltung eines disjunkten Untergrafen ist ebenfalls auf der Basis einer Distanzliste möglich. Dazu wird zunächst eine Kopie des Grafen sowie eine Distanzliste zur Bezugsecke erzeugt. Anschließend können im Ergebnisgrafen alle Ecken mit dem Abstand -1, im Restgrafen alles anderen Ecken entfernt werden.[6] Das Entfernen der Ecken muss dabei

[5] Dies ist ein einfacher, aus dem Distanzalgorithmus resultierender Algorithmus, der jedoch in dieser Form nicht der günstigste sein muss, da hierbei bereits geprüfte Bereiche des Grafen u.U. erneut geprüft werden.

[6] Auch dieser Algorithmus ist aufgrund des Löschaufwands im Ergebnisgrafen nicht optimal. Durch Kopieren der Kanten in einen leeren Grafen kann eine höhere Effizienz erreicht werden, jedoch ist dazu auch eine tabellengesteuerte Indexumsetzung notwendig.

von der Ecke mit dem höchsten Index bis zur Nullecke erfolgen, da sonst die Einträge der Distanzliste ihre Gültigkeit verlieren.

Aufgabe. Implementieren Sie einen Abspaltalgorithmus.

Der Nachteil dieses Verfahrens ist, dass die Indizes der Ecken der neuen Grafen nicht (so ohne weiteres) auf Indizes im alten Grafen zurückgeführt werden können. Wenn ein Bezug auf die ursprüngliche Indizierung notwendig ist, beispielsweise bei Navigationsproblemen, in denen Eigenschaften von Kanten und Knoten im weiteren Verlauf der Berechnung nachgeladen werden müssen, muss eine Tabelle mit den ursprünglichen Indizes mitgeführt werden. Dies kann ein Vektor sein, der zunächst in jedem Feld seinen Index auflistet, in dem aber parallel zum Löschen in den Grafen ein Index gelöscht wird, wodurch die oberen Indizes nach unten aufrücken. Da das Löschen der Ecken nur von oben nach unten erfolgen darf, bleibt auch diese Liste bei der Operation konsistent.

7.2.4 Zyklenfreie (Sub)Grafen

Ein geschlossener Weg innerhalb eines Grafen heißt Zyklus, und ein Weg ist geschlossen, wenn eine Ecke nach Durchlaufen einer beliebigen Anzahl von Kanten erneut erreicht werden kann. Diese Definition würde ungerichtete Grafen generelle als nicht zyklenfrei identifizieren, da eine Kante in beide Richtungen durchlaufen werden kann und bei Prüfung der nächsten Ecke die Ausgangsecke wieder indiziert wird. Auch gerichtete Grafen mit gegenläufigen Kanten zwischen zwei Ecken hätten so generell Zyklen. Um diese Fälle auszuschließen, legen wir fest, dass ein Zyklus mindestens drei Kanten aufweisen muss.

Die Prüfung, ob ein Graf Zyklen enthält, ist in der Erweiterung des Breitensuchalgorithmus enthalten: die untersuchten Kanten werden zusätzlich in der Kantenliste `klist`, die als vierter Übergabeparameter übergeben wird, gespeichert (siehe oben). Wird nun von einer neuen Kante eine Ecke indiziert, die bereits einen von -1 verschiedenen Wert enthält, ist möglicherweise ein Zyklus vorhanden. Die indizierende Kante sei a_{ik}. Um auszuschließen, dass es sich um einen 2er-Zyklus handelt, d.h. die Ecke i zuvor von der Ecke k aus indiziert wurde, muss zusätzlich nur überprüft werden, ob die Kante a_{ki} bereits in der Liste vorhanden ist. Ist dies der Fall, handelt es sich nicht um einen Zyklus, und die Suche kann fortgesetzt werden. Ist die Kante nicht vorhanden, so ist die Ecke zuvor auf einem völlig anderen Weg erreicht worden und es liegt ein Zyklus vor.

Die Prüfung kann in zwei Varianten ausgeführt werden:

- Bei Vorgabe einer Ecke wird überprüft, ob im Teilgrafen dieser Ecke ein Zyklus vorhanden ist. Funktionsaufruf:

```
bool cycle_free (unsigned int,GrafType const&);
```

- Ist dies nicht der Fall, müssen alle in den vorhergehenden Durchläufen noch nicht erreichten Ecken überprüft werden, wenn festgestellt werden soll, ob der Graf insgesamt Zyklen enthält. Funktionsaufruf:

```
bool cycle_free (GrafType const&) ;
```

Da der Breitensuchalgorithmus sämtliche Kanten aller von einer Ecke aus erreichbaren anderen Ecken untersucht, werden Zyklen in diesem Subgrafen sicher gefunden. Bei Auswahl einer erreichbaren Ecke als Startecke wird ein kleinerer Subgraf indiziert, der als Teilgraf des vorhergehenden ebenfalls keine Zyklen enthalten kann. Umgekehrt kann bei Auswahl einer nicht erreichbaren Ecke der bereits untersuchte Graf als Teilgraf auftreten und der neue Gesamtgraf Zyklen aufweisen, die außerhalb des Teilgrafen liegen.

> **Aufgabe.** Man könnte auch auf die Idee kommen, einfach nach einer bereits markierten Ecke zu suchen, die eine um 2 kleinere Markierung aufweist. Diese kann keine direkt verbundene Ecke sein und ist somit Bestandteil eines Zyklus. Konstruieren Sie ein Beispiel, das zeigt, dass man bei dieser Suchmethode Zyklen übersehen kann, also die Aussage „zyklenfrei“ erhält, obwohl dies nicht der Fall ist.

