Graphenalgorithmen: Blatt 4

Lukas Kalbertodt, Elena Resch, Mirko Wagner

14. Mai 2015

A8	A9	Σ	P2
10/10	5/6	15/16	10/10

Aufgabe 8:

(a) Angenommen $G = (V, E)$ sei ein ungerichteter Graph ohne Kreise, also ein Baum (oder Wald). Ein Baum ist immer automatisch bipartit, da man einfach alle Knoten mit geradem Abstand von der Wurzel in eine Partition einordnet und die mit ungeradem Abstand in die andere Partition (die Wahl der Wurzel ist hierbei irrelevant). Im Falle eines Waldes funktioniert das analog, indem man jeden Baum wie oben beschrieben einordnet.
„⇒“: Ein ungerichteter Graph $G = (V, E)$ ist bipartit. Daraus folgt, dass Kanten nur zwischen Knoten der Mengen V_1 und V_2 liegen. Sei ein Kreis der Länge k in G vorhanden und sei V_1 die Knotenmenge mit dem Startknoten, so braucht man 2 Kanten, um wieder in V_1 zu landen. Für $k = 2$ ist der Kantenzug in G offen. Deswegen brauchen mindestens 2 weitere Kanten, um den minimalen Kantenzug zu schließen. Ist der Kantenzug noch nicht geschlossen, so gehören 2 weitere Kanten zum Kreis, um wieder in der Ursprungsmenge V_1 zu landen. Es wird solange wiederholt, bis der Startknoten erreicht wurde. Insgesamt also braucht man k Kanten, $k \geq 4, k$ gerade, um einen geschlossenen Kantenzug zu erzeugen.
„⇐“: Sei $G = (V, E)$ ein ungerichteter Graph. Seien in G Kreise gerader Länge vorhanden. Für jeden Kreis in G partitioniert man die Knoten, indem man sie dem Kreis entlang abwechselnd den unterschiedlichen Partitionen zuordnet. Bei geraden Kreisen verletzen wir so das bipartit-Kriterium nicht, weil nie zwei im Kreis benachbarte Knoten in der selben Partition sind. Bei ungeraden Kreisen würden zwei benachbarte Knoten in der selben Partition sein und es würde somit eine Kante zwischen zwei Knoten einer Partition existieren.

(b) Beweis durch Widerspruch:
Sei $G = (V, E)$ ein gerichteter azyklischer Graph. Angenommen G besitzt zwei verschiedene transitive Reduktionen R_1 und R_2. Betrachte eine Kante $k = (e, f) \in R_1, k \notin R_2$.
z.z. Es existiert ein Weg von e nach f in R_2, der über einen dritten Knoten v führt.
Bew: R_2 ist eine transitive Reduktion und hat den gleichen transitiven Abschluss wie R_1. Da $k \notin R_2$ ist, existiert mindestens ein weiterer Knoten v, sodass gilt: Der Weg von e nach f führt über v.
z.z. Es existiert in R_1 ein Weg von e nach v und ein Weg von v nach f, die beide

nicht die Kante k enthalten.
Bew: Da R_1 den gleichen transitiven Abschluss hat wie R_2, existiert ein Weg von e nach v und ein Weg von v nach f. Wenn k in dem Weg von e nach v oder in dem Weg von v nach f enthalten wäre, dann enthielte R_1 nicht die minimale Kantenzahl. Daraus folgt, dass $k \notin R_1$. Widerspruch zur Voraussetzung, dass $k \in R_1$.
$\Rightarrow R_1 = R_2$.

(c) $\lambda(G) \leq \delta(G)$: Man kann immer einfach alle Kanten des Knoten mit dem niedrigsten Grad entfernen, um diesen Knoten zu isolieren und den Graph in zwei Komponenten zu teilen.
$\kappa(G) \leq \lambda(G)$: Wenn wir k Kanten entfernen können und den Graph so in zwei Komponenten teilen, können wir auch einfach einen der beiden Knoten an der Kante löschen. So wird die Kante implizit mit gelöscht und wir haben den selben Effekt. Also müssen wir maximal so viele Knoten wie Kanten entfernen, oft aber weniger, weil Knoten mehrere Kanten besitzen. Bei dieser Argumentation schließen wir vollständige Graphen aus, da dort $\kappa(G) = \lambda(G) = \delta(G) = |V| - 1$. Dies bedeutet, dass man $|V|-1$ Knoten löschen müsste und somit nur noch einen Knoten überhat (sprich: Es ist unmöglich einen vollständigen Graphen durch Löschen von Knoten in zwei Komponenten aufzuteilen).

Aufgabe 9:

(a) mithilfe des Lemmas 2.1 (Skript, S.33):

```
proc dfs(i)
  push(S, i);
  repeat
    i := pop(S);
    k := k + 1;
    nr[i] := k;
    for all Nachbarn j von i do
      if nr[j] = 0 then
        push(S, j);
        nr[j]:= -1;
      else if nr[j] = nr[i] + 1 then
        print (i, j) "Baumkante"
      else if nr[j] > nr[i] + 1 then
        print (i, j) "Vorwärtskante"
      else if nr[j] < nr[i] then
        if forfahre(j, i) = true
          print (i, j) "Rückwärtskante"
        else
          print (i, j) "Querkante"
  until isEmpty(S);
```

```
//weitere Tiefensuche, ob ein Kreis entsteht,
//d.h. ob i als Nachfolger von j auftaucht
//oder nicht
proc vorfahre(j,i)
  push(S', j);
  repeat
    tmp := pop(S');
    markiere tmp als besucht
    for all Nachbarn next von tmp do
      if next noch nicht besucht then
        if next = i then
          return true;
        else
          push(S', next);
  until isEmtpy(S');
  return false;
```

Kann schiefgehen, wenn ein Knoten mehrere Nachfolger im DFS-Baum hat

In zusammenhängendem, ungerichteten Graphen wird das immer true liefern

(b) Sei (i, j) eine beliebige Kante in G. O.B.d.A. sei $\text{nr}[i] < \text{nr}[j]$. Dann wurde der

Knoten i vor Knoten j gefunden und abgearbeitet worden sein. Falls die Durchmusterung die Kante zuerst in der Richtung von i nach j sondiert, dann ist j bis dahin noch nicht betrachtet worden. Somit wird (i, j) eine Baumkante. Falls die Durchmusterung die Kante (i, j) zuerst in Richtung von j nach i sondiert, dann ist (i, j) eine Rückwärtskante, da i zu dem Zeitpunkt, zu dem die Kante erstmals sondiert wird, schon einmal betrachtet wurde.
Im Breitensuchbaum gilt für eine Kante (i, j) mit level$[i]$ = level$[j]$ oder level$[i]$ = level$[j] + 1$, dass es sich um eine Querkante handelt, wobei es sich für (i, j) mit level$[i]$ = level$[j] + 1$ auch um eine Baumkante handeln kann.

Programmieraufgabe P2:

(c) Nicht alle Startkonfigurationen (von insgesamt 9! Möglichkeiten) haben eine Lösung, den z.B.

1	2	3
4	5	6
8	7	

ist durch ein Vertauschen der Zahlen 7 und 8 entstanden und nicht durch eine Reihe von Verschiebungen von der Spielfeldzielkonfiguration:

1	2	3
4	5	6
7	8	

10/10