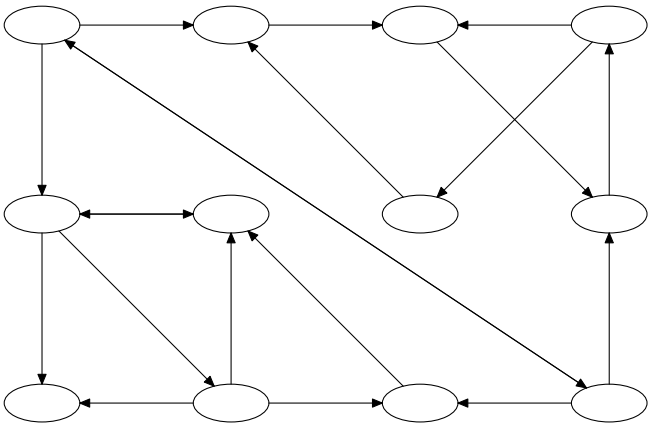
RWTH Aachen **SoSe 2011**
Lehrgebiet Theoretische Informatik
Emmes—Langer—Reidl—Rossmanith 2. August 2011

Klausur zur Vorlesung Datenstrukturen und Algorithmen

Aufgabe 1 (10 Punkte)

a) Führen Sie auf dem folgenden Graphen eine Tiefensuche durch und tragen Sie die *discovery*- und *finish*-Zeiten ein. Geben Sie außerdem den Typ jeder Kante an, indem sie alle Kanten mit B, V, R oder Q markieren. Die Buchstaben stehen für Baumkante, Vorwärtskante, Rückwärtskante und Querkante. Beachten Sie, daß es Doppelkanten gibt, die in Wirklichkeit aus zwei Kanten bestehen. Beide müssen beschriftet werden.



b) Wie kann man nach einer durchgeführten Tiefensuche einfach feststellen, ob der entsprechende gerichtete Graph kreisfrei ist?

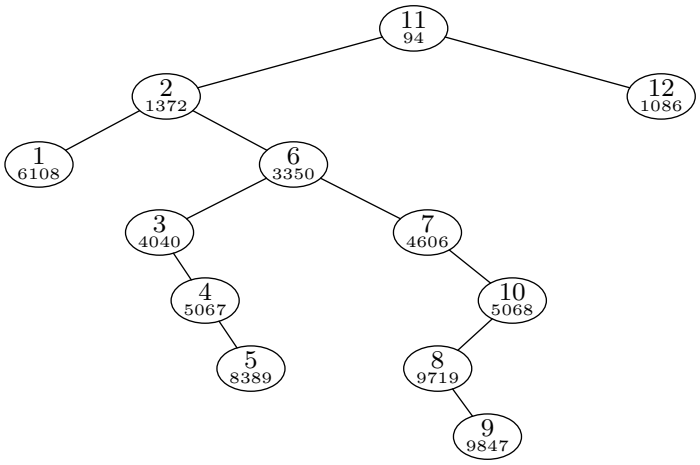
c) Falls sich herausstellt, daß der Graph tatsächlich kreisfrei ist, können wir die Knoten topologisch sortieren. Erklären Sie genau, wie man dies in $O(n)$ Schritten bewerkstelligen kann, falls der Graph n Knoten besitzt und wir die in der Tiefensuche gefundene Information verwenden können.

Aufgabe 2 (10 Punkte)

a) Beweisen Sie, daß ein Treap eindeutig ist, wenn alle Schlüssel und alle Prioritäten verschieden sind.

b) Geben Sie ein Gegenbeispiel für den Fall an, daß zwar alle Schlüssel verschieden sind, aber gleiche Prioritäten erlaubt sind. Genauer: Geben sie zwei verschiedene Treaps an, welche die gleichen Schlüssel enthalten.

c) Erklären Sie, wie ein Element aus einem Treap gelöscht wird. Wie sieht nebenstehender Treap aus, nachdem die 6 gelöscht wurde?



