Algorithmen und Datenstrukturen 1

Ausgearbeitetes Übungsblatt 4

Datum: 21. Mai 2005
Erstellt mit LaTeX

Aufgabe 4.1

Aufgabenstellung:

Entwerfen Sie einen Algorithmus in Pseudocode, der feststellt, ob ein gegebener gerichteter Graph stark zusammenhängend ist. Ein gerichteter Graph ist genau dann stark zusammenhängend, wenn jeder einzelne Knoten des Graphen von jedem anderen Knoten aus über einen gerichteten Pfad erreichbar ist.

Lösung:

Der hier vorgestellte Algorithmus erzeugt eine Matrix, ähnlich einer Adjazenzmatrix, zu dem Graphen $G =< V, E >$; in dieser Matrix soll allerdings nicht gespeichert werden, von welchem Knoten zu welchem Knoten es eine Kante gibt, sondern, von welchem Knoten zu welchem Knoten ein Pfad führt. Dies soll durch eine 1 ausgedrückt werden, eine 0 soll ausdrücken, dass entweder der Knoten der entsprechenden Zeile noch nicht behandelt wurde, der Knoten mit sich selbst in Beziehung steht oder von einem Knoten zu einem anderen Knoten kein Pfad vorhanden ist. Das Array wird bezeichnet mit *erreichbar* wobei in der Bezeichnung erreichbar[a,b] a für den Knoten steht, von dem Pfade gesucht werden, und b für den Knoten, zu dem ein Pfad führen soll. Das Array sei zunächst als Nullmatrix (alle Elemente sind gleich 0) initialisiert.
Der Einfachheit halber soll jeder Knoten eine eindeutige Identifikationsnummer i mit $1 \leq i \leq |V|$ aufweisen, welche mit Hilfe der Methode **num**(v) abrufbar ist.
Die Ermittlung der vorhandenen Pfade wird wie folgt durchgeführt:

1. Wähle den Knoten $v_i \in V$ mit der kleinsten Identifikationsnummer, der noch nicht behandelt wurde. Suche nun mit Hilfe der Tiefensuche Knoten w, welche von v_i aus erreichbar sind. Markiere alle diese Knoten mit einer 1 im Array *erreichbar* an der Position erreichbar[num(v_i),num(w)]. Existiert allerdings ein $j < i$, sodass $v_j = w$, so markiere alle Knoten als von v_i erreichbar, da sie ja auch von w erreichbar sind.

2. Bleibt schließlich ein Knoten übrig, den ich von einem bestimmten Knoten nicht erreichen kann, so kann ich das gesamte Verfahren abbrechen, der Graph kann nicht mehr stark zusammenhängend sein. Ansonsten wiederhole Schritt 1 mit dem nächsten Knoten, bis alle Knoten behandelt worden sind.

```
istZusammenhaengend(G) {
  for each(v in V) {
    if(!DFS(G, v, v)) return false;
  }

  return true;
}

// 2. Parameter (u): Knoten, von dem aus gerade Pfade gesucht werden
// 3. Parameter (v): Knoten, bei dem wir im Rahmen der Tiefensuche gerade "angekommen" sind.

DFS(G, u, v) {
  for each(w in N-(v)) {
    if(num(w) < num(u)) {
      // Knoten gefunden, der bereits behandelt wurde
      // daher alle Kanten als erreichbar kennzeichnen
      for(a=1; a<=|v|; a++) {
        erreichbar[num(w)][a] =1 ;
      }

      // Von u sind sicher alle Knoten erreichbar
      return true;
    }

    if(w != u && erreichbar[num(u)][num(v)] == 0) {
      erreichbar[num(u)][num(v)]=1;
```

```
      DFS(G, u, w);
    }
  }

  for(a=1; a<=|v|; a++) {
    if(erreichbar[num(w)][a]==0 && num(w)!=a) return false;
  }

  return true;
}
```

Aufgabe 4.2

Aufgabenstellung:

Geben Sie den Pseudocode eines Algorithmus an, der feststellt, ob ein gegebener Graph mit zwei Farben knotenfärbbar ist. Ein Graph ist mit zwei Farben knotenfärbbar, wenn jedem Knoten eine Farbe so zugeordnet werden kann, dass Knoten, die durch eine Kante verbunden sind, jeweils verschiedene Farben haben.

