RWTH Aachen **SoSe 2007**
Lehrgebiet Theoretische Informatik Blatt 8
Kneis—Mölle—Richter—Rossmanith—Thiemann 4.6.2007

Übungen zur Vorlesung Datenstrukturen und Algorithmen

T19

Können Sie Quicksort so modifizieren, daß die Worst-Case-Laufzeit von $\Theta(n^2)$ auf $O(n \log n)$ sinkt? Konzentrieren Sie sich auf den Divide-Teil, also nicht auf die Rekursion!

Lösungsvorschlag:
Wir haben in der Vorlesung gesehen, daß sich der Median eines n-elementigen Arrays in linearer Zeit ermitteln läßt. Wenn wir diesen als Pivot-Element verwenden, sind die beiden rekursiv zu bearbeitenden Teile im wesentlichen gleich groß. Dann folgt wie bei Mergesort die Laufzeit $O(n \log n)$.

T20

Für die Vorlesung war noch zu zeigen, daß aus der Rekursionsgleichung

$$C_n = n + 1 + \frac{1}{n} \sum_{i=1}^{k-1} C_{n-i} + \frac{1}{n} \sum_{i=k+1}^{n} C_{i-1}$$

mit $C_0 = C_1 = 0$ folgt: $C_n \leq 4n$.

a) Zeigen Sie zuerst, daß $\displaystyle\sum_{i=1}^{k-1} C_{n-i} + \sum_{i=k+1}^{n} C_{i-1} \leq 2 \sum_{i=\lceil n/2 \rceil}^{n-1} C_i$ gilt!

b) Beweisen Sie nun die Aussage $C_n \leq 4n$ mittels Induktion!

Lösungsvorschlag: Die beiden Summen in der Rekursionsgleichung enthalten die Summanden $C_{n-k+1}, \dots, C_{n-1}$ bzw. $C_k, \dots, C_{n-1}$.
Betrachten wir zunächst den Fall $k = \lceil n/2 \rceil$. Falls n ungerade ist, dann gilt

$$n - k + 1 = n - \lceil n/2 \rceil + 1 = \lfloor n/2 \rfloor + 1 = \lceil n/2 \rceil = k$$

und jeder der Summanden $C_{\lceil n/2 \rceil}$ bis C_{n-1} tritt genau zweimal auf. Für gerade n hingegen gilt

$$n - k + 1 = n - \lceil n/2 \rceil + 1 = n/2 + 1 = k + 1,$$

so daß der Summand $C_{\lceil n/2 \rceil}$ nur einmal auftritt, alle weiteren jedoch nach wie vor doppelt. In beiden Fällen gilt also die Ungleichung.
Wenn wir k verschieben, kommt pro Verschiebung ein Summand C_i mit $i \leq \lceil n/2 \rceil$ dazu, während ein Summand C_j mit $j \geq \lceil n/2 \rceil$ wegfällt. Da die C_i monoton steigen, bleibt die Ungleichung also stets gültig.

Wir kommen zum zweiten Teil. Der Induktionsanfang ist wegen $C_0 = C_1 = 0$ schon erledigt. Wir folgern nun mit vollständiger Induktion

$$\begin{aligned}
C_n &= n+1+\frac{1}{n}\sum_{i=1}^{k-1} C_{n-i} + \frac{1}{n}\sum_{i=k+1}^{n} C_{i-1} \\
&\overset{(a)}{\le} n+1+\frac{2}{n}\sum_{i=\lceil n/2\rceil}^{n-1} C_i \\
&\overset{IH}{\le} n+1+\frac{8}{n}\sum_{i=\lceil n/2\rceil}^{n-1} i \\
&= n+1+\frac{8}{n}\Big(\sum_{i=1}^{n-1} i - \sum_{i=1}^{\lceil n/2\rceil -1} i\Big) \\
&= n+1+\frac{8}{n}\big(n(n-1)/2 - (\lceil n/2\rceil)(\lceil n/2\rceil - 1)/2\big) \\
&= n+1+\frac{4}{n}\big(n(n-1) - (\lceil n/2\rceil)(\lceil n/2\rceil - 1)\big) \\
&\le n+1+4((n-1)-(1/2)(n/2-1)) = n+1+4(n-1-n/4+1/2) = 4n-1.
\end{aligned}$$

T21

Was sind die Best-, Average- und Worst-Case-Laufzeiten von Straight-Radixsort? Ist dieses Verfahren stabil? Arbeitet es in-place?

```
procedure Straight-Radix-Sort:
  for i=w,...,1 do
    pos_0 ← 1; pos_1 ← n+1;
    for k=1,..., n do pos_1 ← pos_1−A[k,i] od;
    for k=1,..., n do
      if A[k,i]=0 then B[pos_0] ← A[k]; pos_0 ← pos_0+1
                  else B[pos_1] ← A[k]; pos_1 ← pos_1+1 fi
    od;
    A ← B;
  od;
```

Lösungsvorschlag:

Man sieht sofort, daß die Schleifen eine Laufzeit von $\Theta(wn)$ erzwingen. Die Reihenfolge der Elemente wird nur in der zweiten inneren Schleife verändert, wobei wir aber bereits wissen, daß die dortige Umsortierung stabil ist. Aufgrund der Benutzung des zweiten Arrays B ist das Verfahren offensichtlich nicht in-place.