Anmerkung. Wie dem aufmerksamen Leser sicher nicht entgangen ist, ist diese Definition einen Zyklus in einem gerichteten Grafen keine Garantie dafür, dass auch ein Weg existiert, der von einem Knoten ausgehend auch wieder zu diesem zurück läuft. Für diese Aussage eines „durchlaufbaren Zyklus“ ist ein anderer Algorithmus anzuwenden, den wir uns weiter unten vornehmen.

7.3 Spannende Bäume

Ein zyklenfreier Graf wird Baum genannt. Auch wenn die meisten in der Praxis auftretenden Grafen keine Bäume sind, ist die Frage, ob man den Grafen auch durch einen Baum realisieren kann, von Interesse. Man denke beispielsweise wieder an das Navigationsproblem, bei dem nun für den Planer die Frage entsteht, wo er die Hauptverkehrsadern wie Autobahnen anlegen muss, um alle Städte erreichen zu können. Diese bilden dann einen Baum innerhalb des Straßennetzes.[7] Spannende Bäume eines Grafen sind mithin Subgrafen, aus denen so viele Kanten gestrichen worden sind, dass zwar weiterhin alle Ecken miteinander verbunden, aber keine Zyklen mehr vorhanden sind. Sind die Grafen verbunden, so enthält ein spannender Baum auch sämtliche Ecken des Grafen.

[7] Die Realität hat uns hier natürlich längst überholt, da auch das Autobahnnetz längst keine Baumstruktur mehr besitzt.

7.3.1 Breitensuche

Um einen spannenden Baum in einem verbundenen ungerichteten Grafen zu finden, kann man bei einer beliebigen Ecke mit der Auswahl der Kanten beginnen. Je nach ausgewählter Ecke findet man natürlich unterschiedliche spannende Bäume.
In einem gerichteten Grafen lassen sich zwei Strategien zur Auswahl von Bäumen verfolgen:

(a) Der Kanten werden wie in einem ungerichteten Baum ausgewählt, wobei bei Existenz der Kanten a_{ik} und a_{ki} eine der beiden zufällig ausgewählt wird.
Der so entstehende Baum ist spannend für den gesamten Grafen, jedoch ist im Mittel höchsten die Hälfte der Ecken von einer gegebenen Ecke aus erreichbar. Meist wird der spannende Baum daher als ungerichteter Graf berechnet. Funktion:

```
GrafType span_tree_broad (GrafType const&);
```

(b) Die Kanten werden unter Berücksichtigung der Richtung ausgewählt. Der Baum enthält alle von der vorgegebenen Wurzelecke aus erreichbaren Ecken und wird daher auch Wurzelbaum genannt. Die Funktion

```
GrafType root_tree_broad (unsigned int,
                                  GrafType const&) ;
```

berechnet den Wurzelbaum für eine gegebenen Ecke, die Funktion

```
GrafType root_tree_broad (GrafType const&) ;
```

den maximalen Wurzelbaum. Wenn der Baum nicht komplett spannend errichtet werden kann, ist das Ergebnis allerdings nur der größte mögliche Wurzelbaum.

Ein Algorithmus für die Berechnung solcher Bäume kann wieder vom Breitensuchalgorithmus abgeleitet werden. Wie man anhand der Beschreibung leicht überlegt, genügt es, die zu einer noch nicht markierten Ecke führende Kante in einer Liste abzulegen. Die Liste enthält am Schluss des Algorithmus sämtlich erfolgreich durchlaufene Ecken und kann in einem Grafenobjekt abgespeichert werden. Die Listennotierung haben wir bereits bei der Überprüfung auf Zyklenfreiheit eingeführt, so dass der Grundalgorithmus nicht mehr verändert, sondern nur noch mit `find_cycle=false` aufgerufen werden muss.

Aufgabe. Implementieren Sie die Algorithmen.

7.3.2 Tiefensuche

Während der Breitensuchalgorithmus iterativ in jedem Schritt die Nachfolgekonten aller bisherigen Endknoten untersucht, verfolgt der als alternative Ansatz des Tiefensuchalgorithmus rekursiv zunächst einzelne Wege bis zu deren Ende und setzt ggf. danach die Suche an bereits durchlaufenen Punkten mit Alternativrouten fort. Er liefert daher im Erfolgsfall nicht nur die Aussage, dass zwei Knoten miteinander verbunden sind, sondern auch gleich einen Weg in Form einer Liste der zu durchlaufenden Knoten dazu. Der wird aber nur in Bäumen die Länge besitzen, die der Breitensuchalgorithmus angibt, und bei nicht zyklenfreien Grafen aber meist wesentlich länger ausfallen.

Auch dieser Algorithmus beginnt bei einer beliebigen Ecke, verläuft hingegen in folgenden Stufen:

(a) Wähle eine beliebige Ecke aus und starte den Algorithmus mit der Ecke als Ausgangsecke.
(b) Wähle eine beliebige ausgehende Kante aus.
(c) Ist die Zielecke noch nicht markiert, markiere die Zielecke, speichere die Kante in der Kantenliste und rufe die Methode rekursiv mit der Zielecke als neuer Ausgangsecke aus.
(d) Ist die Zielecke markiert, suche eine andere ausgehende Kante aus und fahre bei (c) fort.
(e) Ist keine ausgehende Kante mehr vorhanden, verlasse die Methode.

