Datenstrukturen und Algorithmen: Übungsblatt #9

Abgabe am 28. Juni 2018

Finn Hess (378104), Jan Knichel (377779), Paul Orschau (381085)

22.6.2018

 Finn Hess (378104), Jan Knichel (377779), Paul Orschau (381085)

Aufgabe 1

Teil a)
Gerichtete Graphen mit $n \in \mathbb{N}$ Knoten:
Die Adjazenzmatrix ist also eine $n \times n$ Matrix, in der jeder Eintrag 0 oder 1 ist.
Es gibt also $n \cdot n = n^2$ Werte mit je zwei Möglichkeiten.

$$\Rightarrow 2^{(n^2)}$$

Teil b)
Ungerichtete Graphen mit $n \in \mathbb{N}$ Knoten:
Die Adjazenzmatrix eine symmetrische $n \times n$ Matrix, wobei auf der Diagonalen nur Nulleinträge sind (keine Kanten von i nach i für $i \in n$)
Es gibt also $n \cdot n = n^2$ Werte. Von diesen Werten sind n Nullwerte auf der Diagonalen vorgegeben. Von den $n^2 - n$ verbliebenen Werten ist aufgrund der Symmetrie nur die Hälfte frei wählbar, also $\frac{n^2-n}{2}$ Werte mit je zwei Möglichkeiten.

$$\Rightarrow 2^{\frac{n^2-n}{2}}$$

Teil c)
einfache Pfade der Länge k für vollständig ungerichtete Graphen:
$k = 0 : n$ Pfade (entspricht Anzahl der Knoten)
$k = 1 : \frac{n(n-1)}{2}$ Pfade (entspricht Anzahl Kanten: jeder Knoten hat $(n-1)$ Kanten
$\Rightarrow$ Kanten gesamt: $\sum_{i=1}^{n-1} i$)
$k \geq 2$: Es gibt n mögliche Startknoten mit $n-1$ Nachfolgern. Da nur nach einfachen Pfaden gesucht wird, bleiben danach nur $n-2$ Möglichkeiten usw.

$$\Rightarrow \prod_{i=0}^{k} (n-i)$$

Teil d)
Zykel der Länge höchstens 4 für vollständig ungerichtete Graphen:
Zykel in ungerichteten Graphen haben laut Definition mindestens die Länge drei.
Länge 3:
n mögliche Startpositionen mit je $n-1$ Nachfolgern. Da die Knoten paarweise verschieden sein müssen, gibt es danach noch $n-2$ Möglichkeiten. Danach muss die Kante zum Startknoten gewählt werden.
$\Rightarrow n(n-1)(n-2)$
Länge 4:
Nach den $n-2$ möglichen Knoten gibt es im nächsten Schritt $n-3$ mögliche Nachfolger, da von den $n-1$ Nachbarknoten zwei ausgeschlossen sind: der Vorgängerknoten und der Startknoten.
Danach muss zum Startknoten zurückgegangen werden.
$\Rightarrow n(n-1)(n-2)(n-3)$

$$\Rightarrow n(n-1)(n-2) + n(n-1)(n-2)(n-3) = n(n-1)(n-2)^2$$

Aufgabe 1 wird auf nächster Seite fortgesetzt...

Teil e)

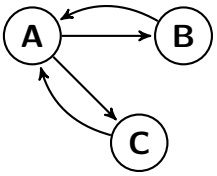
(i) $\hat{G}$ symmetrisch:
Für $(u, v) \in E$ gilt per Definition für transponierte Graphen:

$$(u, v) \in E \Leftrightarrow (v, u) \in E'$$

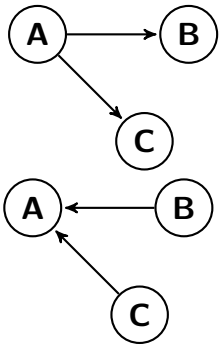
Mit $\hat{E} = E \cup E'$ gilt $(u, v) \in \hat{E}$ und $(v, u) \in \hat{E}$
Das beinhaltet die Definition für Symmetrie: $(u, v) \in \hat{E} \Rightarrow (v, u) \in \hat{E}$

(ii) $\hat{G}$ stark zusammenhängend $\Rightarrow$ G oder G^T stark zusammenhängend:
Gegenbeispiel, $\hat{G}$ stark zusammengängend:

G nicht stark zusammengängend:



G^T nicht stark zusammenhängend:

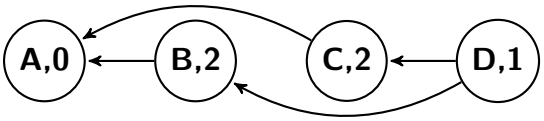


$\Rightarrow$(ii) gilt nicht.