H16 (8 Punkte)
Wie schnell lassen sich die folgenden Operationen in gerichteten Graphen $G = (V, E)$ durchführen, wenn diese durch Adjazenzmatrizen bzw. -listen dargestellt sind?

a) Adjazenztest ($(v, w) \in E$?) für zwei Knoten $v, w \in V$

b+c) Knoten einfügen bzw. löschen

d+e) Kanten zwischen bestehenden Knoten einfügen bzw. löschen

f+g) Alle Vorgänger bzw. Nachfolger eines Knotens $v \in V$ ermitteln

h) Bildung des Komplementgraphen (V, E') mit $E' = (V \times V) \setminus E$

Nehmen Sie hierzu an, daß Matrizen als dynamische zweidimensionale Arrays und Listen als doppelt verkettete Listen gespeichert werden.

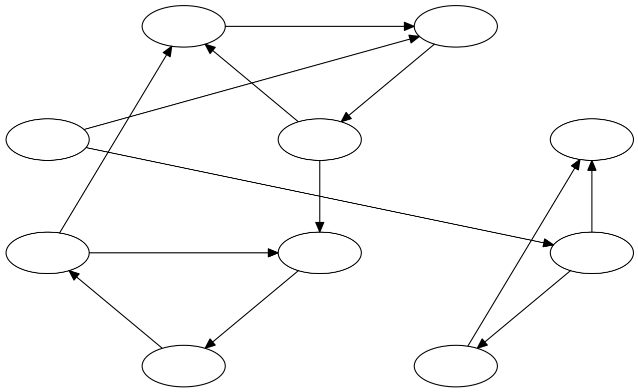
Lösungsvorschlag:
Wir betrachten zunächst die Darstellung durch Adjazenzmatrizen. Hier dauert der Adjazenztest lediglich $O(1)$, da nur die richtige Adresse in der Matrix zu bestimmen ist. Beim Einfügen eines neuen Knotens muß die Matrix um je eine Zeile und Spalte vergrößert werden, die anfangs mit Nullen zu initialisieren sind, womit sich die Zeitkomplexität $O(n)$ ergibt. Beim Löschen mag es für viele Anwendungen reichen, die entsprechende Zeile bzw. Spalte mit Nullen zu überschreiben, was denselben Aufwand bedeutet. Im allgemeinen Fall muß die Matrix aber neu aufgebaut werden, was $O(n^2)$ Zeit erfordert. Das Einfügen und Löschen von Kanten ist genau so komplex wie der Adjazenztest: $O(1)$. Zum Finden der Nachbarn muß die gesamte Zeile bzw. Spalte durchlaufen werden, was $O(n)$ dauert. Den Komplementgraphen erhalten wir durch Negation aller $O(n^2)$ Bits.

Für unsortierte Adjazenzlisten, die in einem Array gehalten werden, ergeben sich die folgenden Komplexitäten. Hierbei bezeichne $d(v)$ den Ausgangsgrad eines Knoten $v \in V$. Der Adjazenztest dauert $O(d(v)) = O(n)$, weil die Adjazenzliste von v durchsucht werden muß. Wenn wir neue Knoten einfügen, geht das sogar in $O(1)$: es muß nur eine neue Adjazenzliste an ein neues Arrayelement gehängt werden. Währenddessen dauert das Löschen $O(m) = O(n^2)$: ein Knoten kann in jeder der $n - 1$ verbleibenden Adjazenzlisten auftauchen. Das Einfügen einer Kante (v, w) geht ebenfalls in $O(1)$, während das Löschen $O(d(v)) = O(n)$ dauert. Wenn wir die Nachfolger eines Knotens zurückliefern wollen, muß eine Kopie der entsprechenden Adjazenzliste erzeugt werden – die Komplexität ist offenbar $O(d(v)) = O(n)$. Die Bestimmung der Vorgänger hingegen ist sehr aufwendig, da alle $O(m) = O(n^2)$ Listeneinträge durchlaufen werden müssen.

Die Bildung des Komplementgraphen benötigt ohnehin mindestens quadratische Zeit, weil aus einem Graphen mit m Kanten, $0 \leq m \leq n(n-1)$, einer mit $n(n-1)-m$ Kanten wird. Man beachte, daß dann $\max\{m, n(n-1) - m\} \geq n(n-1)/2$ gilt. Andererseits bewältigt der folgende Algorithmus die Komplementierung in $O(n^2)$: Wechsle zur Darstellung in Adjazenzmatrizen, benutze die dort angegebene Methode, und wechsle wieder zur alten Darstellung.