Hierdurch gelangt der Algorithmus wieder an das Ende von (c), jedoch mit einer Ecke als Startecke, von der bereits mindestens eine Kante erfolgreich geprüft worden ist. Es werden nun weitere Kanten dieser Ecke geprüft, so dass der Graf im Erfolgsfall hier verzweigt.

```
void Tiefensuche_Baum(unsigned int ecke,
                      GrafType const& graf,
                      Eckenliste& elist,
                     GrafType::kantenliste* klist){
  GrafType::kantenliste kl;
  GrafType::kantenliste::iterator it;
  bool udir;

  udir=((int)graf.get_type()&(int)dir_mask)==0;
  kl=graf.get_outbound_kantenliste(ecke);
  for(it=kl.begin();it!=kl.end();it++){
    if(elist[it->second]==-1){
      elist[it->second]=elist[ecke]+1;
      if(klist) klist->push_back(*it);
      Tiefensuche_Baum(it->second,graf,
                       elist,klist);
    }//endif
```

```
  }//endfor
}//end function
```

Der Algorithmus zeigt im Vergleich zum Breitensuchalgorithmus noch einmal sehr schön den Unterschied zwischen einer Iteration und einer Rekursion auf und endet, wenn keine weitere Kante mit unmarkierter Zielecke gefunden werden kann. Der so ermittelte Baum besitzt in der Regel weniger Verzweigungen als der durch den Breitensuchalgorithmus ermittelte Baum, dafür aber im Gegenzug längere Wege. Wie bei der Tiefensuche kann unterschieden werden, ob ein spannenden Baum oder ein Wurzelbaum gebildet werden soll. Die Methoden heißen

```
GrafType span_tree_depth(GrafType const&);
GrafType root_tree_depht(unsigned int, GrafType const&);
GrafType root_tree_depht(GrafType const&);
```

Aufgabe. Implementieren Sie die Algorithmen. Vergleichen Sie die mit dem Breitensuchalgorithmus und dem Tiefensuchalgorithmus ermittelten spannenden Bäume gleicher Ausgangsgrafen miteinander.

7.3.3 Minimale (Maximale) Bäume

Kommen wir nun erstmalig auch zur Berücksichtigung der Bewertung von Kanten. In bewerteten Grafen stellt sich das Problem, einen Baum mit minimaler Kantenbewertungssumme zu erstellen. Für unser Gedankenbeispiel zu Beginn dieses Kapitel bedeutet dies, dass nicht nur alle Städte durch Autobahnen miteinander verbunden sein sollen, sondern der Baum so angelegt werden soll, dass die Fahrtstrecken möglichst kurz sind. Für diese Aufgabe ist eine grundsätzlich andere Vorgehensweise notwendig.

Der folgende Algorithmus arbeitet mit zwei Grafen, von denen einer abgebaut, der andere zum Baum aufgebaut wird:

(a) Suche die Kante mit der kleinsten Bewertung. Lösche die Kante im Quellgrafen und füge sie in den Baumgrafen ein.
(b) Prüfe, ob der Baumgraf durch das Hinzufügen der letzten Kante einen Zyklus aufweist. Falls ja, lösche die letzte Kante wieder. Sofern weitere Kanten im Quellgrafen vorhanden sind, fahre fort bei (a)

Da wir bei der Umsetzung des Algorithmus wieder mit STL-Algorithmen arbeiten wollen, definieren wir zunächst die `less`-Relation für die Kantenbewertung:

```
struct Less {
  Less(GrafType const& g):graf(g){}
  GrafType const& graf;
```

```
  bool operator()(GrafType::kante const& k1,
                  GrafType::kante const& k2){
    return graf.get_kante(k1.first,k1.second)<
           graf.get_kante(k2.first,k2.second);
  }//end function
};//end struct
```

Der Hauptalgorithmus erledigt den Rest der beschriebenen Arbeit:

```
GrafType span_tree_min(GrafType const& graf){
  GrafType res;
  GrafType::kantenliste klist;
  GrafType::kantenliste::iterator kit;
  Eckenliste elist(graf.anzahl_ecken(),0);

  klist=graf.get_kantenliste();
  while(!klist.empty()){
    kit=min_element(klist.begin(),klist.end(),
                    Less(graf));
    res.set_kante(*kit,graf.get_kante(*kit));
    if(!cycle_free(kit->first,res)){
      res.del_kante(kit->first,kit->second);
    }else{
      res.set_ecke(kit->first,
                   graf.get_ecke(kit->first));
      res.set_ecke(kit->second,
                   graf.get_ecke(kit->second));
    }//endif
    klist.erase(kit);
  }//endwhile
  return res;
}//end function
```

Der Algorithmus liefert einen Baum mit minimaler Kantenbewertungssumme, weil nach Auftreten eines Zyklus im Baumgrafen die Kante mit der höchsten Bewertung gelöscht wird. Da der Zyklus von einer Kante erzeugt wird, handelt es sich um einen einzigen Zyklus im Grafen, der durch Löschen einer einzelnen Kante wieder beseitigt werden kann. Jede andere dafür in Frage kommende Kante hat aber eine geringere Bewertung.

7.4 Wege in Grafen

Navigationsprobleme bestehen darin, in einem Graf einen Weg, also eine Liste zu durchlaufender Ecken, zu finden, die von einem gegebenen Anfangspunkt zu einem ebenfalls gegebenen Endpunkt führen. Dabei muss nicht in jedem Fall die Frage nach dem kürzesten Weg im Vordergrund stehen.