Aufgabe 3 (10 Punkte)

Die Produktionsaufträge $J_1, J_2, \ldots, J_n$ eines Unternehmens benötigen verschiedene Maschinen $M_1, M_2, \ldots, M_m$, wobei ein Auftrag auch mehrere Maschinen belegen kann. Ein Auftrag bringt natürlich einen gewissen Geldbetrag ein. Die Maschinen können entweder gekauft werden—diese Kosten entstehen dann nur einmal und jede weitere Nutzung ist kostenfrei—oder gemietet werden. Letzteres kostet einen vom Auftrag abhängigen Betrag (dieser Betrag variiert also von Auftrag zu Auftrag!).

Für solch ein Szenario mit gegebenen Aufträgen, Maschinen und Kosten soll eine gewinnmaximierende Menge von Aufträgen mit entsprechender Zuteilung von Maschinen berechnet werden. Beschreiben Sie ein allgemeines Verfahren, um dieses Problem möglichst effizient zu lösen. Begründen Sie, warum Ihr Verfahren korrekt funktioniert und geben Sie eine Schranke für die Laufzeit an, die nicht schlechter als $O(n^2m^2(n+m))$ sein sollte. Beantworten Sie die Frage, ob dies asymptotisch schneller ist, als stur alle 2^{n+m} Möglichkeiten zu probieren, welche Maschinen zu kaufen und welche Aufträge anzunehmen sind?

Benutzen Sie Ihr Verfahren, um eine optimale Lösung für das folgende Szenario zu berechnen. Die dritte Tabelle enthält die Mietkosten der Maschinen für die jeweiligen Aufträge. Eine leere Zelle bedeutet, daß diese Maschine für diesen Auftrag nicht benötigt wird. Zum Beispiel kostet Auftrag J_1 auf der Maschine M_1 30 Geldeinheiten (vorausgesetzt, M_1 wurde nicht gekauft).

Auftrag	Zahlung
J_1	80
J_2	80
J_3	120

Maschine	Kaufpreis
M_1	60
M_2	80
M_3	100
M_4	30

	M_1	M_2	M_3	M_4
J_1	30		50	
J_2		60	22	
J_3	30	30	30	30

Aufgabe 4 (10 Punkte)

Nebenstehendes rekursives Unterprogramm, das die Elemente $a[l], \ldots, a[r]$ in einem Array a in die richtige Reihenfolge sortieren soll, kann nicht direkt verwendet werden, um das Array $a[0], \ldots, a[N]$ mittels des Aufrufs *quicksort*$(0, N)$ zu sortieren.

Das Problem träte beispielsweise auf, wenn wir ein Array $a[0], \ldots, a[4]$ mit dem Inhalt $4, 2, 3, 2, 1$ nehmen und das Programm mittels *quicksort*$(0, 4)$ aufrufen.

```
procedure quicksort(L, R) :
  if R ≤ L then return fi;
  p := a[L]; l := L; r := R + 1;
  do
    do l := l + 1 while a[l] < p;
    do r := r − 1 while p < a[r];
    vertausche a[l] und a[r];
  while l < r;
  a[L] := a[l]; a[l] := a[r]; a[r] := p;
  quicksort(L, r − 1); quicksort(r + 1, R)
```

a) Erklären Sie, welches Problem bei dieser Eingabe auftritt.

b) Schreiben Sie ein Programm *sort*(N) in Pseudocode, Java oder C++, welches mithilfe obigen Unterprogramms das Array $a[0], \ldots, a[N]$ stets korrekt sortiert.

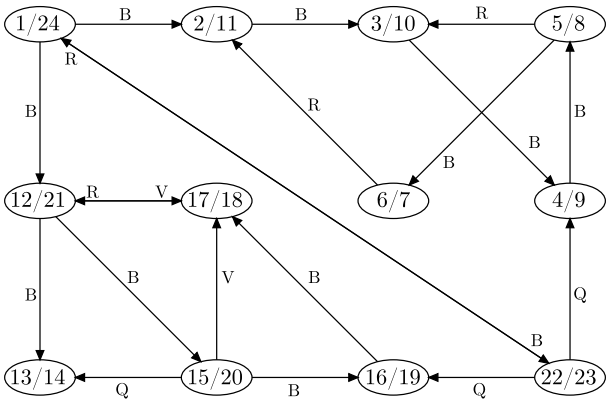
* * * Viel Erfolg! * * *

RWTH Aachen **SoSe 2011**
Lehrgebiet Theoretische Informatik
Emmes—Langer—Reidl—Rossmanith 2. August 2011