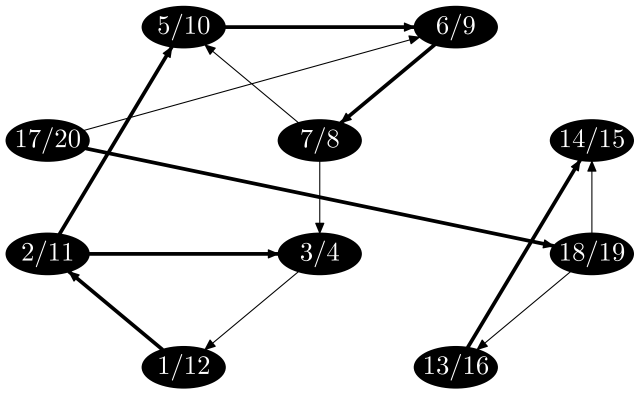
H17 (5 Punkte)

Führen Sie auf dem hier abgebildeten Graphen eine Tiefensuche durch. Geben Sie die *discovery*- und *finish*-Zeiten aller Knoten sowie die Typen aller Kanten an.



Lösungsvorschlag:

Der Graph könnte beispielsweise so aussehen:



Die dicken Kanten sind Baumkanten.

Querkanten verlaufen von 7/8 nach 3/4, von 17/20 nach 6/9 sowie von 18/19 nach 13/16 und 14/15. Die Kanten von 3/4 nach 1/12 und von 7/8 nach 5/10 sind Rückwärtskanten. Vorwärtskanten gibt es keine.

H18 (10 Punkte)

Entwerfen und analysieren Sie einen Algorithmus, der das folgende Problem in Zeit $O(|V| + |E|)$ löst. Gegeben sind ein gerichteter Graph $G = (V, E)$ und eine Zahl k. Gefragt ist, ob man k Schritte entlang gerichteter Kanten auf G machen kann: Gibt es eine endliche Folge $(v_1, \ldots, v_{k+1}) \in V^{k+1}$, so daß $(v_i, v_{i+1}) \in E$ für alle $i \in \{1, \ldots, k\}$?

Lösungsvorschlag:

Wir stellen zunächst mittels Tiefensuche fest, ob der Graph Kreise enthält. Falls dies gilt, dann lautet die Antwort natürlich „ja“. Andernfalls ist der Graph eine Menge von gerichteten azyklischen Graphen. Er ist also azyklisch, und die Antwort lautet genau dann „ja“, wenn es einen Pfad der Länge k gibt.

Die Grundidee unseres Algorithmus ist einfach. Wir wiederholen folgende Operation, bis der DAG leer ist: Bestimme alle Quellen und lösche sie. Die Anzahl der Iterationen ist dann genau die Anzahl von Knoten auf einem längsten Pfad. Die Korrektheit folgt mit

Induktion über die Knotenzahl in einem längsten Pfad: Ist diese null, dann ist der Graph leer und der Algorithmus führt keine einzige Operation aus. Ist sie größer als null, zum Beispiel k, dann werden die Quellen in der ersten Iteration entfernt und gleichzeitig sinkt die Knotenzahl des längsten Pfads um eins. Nach Induktionsvoraussetzung führt der Algorithmus jetzt noch genau $k-1$ Iterationen aus.

```
 List  S ← emptyset;
for v in V do d[v] ← 0; visited[v] ← false od;
for (u,v) in E do d[v] ← d[v]+1 od;
for v in V do
   if d[v]=0 then S ← S cup {x} fi
od;
length ← 0;
while S ≠ emptyset sp do
   N ← emptyset;
   for v in S do
      forall successors w plainof v do
        d[w] ← d[w]−1;
        if neg visited [w] ∧ d[w]=0 then
           N ← N cup {w};
            visited [w] ← true;
        fi
      od
   od;
   S ← N;
   length ← length+1;
od
return length;
```

Die Kunst besteht darin, diese Grundidee in linearer Zeit umzusetzen. Jeder Knoten $v \in V$ kommt in genau einer Iteration der while-Schleife in der Liste S vor: Die ursprünglichen Quellen werden nur am Anfang in S eingefügt und können nicht in N eingefügt werden, da sie keinen Vorgänger haben. Alle anderen Knoten können nur einmal in N eingefügt werden, da sie dann als *visited* markiert sind. Die forall-Schleife wird für jeden Knoten w nur sooft durchlaufen, wie es Kanten $(v,w) \in E$ gibt, insgesamt also nur $|E|$-mal. Ihr Rumpf benötigt nur konstante Zeit. Die Laufzeit ist daher $O(|V|+|E|)$.