7.4.1 Beliebige Wege und Zyklen

Bei einem beliebigen Weg ist weder die Anzahl der durchlaufenen Kanten noch deren Bewertung von Bedeutung. Ein beliebiger Weg kann durch eine einfache Variation des Tiefensuchalgorithmus gefunden werden. Hierzu ist es lediglich notwendig, die notierten Kanten beim Verlassen der rekursiven Methode wieder zu löschen, sofern auf diesem Zweig die Zielecke nicht erreicht werden kann. Der Algorithmus bricht bei Erreichen der Zielecke ab, so dass im Mittel auch nicht alle möglichen Zweige des bei der Tiefensuche ermittelten Wurzelbaumes durchlaufen werden müssen.

Da keinerlei Bedingungen an den Weg gestellt werden, kann der Algorithmus auch genutzt werden, einen konkreten Zyklus im Grafen zu finden, indem die Zielecke gleich der Anfangsecke gesetzt wird.

Aufgabe. Implementieren Sie einen solchen Algorithmus.

7.4.2 Wege mit kleiner Kantenanzahl

Hier ist ein Weg zu bestimmen, der die minimale Anzahl von berührten Ecken zwischen den beiden Endecken realisiert. Diese wird durch den Breitensuchalgorithmus gegeben, der Ausgangspunkt eines entsprechenden Algorithmus ist. Durch Kombination mit einer Variante des Tiefensuchalgorithmus lässt sich das Problem sehr einfach lösen:

(a) Führe eine Breitensuche von der Ausgangsecke durch. Der Algorithmus bricht ab, sobald die Zielecke erreicht ist.
(b) Starte bei der Zielecke.
(c) Wähle eine (beliebige) eingehende Kante, deren Ausgangseckenbewertung um 1 niedriger ist als die der Bezugsecke.
(d) Trage die Ecke am Beginn des Weges (Eckenliste) ein setze die Bezugsecke auf die Ausgangsecke der Kante. Sofern die Bezugsecke noch nicht die Ausgangsecke des gesuchten Weges darstellt, fahre fort bei (c)

Aufgabe. Implementieren Sie den Algorithmus unter Berücksichtigung der Aufgabe, minimale Zyklen zu finden.

7.4.3 Minimale (Maximale) Wege

In der Praxis ist jedoch meist nicht ein Weg mit minimaler Anzahl von Transitecken zu bestimmen, sondern die Kantenbewertung ist zu berücksichtigen, d.h. ein minimaler Weg ist ähnlich wie beim minimalen Baum ein Weg mit minimaler Kantenbewertungssumme. Die Wegsuche kann durch Abbau des Grafen und Änderung der Kantenbewertung realisiert werden:

A. Markiere die Ausgangsecke als aktiv. Erzeuge einen Zielgrafen mit der Eckenanzahl des Quellgrafen, jedoch ohne Kanten. Setze alle Eckenbewertungen auf Null.
B. Suche unter allen markierten Ecken die Kante mit der geringsten Gesamtbewertung. Die gesamtbewertung ist die Summe aus Ecken- und Kantenbewertung $g_{ik} = b(e_i) + b(a_{ik})$.
   Je nach Status der Endecke der ausgewählten Kante ist eine Fallunterscheidung zu treffen:

   I. Die Endecke ist nicht aktiv. Sie wird im Algorithmus erstmalig erreicht.
      Setze die Ecke aktiv, setze die Eckenbewertung auf die Gesamtbewertung der Kante, füge die Kante in den Ergebnisgrafen ein und lösche die Kante aus dem Quellgrafen. Fahre anschließend bei B. fort.
   II. Die Endecke ist bereits aktiv, also schon auf einem anderen Weg erreicht worden. Das macht eine erneute Fallunterscheidung notwendig:

      a. Die Gesamtbewertung ist größer als die Eckenbewertung der Endecke. Der neue gefundene Weg ist ungünstiger als der alte Weg. Die Kante wird gelöscht und der Algorithmus bei B. fortgesetzt.
      b. Die Gesamtbewertung ist kleiner als die Eckenbewertung, d.h. der neue Weg ist günstiger als der bereits gefundene.
         Setze die Eckenbewertung auf die Gesamtbewertung und notiere die Differenz zu alten Bewertung.
         Verfolge im Ergebnisgrafen rekursiv die von der Ecke ausgehende Kanten (*sofern vorhanden*) und ändere im Quellgrafen die Eckenbewertung der Endecke dieses Weges um die Differenz.
         Lösche die Kante im Quellgrafen. Lösche die eingehende Kante der Endecke im Ergebnisgrafen und füge die Kante in den Ergebnisgrafen ein. Fahre fort bei B.

   III. Die Endecke ist die Zielecke. Fahre fort bei C.

C. Gehe von der Endeecke entlang der jeweils eingehenden Kanten bis zur Anfangsecke. Die dabei bestimmte Kantenliste ist der gesuchte Weg.

Der Algorithmus umfasst mehrere Teile. Zunächst ist die nächste minimale Kante unter Berücksichtigung der Vorgeschichte zu ermitteln:

```
GrafType::kante get_min_kante(GrafType const& g1,
                              GrafType const& g2){
  unsigned int minimum = INT_MAX;
  unsigned int value,test;
  GrafType::kante kant(0,0);
  GrafType::kantenliste klist;
  GrafType::kantenliste::iterator it;
  for(int i=0;i<g1.anzahl_ecken();i++){
    if((value=g1.get_ecke(i))!=-1){
```

```
      klist=g2.get_outbound_kantenliste(i);
      for(it=klist.begin();it!=klist.end();it++){
        test=value+g2.get_kante(*it);
        if(test<minimum){
          minimum=test;
          kant=*it;
        }//endif
      }//endfor
    }//endif
  }//endfor
  return kant;
}//end function
```

Der Rest wird im zweiten Teil erledigt, wobei Sie nun genau hinschauen sollten, um die oben beschriebene Vorgehensweise im Code wiederzufinden.