Klausur zur Vorlesung Datenstrukturen und Algorithmen
Lösungsvorschläge

Aufgabe 1

1.a



1.b

Sind nach der Tiefensuche Kanten vorhanden, die durch die *discovery*- und *finish*-Zeiten als Rückwärtskanten klassifiziert werden, so enhält der Graph mindenstens einen Kreis. Um dies zu sehen, nehmen wir an eine Kante (u, v) sei im Tiefensuchwald eine Rückwärtskante. Dann muss es innerhalb des Tiefensuchwalds einen Pfad von v nach u geben. Dieser Pfad bildet zusammen mit der Kante (u, v), welche offensichtliche nicht Teil des Pfads sein kann, einen Kreis im Ursprungsgraphen. Die umgekehrte Richtung ist aus der Vorlesung bekannt.

Somit ist ein Graph genau dann Kreisfrei, wenn im Tiefensuchwald *keine* Rückwärtskante vorhanden ist.

1.c

Aus der Vorlesung ist bekannt, daß die Anordnung der Knoten eines kreisfreien Graphen nach absteigenden *finish*-Zeiten eine topologische Sortierung ergibt. Es bleibt zu klären, wie dies in $O(n)$ Zeit möglich ist. Dazu mache man sich klar, daß die größte *finish*-Zeit den Wert $2n$ annimmt.

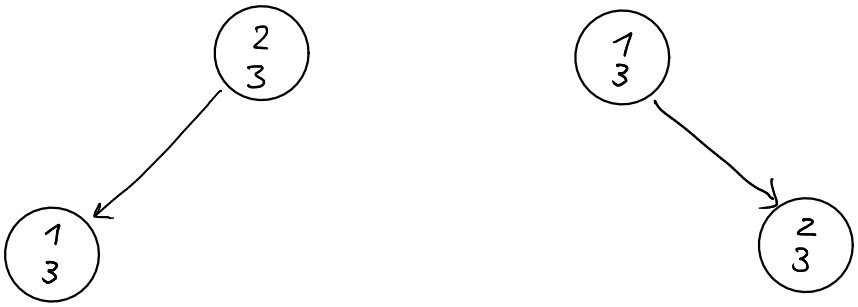
Wir können demnach ein Array der Größe $2n$ verwenden, um die Knoten des Graphen in linearer Zeit zu sortieren, dazu wird Knoten $v \in V(G)$ an Position i des Arrays geschreiben, wenn i gerade die *discovery*-Zeit von v ist. Anschließend durchläuft man das Array in umgekehrter Reihenfolge und gibt dabei die Knoten in topologische Reihenfolge aus (leere Zellen im Array werden einfach übergangen).

Beachten Sie, daß eine Sortierung mit Radixsort hier nicht die gewünschte lineare Laufzeit liefert, da die Kodierung der *discovery*-Zeiten $O(\log n)$ Bits benötigt und damit die Laufzeit nur durch $O(n \log n)$ beschränkt werden kann.