Lösung:

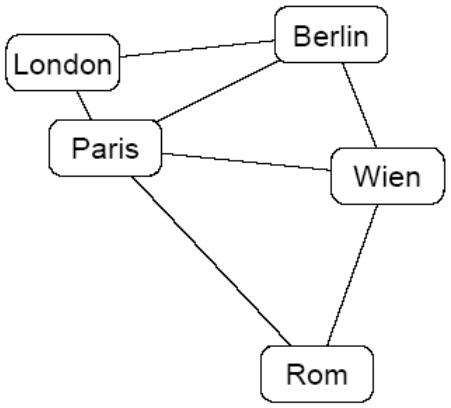
Der vorliegende Algorithmus durchmustert den Graphen mit Hilfe von Tiefensuchen und speichert für die einzelnen Knoten abwechselnd die Werte 1 und -1 (für die beiden Farben) in einem globalen Array *markiert*[], welches zu Beginn der Ausführung mit lauter Nullen gefüllt sein soll. So lange bei dieser Speicherung von 1 und -1 keine Probleme auftreten, ist der Graph mit zwei Farben knotenfärbbar. Tritt jedoch der Fall ein, dass für einen Knoten einmal 1 und einmal -1 gespeichert werden soll, so ist der Graph nicht mit zwei Farben knotenfärbbar.

```
istFaerbbar(G) {
  for each(v in V) {
    if(markiert[v] == 0) {
      if(!DFS(G, v, 1)) return false;
    }
  }

  return true;
}

DFS(G, v, farbe) {
  markiert[v] = farbe;
  for each(w in N(v)) {
    if(markiert[w] == 0)
      if(!DFS(G, w, -farbe)) return false;
    else if(markiert[w] != -farbe)
      return false;
  }

  return true;
}
```

Aufgabe 4.3

Aufgabenstellung:

Der folgende ungerichtete, gewichtete Graph stellt das Flugnetz einer kleinen Fluglinie dar. Die Knoten sind durchnummeriert. Jedem Knoten ist zusätzlich zur Nummer auch eine Bezeichnung zugeordnet (siehe Tabelle).
Die Kantengewichte entsprechen der Luftlinienentfernung zwischen den einzelnen Flughäfen. Für alle Kantenbewertungen $w_{i,j}$ gilt, dass $w_{i,j} \geq 0$.



Knoten	Bezeichnung
1	London
2	Berlin
3	Wien
4	Rom
5	Paris

(a) Schreiben Sie einen Algorithmus in Pseudocode, der ausgehend vom Knoten mit der Nummer 1 mittels Tiefensuche den Graphen durchläuft und dessen Adjazenzmatrix ausgibt. Dabei soll in der Matrix an der Position $A[i, j]$ das Kantengewicht der Kante zwischen dem Knoten i und j gespeichert werden, falls eine solche existiert. Alle anderen Werte sollen 0 sein.

(b) Geben Sie den Aufwand Ihres Algorithmus in O-Notation an.

(c) Verdeutlichen Sie die Vorgehensweise Ihres Algorithmus, indem Sie für den gegebenen Graphen die Adjazenzmatrix erstellen. Geben Sie dabei die Reihenfolge an, in der die Matrixeinträge erfolgen.

Lösung:

(a)

```
AdjazenzMatrix(G) {
  // Initialisierung
  for(a=1; a<=5; a++) {
    for(b=1; b<=5; v++) {
      A[a, b] = -1;
    }
  }

  // Tiefensuche beginnt bei Knoten 1
  DFS(G, 1);

  // Für Kanten, die nicht gefunden wurden, Gewicht 0 statt -1 eintragen
  for(a=1; a<=5; a++) {
    for(b=1; b<=5; v++) {
      if(A[a, b]==-1) {
        A[a, b] = 0;
      }
    }
  }

  return A;
}

DFS(G, v) {
  for each(w in N(v)) {
    if(A[v, w]==-1) {
      A[v, w]=w(v,w);
      A[w, v]=w(w,v);
      DFS(G, w);
    }
  }
}
```

In diesem Algorithmus liefert $w(a, b)$ das Kantengewicht der Kante von a nach b.