Aufgabe. Markieren/kommentieren Sie die einzelnen Punkte der Ablaufbeschreibung im Code (falls Sie das nicht im Buch selbst machen wollen, kopieren Sie zuvor den Code).

```
GrafType::kantenliste
    find_min_path(unsigned int estart,
                  unsigned int eziel,
                  GrafType graf){
  if(((int)graf.get_type()&(int)val_mask)==0)
    return find_short_path(estart,eziel,graf);
  int d; GrafType tmp;
  tmp.set_type(dir_uval);
  GrafType::kante kant;
  GrafType::kantenliste klist,kl;
  GrafType::kantenliste::iterator kit;
  for(int i=0;i<graf.anzahl_ecken();i++)
    tmp.set_ecke(i,-1);
  tmp.set_ecke(estart,0);

  while(true){
    kant=get_min_kante(tmp,graf);
    if(kant.first==0 && kant.second==0){
      return kl;
    }else if(tmp.get_ecke(kant.second)==-1){
      tmp.set_ecke(kant.second,
                   tmp.get_ecke(kant.first)+
                   graf.get_kante(kant));
      tmp.set_kante(kant);
      if(kant.second==eziel) break;
```

```
    }else if(tmp.get_ecke(kant.second)>
                tmp.get_ecke(kant.first)+
                graf.get_kante(kant)){
      d=tmp.get_ecke(kant.second)-
          (tmp.get_ecke(kant.first)+
           graf.get_kante(kant));
      tmp.set_ecke(kant.second,
           tmp.get_ecke(kant.first)+
                       graf.get_kante(kant));
      klist=tmp.get_inbound_kliste(kant.second);
      tmp.del_kante(*(klist.begin()));
      tmp.set_kante(kant);
      klist=tmp.get_outbound_kliste(kant.second);
      for(kit=klist.begin();kit!=klist.end();kit++)
        tmp.set_ecke(kit->second,
                     tmp.get_ecke(kit->second)-d);
    }//endif
    graf.del_kante(kant);
  }//endwhile
  while(eziel!=estart){
    klist=tmp.get_inbound_kantenliste(eziel);
    kl.push_front(*(klist.begin()));
    eziel=klist.begin()->first;
  }//endwhile
  return kl;
}//end function
```

Anmerkung. Technisch wird der zu untersuchende Graf nicht als Referenz übergeben, sondern als Kopie, da der beschriebene Algorithmus mit Kantenlöschoperationen arbeitet. Der intern aufgebaute Zielgraf wird mit dem Typ **gerichtet-unbewertet** definiert, um eindeutige Kantenrichtungen für die in Abschnitt A.II.b. beschriebenen Operationen zu erhalten.
Gewissermaßen spiegelsymmetrisch dazu können Sie auch einen Algorithmus schreiben, der einen Weg mit maximaler Kantenbewertung sucht.

7.4.4 Rundwege in Grafen

Bei Rundwegen in Grafen wird jede Ecke unter Umständen mehrfach besucht, aber jede Kante nur einmal durchlaufen. Das Rundwegproblem wird auch als Briefträgerproblem bezeichnet, der auf seine Tour möglichst nicht mehr Straßen (Kanten) benutzen muss, die er schon beliefert hat.

Voraussetzung für Rundwege sind ***gleiche*** Anzahlen von ***ausgehenden und eingehenden*** Kanten an jeder Ecke mit Ausnahme von eventuell genau zwei Ecken. Der Grund ist klar: außer an der Start- und der Endecke muss jede erreichte Ecke

auch wieder verlassen werden können. Die Startecke muss eine ausgehende Kante mehr aufweisen als eingehende, die Endecke eine eingehende Kante. Sind diese Nebenbedingungen nicht erfüllt, ist das Problem nicht lösbar. Im Startteil des Algorithmus sind zunächst diese Bedingungen zu prüfen:

A. Besitzen alle Ecken gerade gleiche Anzahlen von ausgehenden und eingehenden Kanten, wähle eine beliebige Ecke als Anfangsecke für den Algorithmus.
B. Sind genau jeweils eine Start- und eine Endecke vorhanden, suche einen beliebigen Weg von der Start- zur Endecke, lösche ihn aus dem Grafen und trage ihn in den Ergebnisweg ein. Die Startecke ist Anfangsecke des Algorithmus.
C. Sind Start oder Endecke vorhanden, aber nicht jeweils genau eine, so breche ab.