(iii) G oder G^T stark zusammenhängend $\Rightarrow$ $\hat{G}$ stark zusammenhängend:
Durch die Menge der Kanten E oder E' sind per Voraussetzung schon alle Knoten von jedem anderen beliebigen Knoten erreichbar. Da $\hat{E}$ die Vereinigung der beiden Kantenmengen darstellt und die Knoten in $\hat{G}$ die gleichen wie in G und G^T sind, müssen in $\hat{G}$ also auch alle Knoten erreichbar sein, $\hat{G}$ ist also stark zusammenhängend.

(iv) G schwach zusammenhängend $\Leftrightarrow$ G^T schwach zusammenhängend:
Bei der Eigenschaft "schwach zusammenhängend" werden die Graphen als ungerichtete Graphen betrachtet, die Pfeilrichtungen werden als ignoriert. Da durch das Transponieren nur die Pfeilrichtungen vertauscht werde, gilt: $G_{ungerichtet} = G^T_{ungerichtet}$. Die beiden ungerichteten Graphen sind also gleich, daher gilt insbesondere, dass immer beide oder keiner stark zusammenhängend ist, also für die gerichteten Graphen:
G und G^T sind entweder beide schwach zusammenhängend oder keiner ist es.
Folglich: G schwach zusammenhängend $\Leftrightarrow$ G^T schwach zusammenhängend.

Teil f)
Jeder knotengewichtete DAG hat genau einen kritischen Pfad.
Der kritische Pfad ist der längste Pfad (bezogen auf das Gesamtgewicht) im DAG und startet bei einem Knoten v_0 ohne Abhängigkeiten. Falls es ein eindeutiges Maximum $eft(v_k)$ gibt, stimmt die Aussage. Dies muss allerdings nicht der Fall sein, wenn z.B. zwei Knoten gleich gewichtet sind:



Kritischer Pfad 1:

$$A = v_0, \;\; B = v_1, \;\; D = v_2, \;\; eft(v_2) = 3$$

Kritischer Pfad 2:

$$A = v_0, \;\; C = v_1, \;\; D = v_2, \;\; eft(v_2) = 3$$

Aufgabe 2

Teil a)
These: Die Funktion `DFS(...)` wird auf jeden der n Knoten genau einmal ausgeführt.
Beweis: Am Anfang sind offensichtlich alle Knoten **weiß** gefärbt (Zeile 16) und am Ende alle **schwarz**. Eine komplette Tiefensuche findet alle Knoten, selbst wenn es gar keine Kanten gibt, da die `for`-Schleife in Zeile 17 alle Knoten **mindestens einmal** prüft und ggf. `DFS(...)` **mindestens einmal** ausführt. Da vor den beiden Aufrufen von `DFS(...)` in den Zeilen 7 und 19 jeweils die Farbe des Knotens überprüft wird, kann die Funktion **nicht öfter als einmal** pro Knoten ausgeführt werden, denn `DFS(...)` färbt jeden Knoten sofort grau (Zeile 3) und verhindert somit, dass ein Knoten doppelt besucht wird.
Folgerung: Zeilen 2, 3, 11, 12 liegen in $\Theta(n)$.

Da die `for`-Schleife in Zeile 4 für jeden Knoten, also für jedes n, genau einmal erreicht wird, liegen Zeilen 4 und 5 in $\Theta(n^2)$. Jeder Eintrag der Adjazenzmatrix wird also genau einmal überprüft, und abhängig von der Anzahl von `true` Werten, also der Anzahl der Kanten m, wird Zeile 6 ausgeführt, sie liegt also in $\Theta(m)$. Da aber gilt $m \leq n^2$ kann diese Zeile die asymptotische Laufzeit nicht weiter verschlechtern, ausschlaggebend sind also weiterhin Zeilen 4 und 5.
Insgesamt gilt also:

$$T(n, m) \in \Theta(n^2) \tag{1}$$

Teil b)
Siehe Begründung der a), alle `print(...)` Anweisungen stehen in der `DFS(...)` Funktion, welche genau n-mal ausgeführt wird, somit gilt:

$$T(n, m) = 2n \in \Theta(n) \tag{2}$$

Aufgabe 3

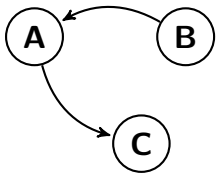
SCCs in gerichteten Graphen

Teil a)
w= white, g=grey, b=black

1. Phase	1	2	3	4	5	6	7	8	9	10	11
1: color	b	w	w	b	b	b	b	b	b	b	b
S	4	11	8	9	10	7	6	5	1		
2: color	b	b	b	b	b	b	b	b	b	b	b
S	4	11	8	9	10	7	6	5	1	3	2