(b) Die Laufzeit der Initialisierung liegt in $\Theta(|V|^2)$, da sie für jeden Knoten einmal ausgeführt wird. Die Kosten eines eines DFS-Aufrufs für einen Knoten v ohne Berücksichtigung rekursiver DFS-Aufrufe entsprechen im Worst-Case dem Grad $d(v)$ dieses Knotens, liegen also in $O(d(v))$. Der DFS-Algorithmus wird für jeden Knoten einmal

aufgerufen. Schließlich liegt die Ersetzung von -1 durch 0 wieder in $\Theta(|V|^2)$. Es ergibt sich daher insgesamt für die Kosten dieses Algorithmus: $\Theta(|V|^2) + \sum_{v \in V)} O(d(v)) + \Theta(|V|^2) = O(|V|^2) + O(|E|) = O(|V|^2 + |E|)$.

(c)

Aufruf DFS(1): Eintragung von A[1,2], A[2,1]
Aufruf DFS(2): Eintragung von A[2,3], A[3,2]
Aufruf DFS(3): Eintragung von A[3,4], A[4,3]
Aufruf DFS(4): Eintragung von A[4,5], A[5,4]
Aufruf DFS(5): Eintragung von A[5,1], A[1,5], A[5,2], A[2,5], A[5,3], A[3,5]

Aufgabe 4.4

Aufgabenstellung:

Der sogenannte *transponierte Graph* eines gerichteten Graphen $G = (V, E)$ ist der Graph $G^T = (V, E^T)$, wobei

$$E^T = \{(v, u) \in V \times V | (u, v) \in E\}$$

d.h. gegenüber dem ursprünglichen Graphen kehrt sich die Richtung aller Kanten um.
Geben Sie einen effizienten Algorithmus an, der G^T ausgehend von G berechnet, wenn G in Adjazenzlistendarstellung vorliegt. Analysieren Sie die Laufzeit Ihres Algorithmus.

Lösung:

Diese Lösung geht davon aus, dass ein Array von Objekten vorliegt, welches den Graphen G in Adjazenzlistendarstellung speichert. Jedes Element dieses Arrays speichert die Knoten, zu denen von einem bestimmten Knoten Kanten vorhanden sind. Die Objekte, welche zur Speicherung einer solchen Liste dienen, seien einfach verkettete Listen, welche folgende Methoden aufweisen:

- **get**() ruft beim ersten Aufruf das erste Element der Liste ab, bei jedem weiteren Aufruf das jeweils nächste Element.
- **hasNext**() ist logisch wahr, wenn das zuletzt mit get() abgerufene Element nicht das letzte Element in der Liste ist oder get() noch nicht aufgerufen wurde.
- **add**(el) fügt den Knoten *el* an das Ende der Liste an.

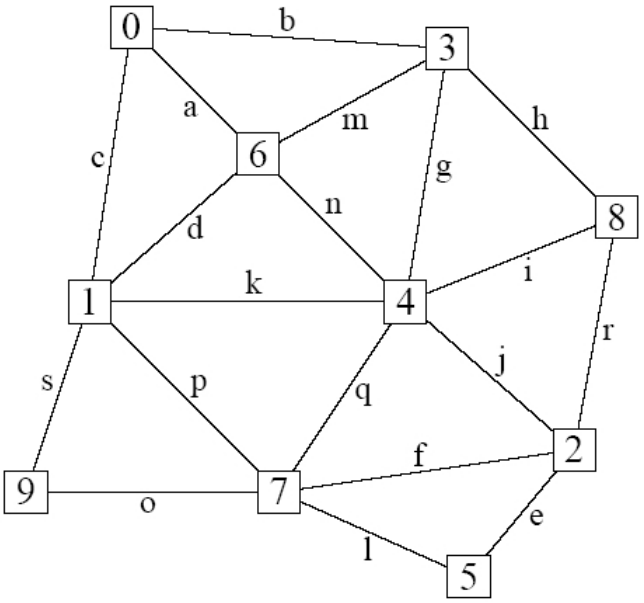
```
for each(n in V) {
  save = alt[n].get();
  while(alt[n].hasNext()) {
    temp = alt[n].get();
    neu[temp].add(save);
  }
}
```

Die äußere Schleife wird für jeden Knoten einmal ausgeführt. Die innere Schleife wird für jede Kante eines bestimmten Knotens ausgeführt, die von diesem Knoten wegführt. Insgesamt hängt die Anzahl der Schleifendurchgänge also von der Anzahl aller Kanten ab; die Laufzeit dieses Algorithmus liegt also in $\Theta(|E|)$.