Der Algorithmus benötigt zunächst eine Reihe von Arbeitsparametern.

```
GrafType::kantenliste round_trip(GrafType graf){
  int e1=-1,e2=-1,i;
  GrafType::kantenliste klist,rlist;
  GrafType::kantenliste::iterator kit,kit2;
  GrafType::gradliste ekl;
  GrafType::gradliste::iterator ekit;

  if(!connect_weak(graf)) return klist;
  ekl=graf.grad_ecken();
```

Der folgende Teil für die Grundprüfung für gerichtete

```
if((int)graf.get_type()&dir_mask !=0){
  for(ekit=ekl.begin(),i=0;ekit!=ekl.end();
      ekit++,i++){
    if(abs((int)ekit->first-(int)ekit->second)>1)
      return rlist;
    if(ekit->first-ekit->second==1){
      if(e1!=-1) return rlist;
      e1=i;
    }//endif
    if(ekit->second-ekit->first==1){
      if(e2!=-1) return rlist;
        e2=i;
    }//endif
  }//endfor
```

und ungerichtete Grafen durch

```
}else{
  for(ekit=ekl.begin(),i=0;ekit!=ekl.end();
      ekit++,i++){
    if(e1!=-1 && ekit->first%2!=0){
```

```
      if(e2!=-1) return rlist;
      e2=i;
    }//endif
    if(e1==-1 && ekit->first%2!=0) e1=i;
  }//endfor
  if(e1==-1 ^ e2==-1) return rlist;
}//endif
```

Der restliche Algorithmus sammelt nun iterativ Zyklen im Restgrafen. Die Idee beruht darauf, einen Zyklus zu suchen und den Gesamtweg entlang dieses Weges zu konstruieren, indem in jedem Punkt zunächst rekursiv ein weiterer Zyklus durchlaufen und erst dann der Weg fortgesetzt wird. Da jeder Zyklus aus dem Grafen entfernt wird, kann jede Kante nur einmal durchlaufen werden. Die gleiche Anzahl von Ein- und Ausgangsgraden stellt sicher, dass alle Kanten verbraucht werden können. Begonnen wird an der Startecke des Ergebnisweges, der in Fall A. leer ist, in Fall B. jedoch schon den Weg zwischen Start- und Endecke enthält.

1. Suche eine Zyklus(weg) im Restgrafen mit der aktuellen Ecke als Startecke. Unterscheide folgende Fälle:
   (a) Die aktuelle Ecke besitzt keine weiteren Kanten: nehme die Endecke der von der aktuellen Ecke ausgehenden Kante im Ergebnisweg als neue aktuelle Kante und setze bei A. fort.
   (b) Ein Zyklus wurde gefunden: füge den gefundenen Weg ***vor*** der von der aktuellen Ecke ausgehenden Kante im Ergebnisweg ein. Lösche den Zyklus aus dem Grafen. Fahre fort bei A.
   (c) Es wurde kein Zyklus gefunden: breche mit einem Fehler ab.
2. Der Algorithmus endet, wenn das Ende des Ergebnisweges erreicht ist und die Ecke im Ausgangsgrafen keine weiteren Kanten mehr aufweist. Der Ausgangsgraf sollte am Ende kantenlos sein.

Der Implementationsteil ist etwas übersichtlicher als die Prüfung, ob der Graf die Anforderungen erfüllt.

```
if(e1!=-1 && e2!=-1)
  rlist=find_short_path(e1,e2,graf);
else
  rlist=find_any_path(0,0,graf);
for(kit=rlist.begin();kit!=rlist.end();kit++)
  graf.del_kante(*kit);

e1=rlist.begin()->first; i=0;
while(i<rlist.size()){
  klist=find_any_path(e1,e1,graf);
  kit=rlist.begin();
  advance(kit,i);
```

```
    if(!klist.empty()){
      for(kit2=klist.begin();
          kit2!=klist.end();kit2++)
        graf.del_kante(*kit2);
      rlist.insert(kit,klist.begin(),klist.end());
    }else{
      e1=kit->second;  i++;
    }//endif
  }//endfor
  return rlist;
}//end function
```

Der Algorithmus bricht ab, wenn das Ende des Ergebnisweges erreicht ist. Dabei können alle Ecken mehrfach als aktive Ecken an unterschiedlichen Punkten des Weges auftreten. Da sie nach dem ersten Bearbeiten jedoch über keine weiteren Kanten verfügen, entsteht keine weitere Suchzeit nach Zyklen.

Aufgabe. Liefert der Algorithmus immer einen Rundweg ab? Wenn Kantenanzahlen von Ausgangs- und Endgrafen übereinstimmen, ist dies sicher der Fall. Falls die Kantenanzahl im Endgrafen kleiner wäre, so wäre ein zufälliger Weg ohne Kantenwiederholung bestimmt, der jedoch nicht notwendigerweise der längste mögliche Weg sein muss. Lässt sich so ein Fall konstruieren?

7.4.5 Rundreise durch die Ecken

Eine Rundreise unterscheidet sich von einem Rundweg dadurch, dass alle Ecken höchstens einmal besucht werden sollen. Dieses Problem ist auch als Handlungsreisendenproblem bekannt, der in verschiedenen Städten Geschäfte abwickeln, dabei aber jede Stadt nur einmal besuchen soll.
Die Suche kann auf zwei Arten durchgeführt werden:

(a) Sämtliche Ecken besitzen eingehende und ausgehende Kanten. Die Suche kann an einer beliebigen Ecke begonnen und als Suche nach dem längsten Zyklus durchgeführt werden.
(b) Maximal eine Ecke besitzt nur ausgehende Kanten, maximal eine Ecke besitzt nur eingehende Kanten. Damit liegen Start- oder Endecke fest, und es wird nach dem Weg gesucht.

Der folgende Algorithmus findet zumindest den längsten möglichen Weg:

A. Bestimme eine Start- und eine Endecke nach dem angegebenen Schema. Rufe die Rekursionsfunktion mit der Startecke und einem leeren Weg auf.
B. Markiere die Ecke als besucht.

I. Besitzt die Ecke keine ausgehenden Kanten, notiere den gefundenen Weg als Ergebnisweg, sofern er länger ist als der zuvor gefundene Ergebnisweg.
II. Sonst werte nacheinander sämtliche ausgehende Kanten in der folgende Weise aus.