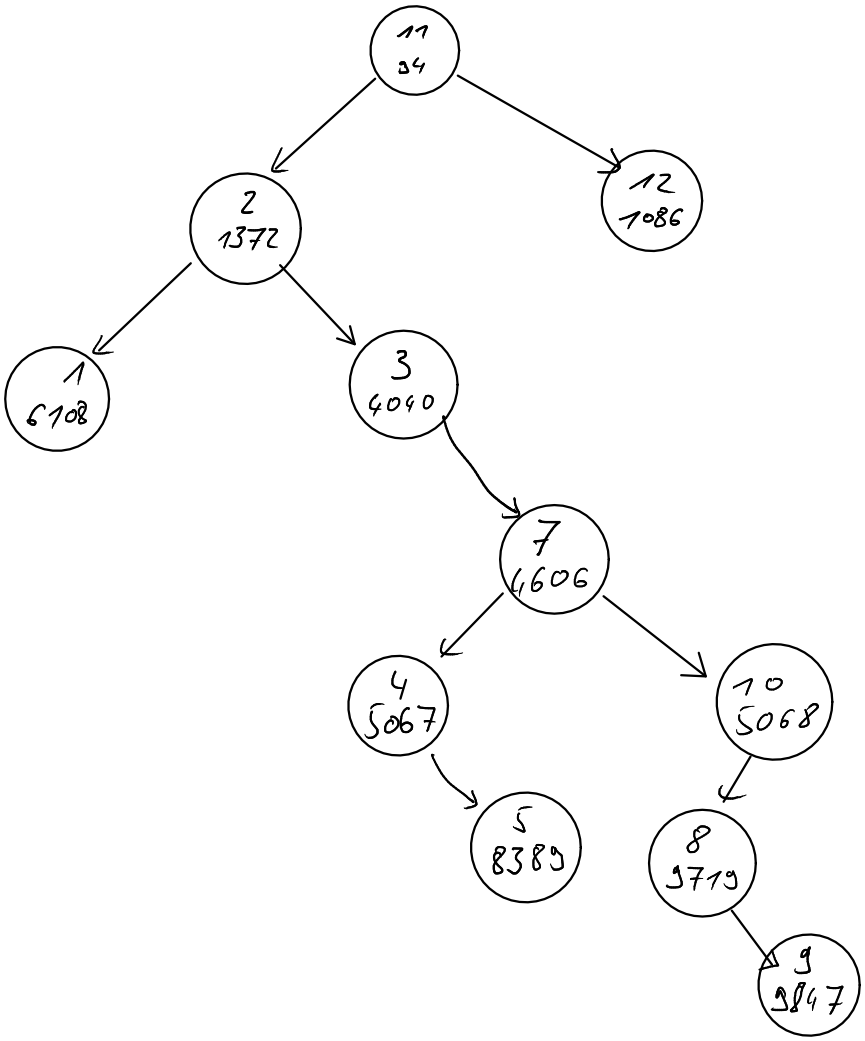
Aufgabe 2

2.a Wir verwenden Induktion über die Höhe des Treaps. Da alle Prioritäten verschieden sind, ist die Wurzel des Treaps wegen der Heapeigenschaft eindeutig. Wegen der Suchbaumeigenschaft sind im linken Unterbaum der Wurzel alle Elemente zu finden, die einen kleineren Schlüssel haben als die Wurzel, und im rechten Unterbaum alle Elemente, die einen größeren Schlüssel haben als die Wurzel. Diese Unterbäume sind nach dem Induktionsprinzip eindeutig, so daß auch die Eindeutigkeit des Treaps folgt.

2.b



2.c



Aufgabe 3

Wir verwenden eine Konstruktion analog zur Tutoraufgabe T38: Die Knotenmenge des Flußnetzwerks ist $\{s, t\} \cup \{J_1, \ldots, J_n\} \cup \{M_1, \ldots, M_m\}$. Die Kanten des Netzwerks sind wie folgt:

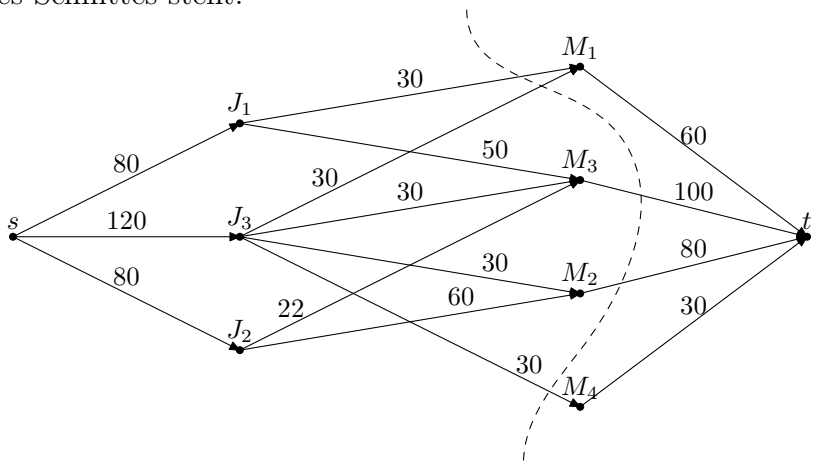
1. Von der Quelle s läuft zu jedem Knoten J_i eine Kante mit Kapazität j_i, wenn j_i die für einen angenommenen Auftrag J_i erhaltene Zahlung ist.