Aufgabe 4.5

Aufgabenstellung:

Führen Sie anhand des nachstehend abgebildeten Graphen und der in der Tabelle gegebenen Kantenkosten den Algorithmus von Kruskal für das Finden eines minimalen Spannbaums durch. Geben Sie immer, wenn der Algorithmus eine Kante zu dem Baum hinzunimmt, den Buchstaben der Kante an.

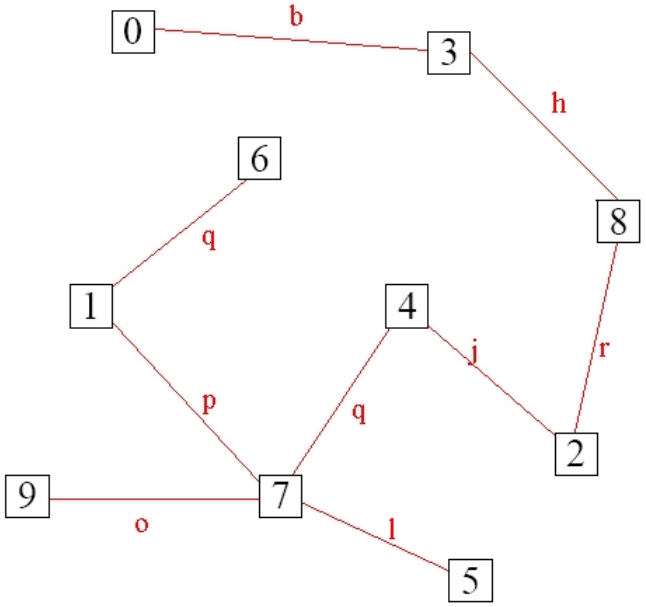


Kante	Kosten
a	42
b	13
c	46
d	9
e	50
f	25
g	24
h	21
i	43
j	6

Kante	Kosten
k	46
l	8
m	39
n	45
o	23
p	26
q	22
r	7
s	62

Lösung:

Es werden folgende Kanten in der angegebenen Reihenfolge aufgenommen: j, r, l, d, b, h, q, o, p
Es entsteht schließlich der folgende Baum:



Aufgabe 4.6

Aufgabenstellung:

Führen Sie anhand des in Aufgabe 4.5 abgebildeten Graphen und der in der entsprechenden Tabelle gegebenen Kantenkosten den Algorithmus für das Finden des minimalen Spannbaums von Prim durch. Beginnen Sie mit Knoten 4.

(a) Geben Sie den Zustand aller benätigten Datenstrukturen (Knotenmenge, Kantenmenge und Gewicht des aktuellen Spannbaums, Menge der freien Knoten) nach jeder Iteration des Algorithmus an. Schreiben Sie insbesondere in jeder Iteration deutlich den neu hinzugekommenen Knoten und die entsprechende Kante dazu.

(b) Markieren Sie am Ende Ihrer Berechnungen jene Kanten des Graphens in der Abbildung, die den minimalen Spannbaum bilden, und geben Sie das Gewicht des minimalen Spannbaums an.

(c) Wie würde sich das Ergebnis ändern, wenn Knoten 6 der Startknoten wäre? Begründen Sie Ihre Antwort mit wenigen Worten.

Lösung:

(a)

Iteration	Knoten	Kanten	Gewicht	freie Knoten
0	4	-	0	0,1,2,3,5,6,7,8,9
1	2	j	6	0,1,3,5,6,7,8,9
2	2,4,8	j,r	13	0,1,3,5,6,7,9
3	2,3,4,8	h,j,r	34	0,1,5,6,7,9
4	0,2,3,4,8	b,h,j,r	47	1,5,6,7,8
5	0,2,3,4,7,8	b,h,j,q,r	69	1,5,6,9
6	0,2,3,4,5,7,8	b,h,j,l,q,r	77	1,6,9
7	0,2,3,4,5,7,8,9	b,h,j,l,o,q,r	100	1,6
8	0,1,2,3,4,5,7,8,9	b,h,j,l,o,p,q,r	126	6
9	0,1,2,3,4,5,6,7,8,9	b,d,h,j,l,o,p,q,r	135	-

(b) Der entstehende Graph ist identisch mit jenem in Aufgabe 4.5. Das Gesamtgewicht beträgt 135.