   Füge die Kante in den aktuellen Weg ein. Untersuche die Endecke der Kante.

   a. Wenn die Endecke die Zielecke ist, notiere den gefundenen Weg als Ergebnisweg, sofern er länger ist als der zuvor gefundene Ergebnisweg. Enthält der Weg sämtliche Ecken, ist der Algorithmus fertig.
   b. Wenn die Endecke der Kante als besucht markiert ist, übergehe die Kante.
   c. Wenn die Endecke der Kante nicht als besucht markiert ist, rufe die Rekursionsfunktion mit der Endecke und dem aktuellen Weg auf.

   Lösche die Kante aus dem aktuellen Weg.

C. Sind alle Kanten untersucht, markiere die Ecke als nicht besucht und verlasse die Rekursionsfunktion.

Der Algorithmus findet für eine gegebene Startecke zumindest den längsten möglichen Weg im Grafen. Sofern der Weg nicht alle Ecken umfasst, muss er aber weder der längste tatsächlich mögliche sein noch die Zielecke enthalten. Sind Start- und Zielecke nicht nach b) festgelegt, kann versucht werden, das Ergebnis durch Auswahl einer anderen Startecke (beispielsweise einer bislang nicht erreichten) zu verbessern.

Aufgabe. Versuchen Sie sich an einer Implementation.

Das Handlungsreisendenproblem gehört zu den härteren Problemen der Grafentheorie, insbesondere, wenn noch die Nebenbedingung erfüllt werden soll, dass nicht nur ein Weg, sondern der kürzeste Weg gefunden werden soll. Da wir hier aber nicht einem Lehrbuch über Grafentheorie Konkurrenz machen wollen, vertiefen wir dieses Thema nicht weiter.

7.5 Netzwerke

Ist eine Wegbestimmung in einem Grafen kein einmaliger Vorgang, sondern konkurrieren mehrere Nutzer von Wegen miteinander, so entsteht ein Netzwerkproblem. So führt die optimale Lösung eines Navigationsproblems zwangsweise zu einem Stau, wenn zu viele Verkehrsteilnehmer gleichzeitig diese Route verwenden wollen, und eine weniger guter Route wäre unter Umständen trotz allem günstiger gewesen. Neben der Länge eines Weges spielt dann auch dessen Kapazität eine Rolle, d.h. wir kommen zu Grafen, die Eckenbewertungen (die Anzahl der Nutzer) und eine bis mehrere Kantenbewertungen (Kapazität, Länge) enthalten.

7.5.1 Flüsse in Netzwerken

Netzwerke zum Transport irgendwelcher Größen (Datenverkehr, Güterverkehr, usw) enthalten Quellen und Senken für dieses Größen, was durch die Eckenbewertung angezeigt wird. Die Kantenbewertung gibt die Transportkapazität der Kanten an. Typische Fragestellungen in solchen Netzwerken sind:

- Welche Menge kann maximal von Quelle A zu Senke B transportiert werden?
- Welche Reserven sind noch vorhanden (kann die Quellen- oder Senkenkapazität erhöht werden, können zusätzliche Quellen/Senken in das System implementiert werden?
- Wie sind die Transportkapazitäten zu erweitern, um bestimmte Gesamtflüsse zu erreichen.

Ein Algorithmus zur Bestimmung des Flusses ist mit Hilfe des Wegalgorithmus für beliebige Wege implementierbar und fällt erstaunlich kurz aus:

A. Suche den maximalen Weg von einer Quelle zu einer Senke. Ist kein Weg mehr zu finden, gebe den Gesamtfluss aus und beende den Algorithmus.
B. Bestimme die Kante mit der geringsten Kapazität auf diesem Weg. Vergleiche die Kapazität mit den Eckenbewertungen von Quelle und Senke. Unterscheide:
   I. Die Kantenbewertung ist kleiner als die (Absolutwerte) der Eckenbewertungen. Subtrahiere die Kantenbewertung von den Eckenbewertungen (bei der Senke addieren) und allen Kanten des Weges. Addiere die Bewertung zum Gesamtfluss. Fahre fort bei A.
   II. Die Kantenbewertung ist größer als das Minimum der (Absolutwerte) der Eckenbewertungen. Subtrahiere das Minimum von den Eckenbewertungen und den Kantenbewertungen des Weges, addiere es zum Gesamtfluss und beende den Algorithmus.

Der Algorithmus passt bei jedem Durchlauf die Kapazitäten der Kanten gemäß der auf dem gefundenen Weg maximal transportierbaren Menge an und entfernt „erschöpfte“ Kanten. Weiter Wege, die gefunden werden, sind mithin andere Wege. In der Implementation greifen wir auf die entsprechenden Vorarbeiten zurück.

```
unsigned int flux_with_reduction(unsigned int e1,
                                 unsigned int e2,
                                 GrafType& graf){
  int fluss=0; int red;
  GrafType::kantenliste klist;
  GrafType::kantenliste::iterator kit;

  while(graf.get_ecke(e1)>0&&graf.get_ecke(e2)<0){
    klist=find_max_path(e1,e2,graf);
    if(klist.empty()) break;
```

```
    kit=min_element(klist.begin(),klist.end(),
                    Less(graf));
    red=min(graf.get_ecke(e1),
     min(-graf.get_ecke(e2),graf.get_kante(*kit)));
    graf.set_ecke(e1,graf.get_ecke(e1)-red);
    graf.set_ecke(e2,graf.get_ecke(e2)+red);
    for(kit=klist.begin();kit!=klist.end();kit++)
      graf.set_kante(*kit,
                     graf.get_kante(*kit)-red);
      fluss+=red;
  }//endwhile
  return fluss;
}//end function
```

Sind mehrere Quellen oder Senken vorhanden, kann nach erschöpfender Untersuchung eines Paares (es lässt sich kein weiterer Weg mehr finden oder die Kapazität einer Ecke ist erschöpft) ein weiteres Paar untersucht werden, bis alle Kombinationen von Ecken erschöpft sind. Der Algorithmus gibt so Auskunft über mögliche Flüsse zwischen jedem Quellen/Senkenpaar.