2.Phase	1	2	3	4	5	6	7	8	9	10	11
1: color	w	b	b	w	w	w	w	w	w	w	w
S	4	11	8	9	10	7	6	5	1	3	
Suc	0	2	2	0	0	0	0	0	0	0	0
2: color	b	b	b	w	b	b	b	b	b	b	b
S	4	11	8	9	10	7	6	5			
Suc	1	2	2	0	1	1	1	1	1	1	1
3: color	b	b	b	b	b	b	b	b	b	b	b
S											
Suc	1	2	2	4	1	1	1	1	1	1	1

Teil b)
Kondensationsgraph:
A: 1,5,6,7,8,9,10,11
B: 2,3
C: 4



Aufgabe 4

Teil a)

topo(v)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Knoten v	2	1	16	15	8	9	13	14	7	10	12	17	4	6	11	19	3	20	5	18

Teil b)

```
1  void topoSort(List adj[n], int n, int &topo[n]) {
2    int color[n] = WHITE; topoNum = 0;
3    for (int start=0;start<n;start++) {
4      // Starte so lange von diesem Knoten,
5      // bis es keine Pfade von hier aus mehr gibt,
6      // dann ist der Startknoten selbst das Ende
7      // eines Pfades gewesen und schwarz
8      while (color[start] == WHITE) {
9        // Beginne (wieder) vom Startknoten aus
10       int next = start;
11       // Gehe so lange weiter,
12       // bis es keinen naechsten weissen Knoten mehr gibt
13       while (color[next] == WHITE) {
14         // Merke die den (neuen) aktuellen Knoten
15         int curr = next;
16         // Durchsuche alle seine Nachbarn nach
17         // einem geeigneten naechsten Knoten
18         for (neighbor in reverse(adj[curr])) {
19           if (color[neighbor] == WHITE) {
20             // Kommt als naechster Knoten in Frage
21             next = neighbor;
22           }
23           // Durch die umgekehrte Liste ist sichergestellt,
24           // dass die Reihenfolge stimmt
25         }
26         if (curr == next) {
27           // Kann nicht mehr weiter, da es keinen geeigneten
28           // naechsten Knoten mehr gibt, z.B. weil alle Nachbarn
29           // schon schwarz sind oder es keine Nachbarn gibt
30           topo[curr] = ++topoNum;
31           color[curr] = BLACK;
32           // Jetzt wird in die erste while Schleife gesprungen,
33           // also wieder mit dem Startknoten begonnen
34         } else {
35           // Fahre beim Nachbarn fort
36         }
37       }
38     }
39   }
40 }
```

Aufgabe 5

Knoten	A	B	C	D	E	F	G	H	I	J	K	L	M	N
est	16	11	10	8	13	3	6	6	0	4	0	0	0	0
critDep	E	G	D	F	C	M	J	L	-1	N	-1	-1	-1	-1
eft	20	13	13	10	16	8	11	9	3	6	3	6	3	4

Schwarzfärbung: N,J,G,B,M,F,K,D,I,C,L,H,E,A
Kritischer Pfad: A $\rightarrow$ E $\rightarrow$ C $\rightarrow$ D $\rightarrow$ F $\rightarrow$ M
Gesamtdauer: 20

Aufgabe 6

```
1  bool isBipartit(list adj[n]){
2          queue q;
3          q.enqueue(1);
4          while(!q.empty()){
5                  u = q.dequeue();
6                  foreach v in adj[u]{
7                          if(!v.color){
8                                  if(u.color == red)      v.color = black;
9                                  else                    v.color = red;
10                                 q.enqueue(v);
11                         } else if (v.color == u.color) return false;
12                 }
13         }
14         return true;
15 }
```

Die Funktion basiert auf der Breitensuche. Wir gehen von einem zusammenhängenden Graphen aus.
Die Korrektheit der Funktion besteht darin, dass alle durch eine Kante verbundenen Knoten abwechselnd rot und schwarz gefärbt werden. So werden sie jeweils einer Menge U und W zugeordnet.
Dass keine Kanten innerhalb dieser Mengen existieren, wird durch Abgleich der Farben gewährleistet. Haben zwei aufeinanderfolgende Knoten die gleiche Farbe, wird `false` zurückgegeben.
Da es sich um zusammenhängende Graphen handelt, wird durch die Breitensuche der gesamte Graph abgedeckt.