(c) Das Ergebnis des Algorithmus muss hier gleichbleiben, da der MST eindeutig ist (es existieren zwar zwei Kanten mit gleichem Gewicht, jedoch werden diese nicht hinzugefügt). Lediglich die Reihenfolge des Hinzufügens von Knoten und Kanten ändert sich.

Iteration	Knoten	Kanten	Gewicht	freie Knoten
0	6	-	0	0,1,2,3,4,5,7,8,9
1	1,6	d	9	0,2,3,4,5,7,8,9
2	1,6,7	d,p	35	0,2,3,4,5,8,9
3	1,5,6,7	d,l,p	43	0,2,3,4,8,9
4	1,4,5,6,7	d,l,p,q	65	0,2,3,8,9
5	1,2,4,5,6,7	d,j,l,p,q	71	0,3,8,9
6	1,2,4,5,6,7,8	d,j,l,p,q,r	78	0,3,9
7	1,2,3,4,5,6,7,8	d,h,j,l,p,q,r	99	0,9
8	0,1,2,3,4,5,6,7,8	b,d,h,j,l,p,q,r	112	9
9	0,1,2,3,4,5,6,7,8,9	b,d,h,j,l,o,p,q,r	135	-

Aufgabe 4.7

Aufgabenstellung:

Gegeben sind zwei Graphen G_1 und G_2 mit jeweils einer Million Knoten. Graph G_1 hat vier Millionen Kanten, während G_2 250 Milliarden Kanten hat. Die Kanten in beiden Graphen haben ganzzahlige Kosten. Sie wollen nun in beiden Graphen jeweils einen aufspannenden Baum mit minimalen Kosten berechnen.
Welchen Algorithmus benutzen Sie für G_1 und welchen für G_2, um möglichst kurze Laufzeiten zu erreichen? Begründen Sie Ihre Antwort, indem Sie die Laufzeiten der von Ihnen verwendeten Algorithmen angeben. Dabei sollten Sie auch angeben, warum die Algorithmen die von Ihnen angegebenen Laufzeiten aufweisen (kurze Beschreibung der Algorithmen und der Funktionsweise und Eigenschaften der verwendeten Datenstrukturen).

Lösung:

Für einen dünnen Graphen ($|E| \approx \Theta(|V|)$) ist der Algorithmus von Kruskal mit seiner Laufzeit in $O(|V| \cdot |E|)$ (bzw. $O(|E| \cdot log|E|)$ bei der Verbesserung des Algorithmus mittels Union Find) besser geeignet; die Laufzeit von $O(|V| \cdot |E|)$ ergibt sich dabei folgendermaßen:

Das Sortieren der Kanten liegt mittels HeapSort oder QuickSort in $O(|E| \cdot log|E|)$. Die Schleife des Algorithmus, in welcher eine Kante nach der anderen durchlaufen wird, wird $|E|$-mal, eben für jede Kante einmal, ausgeführt. Mittels Tiefensuche wird jeweils überprüft, ob das Hinzufügen einer Kante einen Kreis erzeugt oder nicht; der Aufwand für diese Überprüfung liegt in $O(|V|)$. Das Suchen der hinzuzufügenden Kanten liegt also in $O(|V|) \cdot O(|E|) = O(|V| \cdot |E|)$. Insgesamt ergibt das eine Laufzeit des Algorithmus von Kruskal von $O(|E| \cdot log|E|) + O(|E| \cdot |V|) = O(|E| \cdot |V|)$. Darüber hinaus ermöglicht Union-Find eine nahezu lineare Laufzeit der Schleife, sodass die Laufzeit durch das Kantensortieren bestimmt wird, also in $O(|E| \cdot log|E|)$ liegt.