Das Ergebnis kann abhängig sein von der Reihenfolge der Paarungen: versorgt eine Quelle zwei Senken, besitzt jedoch nicht die Kapazität, beide Senken zu füllen, so wird bei einem Wechsel der Reihenfolge der Senken auch die voll bediente Senke ausgetauscht (entsprechende Kantenkapazitäten vorausgesetzt).

Aufgabe. Anstelle des tatsächlichen Flusses, den der beschriebene Algorithmus liefert, kann auch der theoretisch maximal mögliche Fluss zwischen zwei Ecken bestimmt werden. Entwerfen Sie eine Modifikation, die dies macht.

7.5.2 Flüsse mit Nebenbedingungen

Bei Transportproblemen besteht häufig die Nebenbedingung, nicht nur eine bestimmte Menge von der Quelle zur Senke zu transportieren, sondern dazu auch die günstigsten Wege zu verwenden. Da beim ersten Algorithmus die Wegeauswahl zufällig erfolgt, kann es passieren, dass ein ungünstiger Weg vollständig ausgelastet ist, während ein günstiger Weg noch Kapazität frei hat.

Zur Lösung dieses Problem sind im Rahmen dieser Implementation zwei Grafen mit identischer Eckenbelegung zu konstruieren, von denen einer die Kapazitätsbewertung, der andere die Wegbewertung enthält. Der Algorithmus erhält folgende Modifikation:

A. Suche den kürzesten Weg im zweiten Grafen
B. Fahre fort, wie im ersten Algorithmus beschrieben, lösche „verbrauchte“ Kanten auch im zweiten Grafen

```
unsigned int flux_red_cond(unsigned int e1,
                           unsigned int e2,
                           GrafType& gf,
                           GrafType& gw,
                           Wegliste& wl){
  unsigned int fluss=0; unsigned int red;
  GrafType::kantenliste klist;
  GrafType::kantenliste::iterator kit;
  if(gf.get_kantenliste()!=gw.get_kantenliste())
    return 0;

  while(gf.get_ecke(e1)!=0 && gf.get_ecke(e2)!=0){
    klist=find_min_path(e1,e2,gw);
    if(klist.empty()) break;
    kit=min_element(klist.begin(),klist.end(),
                    Less(gf));
    red=min((int)gf.get_ecke(e1),
     min((int)gf.get_ecke(e2),gf.get_kante(*kit)));
    gf.set_ecke(e1,gf.get_ecke(e1)-red);
    gf.set_ecke(e2,gf.get_ecke(e2)-red);
    for(kit=klist.begin();kit!=klist.end();kit++){
      gf.set_kante(*kit,gf.get_kante(*kit)-red);
      if(gf.get_kante(*kit)==0) gw.del_kante(*kit);
    }//endfor
    fluss+=red;
    wl.push_back(Weg(red,klist));
  }//endwhile
  return fluss;
}//end function
```

Die Algorithmen liefern auch die ermittelten Wege und deren Kapazitäten. Als Nebenbedingung wird überprüft, ob die beiden übergebenen Grafen identische Kantenlisten aufweisen.

7.5.3 Belegungsprobleme

Ein anderes Problem ist das Buchbinderproblem, das jedoch ähnlich hart ist wie das Handlungsreisendenproblem und daher nur der Vollständigkeit halber genannt werden soll. Es besteht darin, dass ein Produkt (ein Buch) mehrere Stationen eines Produktionsprozesses (Drucken, Schneiden, Binden) durchlaufen muss und für jede Station mehrere Maschinen unterschiedlicher Kapazität zur Verfügung stehen. Liegen nun mehrere verschiedene Aufträge vor, so kann für jede Station die Zeit

berechnet werden, die für die Durchführung benötigt wird.[8] Wie sind die verschiedenen Aufträge auf die einzelnen Maschinen zu verteilen, damit die Auslastung der Maschinen optimal ist?

Aufgabe. Eine Strategie besteht in der Suche nach einem Weg, auf dem möglichst keine Zwischenlager entstehen. Das System startet mit einem Auftrag, der nach der Zeit a [1] die erste und $a_1 + b_1$die zweite Maschine verlässt. Der zweite Auftrag steht nach der Zeit $a_1 + a_2$ an der zweiten Maschinen zur Verfügung, die dann möglichst schon fertig sein sollte, also $a_1 + a_2 \geq a_1 + b_1$. Der zweite Auftrag ist in diesem Fall nach $a_1 + a_2 + b_2$ Zeiteinheiten abgeschlossen, im anderen erst nach $a_1 + b_1 + b_2$. Versuchen Sie sich an einer Formulierung eines solchen Algorithmus und an einer Implementation.

[8] Dabei besteht in der Regel kein linearer Zusammenhang zwischen Auftragsgröße und Maschinenkapazität, denn neben der reinen Auftragsabwicklung sind auch Einrichtungszeiten usw. zu berücksichtigen. Dadurch gewinnen solche Probleme u.U. an zusätzlicher Dynamik.