2. Von einem Auftrag J_i läuft eine Kante zur Maschine M_j, wenn M_j für den Auftrag J_i benötigt wird. Die Kapazität auf dieser Kante sind die Kosten, die für eine einmalige Anmietung der Maschine M_j für den Auftrag J_i entstehen (Tabelle 3).

3. Von jedem Knoten M_i läuft eine Kante mit Kapazität m_i in die Senke t, wenn m_i die Kosten sind, um die Maschine M_i zu kaufen.

Ein Min-Cut (S, T) für dieses Netzwerk ist natürlich endlich und ganzzahlig und kann effizient mit der Methode von Edmonds und Karp (die Methode von Ford und Fulkerson funktioniert, kann uns aber nicht die gewünschte Laufzeit garantieren) berechnet werden. Wir wählen nun diejenigen Aufträge aus, die in S enthalten sind. Ebenso kaufen wir alle Maschinen in S.

Zur Begründung, warum dieses Verfahren korrekt ist: Die Kanten im Schnitt entsprechen wie in Aufgabe T38 anschaulich einem *Verzicht* auf einen Geldbetrag, welche der Kapazität dieser Kante entspricht. Wird etwa eine Kante $s \to J_i$ für einen Auftrag J_i vom Schnitt (S, T) geschnitten, dann verzichtet man auf die Einnahmen in Höhe von j_i. Der Verzicht auf die Einnahmen kann günstiger sein (und zu einem kleineren Schnitt führen), als wenn man den Job annimmt und dafür viele andere Kanten (für Aufträge oder Maschinen) schneiden muß. Schneidet man analog eine Kante $M_i \to t$, dann muß man den entsprechenden Kaufpreis für eine Maschine bezahlen. Auch dieses kann günstiger sein, als die Maschine für jeden angenommen Job zu mieten. Liegt andererseits eine Maschine M_j in T, dann ist es offenbar günstiger, M_j für jeden angenommenen Auftrag J_i anzumieten; andernfalls könnte man im leicht einen kleineren Schnitt konstruieren, indem man M_j nach S verschiebt (und dem Fall entspricht, M_j zu kaufen), was im Widerspruch zur Minimalität des Schnittes steht.



Erinnern wir uns, daß die Laufzeit des Edmonds-Karp-Algorithmus $O(|V| \cdot |E|^2)$ beträgt. Das von uns konstruierte Netzwerk besitzt $n + m + 2$ Knoten und maximal $n + m + n \cdot m$ Kanten. Damit ist die Laufzeit durch $O((n + m + 2)(n + m + n \cdot m)^2) = O(n^2 m^2 (n + m))$

beschränkt. Um zu zeigen, daß dies sicherlicher besser ist, als alle 2^{n+m} Möglichkeiten auszuprobieren, müssen wir folgende Behauptung widerlegen:

$$2^{n+m} = O(n^2 m^2 (n+m))$$

Wenden wir auf beiden Seiten den Logarithmus an, so ergibt sich

$$n + m = O(\log n \cdot \log m \cdot \log n + m)$$

was sicherlich nicht gilt. Man beachte, daß dies nur möglich ist, weil der Logarithmus eine monotone Funktion ist.

Aufgabe 4

Falls das im ersten Aufruf gewählte Pivotelement $a[0]$ das größte Element im Array ist, läuft die erste innere **while**-Schleife für l über die Arraygrenzen hinaus, da der Test $a[l] < p$ jedesmal wahr ist und das Array nicht geeignet durch ein Sentinel-Element bewacht wird. Das Problem tritt nicht auf, falls das Pivotelement nicht das maximale Element im Array ist. Es reicht daher aus, im geforderten *sort*-Programm zunächst sicherzustellen, daß $a[N]$ das maximale Element des Array ist und als Sentinel dient.

procedure $sort(N)$:
  $max := 0; i := 0;$
  for $(i := 0; i \leq N; i{+}{+})$
    if $a[i] > a[max]$ **then** $max := i$ **fi**;
  vertausche $a[N]$ *und* $a[max]$;
  $quicksort(0, N-1);$