Der Algorithmus von Prim sucht |V|-mal (so lange, bis alle Knoten verbunden sind) nach der günstigsten Kante, die den bereits zusammenhängenden Graphen mit einem noch nicht verbundenen Knoten verbindet. Die Laufzeit für die Suche nach einer derartigen Kante liegt in $O(|V|)$, die Laufzeit für den gesamten Algorithmus also in $O(|V|) \cdot O(|V|) = O(|V|^2$. Dieser Algorithmus ist daher für dichte Graphen vorzuziehen ($|E| \approx \Theta(|V|^2)$).

Aufgabe 4.8

Aufgabenstellung:

Beim fraktionalen Rucksackproblem darf man im Gegensatz zu dem in der Vorlesung vorgestellten 0/1-Rucksackproblem nicht nur ganze, sondern beliebig große Bruchteile von Gegenständen einpacken. Geben Sie den Pseudocode für einen Greedy-Algorithmus an, um das fraktionale Rucksackproblem zu lösen. Wie ist die Laufzeit Ihres Algorithmus in Θ-Notation? Liefert Ihr Greedy-Algorithmus immer eine optimale Lösung?

Lösung:

```
FractionalKnapsack(c, w, K) {
  // Sortiere absteigend nach c/w

  i = 1;
  weight = 0;
  cost = 0;
  while(weight < K && i <= count(c)) {
    if(K-weight > w[i]) {
      weight += w[i];
      cost += c[i];
    }
    else {
      cost += c[i] * (K-weight) / w[i];
      weight += K;
    }
  }
}
```

c ist ein Array der Kosten der einzelnen Gegenstände, w ein Array mit den einzelnen Gewichten. K ist das Grenzgewicht des Rucksacks, *count(c)* liefert die Anzahl der Elemente des Arrays c, also die Gesamtanzahl der Gegenstände.

Die Laufzeit für die Sortierung beläuft sich auf $\Theta(n \cdot log\, n)$ (wobei n die Anzahl der Gegenstände ist), die Laufzeit der Schleife liegt zwischen $\Theta(n)$ im Average und Worst Case sowie $\Theta(1)$ im Best Case. Insgesamt ergibt sich somit eine Laufzeit von $\Theta(n \cdot log\, n)$. Selbstverständlich liefert dieser Greedy-Algorithmus immer eine optimale Lösung.

Aufgabe 4.9

Aufgabenstellung:

Wie viele mögliche Lösungen finden Sie für das 5-Damen-Problem? Skizzieren Sie diese.

Lösung:

Durch Probieren findet man zunächst die folgende Lösung:

X				
			X	
	X			
				X
		X		

Horizontales sowie vertikales Spiegeln führen zu diesen Lösungen:

				X
	X			
			X	
X				
		X		

		X		
X				
			X	
	X			
				X

		X		
				X
	X			
X				

Durch Spiegelung der ersten Lösung an der Diagonale von links oben nach rechts unten erhält man diese Lösung:

X				
		X		
				X
	X			
			X	

Wieder wird horizontal und vertikal gespiegelt...

				X
		X		
X				
			X	
	X			

	X			
			X	
X				
		X		
				X

			X	
	X			
				X
		X		
X				

Ein letztes Mal muss noch gesucht werden, um diese Lösung zu finden; horizontale und vertikale Spiegelung liefern hier allerdings dasselbe Ergebnis, daher erhält man durch Spiegelung nur eine weitere Lösungen (und nicht drei, wie bei den beiden anderen Lösungen oben).

	X			
				X
		X		
X				
			X	

			X	
X				
		X		
				X
	X			

Da dies alle Lösungen für das Fünf-Damen-Problem sind, gibt es also zehn Lösungen dieses Problems.

Aufgabe 4.10

Aufgabenstellung:

Wenden Sie den Algorithmus für Dynamische Programmierung auf das folgende Beispiel für das 0/1-Rucksackproblem an (Grenzgewicht $K = 20$). Veranschaulichen Sie alle Schritte, die zur Konstruktion einer Lösung führen.

Gegenstände	**A**	**B**	**C**	**D**
Gewichte	5	8	6	10
Werte	5	7	5	8

Lösung:

Verwendet man einen Entscheidungsbaum, so kommt man (in diesem Fall) sehr schnell zur Lösung. Packt man die Gegenstände A, B sowie C ein, allerdings nicht den Gegenstand D, so erhält man ein Gewicht von 19 bei einem Wert von 17.