Name:
Vorname:
Matrikelnummer:

Klausur-ID:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders

25.09.2020

Klausur Algorithmen II

Aufgabe 1.	Kleinaufgaben	13 Punkte
Aufgabe 2.	FPT Algorithmen: d-Bounded Degree Deletion	10 Punkte
Aufgabe 3.	Geometrische Algorithmen: Pareto-optimale Punkte	9 Punkte
Aufgabe 4.	Flussalgorithmen: Partielle Ordnungen	9 Punkte
Aufgabe 5.	Randomisierte Algorithmen: Multi-Hashing	8 Punkte
Aufgabe 6.	Stringalgorithmen: LCP-Interval Bäume	11 Punkte

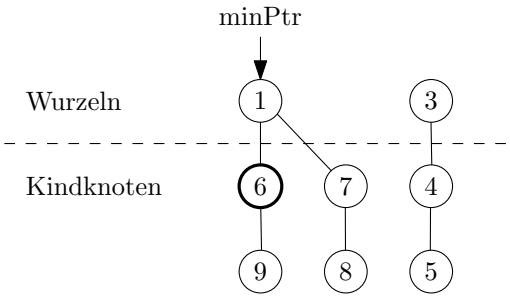
Bitte beachten Sie:

- Als Hilfsmittel ist nur **ein** DIN-A4 Blatt mit Ihren **handschriftlichen** Notizen zugelassen.
- **Schreiben** Sie auf **alle** Blätter der Klausur und Zusatzblätter Ihre **Klausur-ID**.
- Merken Sie sich Ihre **Klausur-ID** auf dem Aufkleber für den Notenaushang.
- Die Klausur enthält **20 Blätter**.
- Zum Bestehen der Klausur sind 20 Punkte hinreichend.

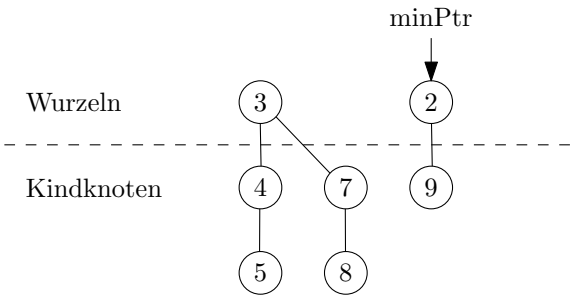
Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [13 Punkte]

a. Gegeben sei der unten abgebildete Pairing Heap. Weiterhin sei h ein Zeiger auf den Knoten mit Schlüssel 6 (markierter Knoten). Zeichnen Sie den finalen Zustand des Pairing Heaps nach Ausführen der beiden Operationen `decreaseKey(h,2)` und `deleteMin()`. [2 Punkte]



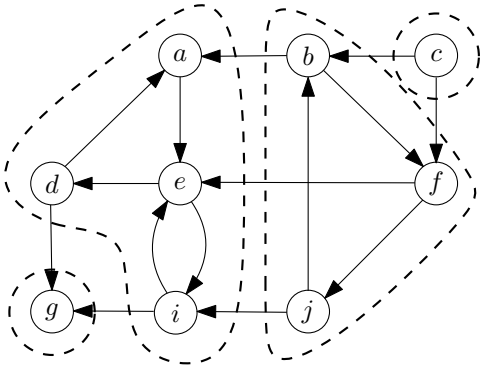
Lösung



Es kann hier mehrere Lösungen geben, abhängig davon, welche beiden Bäume nach dem `deleteMin()` gemerged werden.

b. Markieren Sie im unten abgebildeten Graphen alle starken Zusammenhangskomponenten (SCCs). Kann man durch Hinzufügen einer einzigen Kante erreichen, dass der Graph aus einer einzigen SCC besteht? Falls ja, geben Sie eine solche Kante an. [2 Punkte]

Lösung



Durch Hinzufügen der Kante (g,c) fallen die vier SCCs zu einer einzigen SCC zusammen.

c. Wir definieren das Problem `MaximumAcyclicSubgraph` für einen gerichteten Graphen $G = (V,E)$ mit $V = \{1,\ldots,n\}$wie folgt: Finde eine maximale Teilmenge $E' \subseteq E$, so dass $G' = (V,E')$ keine Kreise enthält. Zeigen Sie, dass der unten abgebildete Algorithmus eine *gültige* Lösung $\tilde{G} = (V,\tilde{E})$ berechnet, wobei für die Kantenmenge $\tilde{E}$ gilt: $\frac{1}{2}OPT \leq |\tilde{E}| \leq OPT$.
Anmerkung: $G = (V,E)$ enthält keine Schleifen der Form $(u,u) \in E$.

Algorithmus 1 `ApproxMaximumAcyclicSubgraph(G = (V, E))`

$E_1 := \{(u,v) \in E \mid u < v\}$
$E_2 := \{(u,v) \in E \mid u > v\}$
if $|E_1| > |E_2|$ **then**
    return $\tilde{G} = (V,E_1)$
else
    return $\tilde{G} = (V,E_2)$

[3 Punkte]

Lösung

G' enthält keine Kreise.

Beweis. Angenommen $|E_1| > |E_2|$ und $G' = (V,E_1)$ enthält einen Kreis $C = (\,(x,v_1),\ \ldots,\ (v_k,y),\ (y,x)\,)$. Für alle Kanten $(u,v) \in E_1$ gilt $u < v$. Da im Kreis C ein Pfad von x nach y existiert folgt, dass $x < y$. Da aber auch $(y,x) \in C$ folgt auch $y > x$, was im Widerspruch zur Definition von E_1 steht. Der Fall $|E_1| < |E_2|$ ist analog. □

Im Folgenden sei die optimale Lösung für das `MaximumAcyclicSubgraph` Problem $OPT = |E|$. Für eine Kante $(u,v) \in E$ gilt entweder $u < v$ oder $u > v$, daher ist $E_1 \cup E_2 = E$ und $E_1 \cap E_2 = \emptyset$. Für die Ausgabe $G' = (V,E')$ gilt: $|E'| = \max(|E_1|,|E_2|)$. Daraus folgt:

$$\frac{1}{2}OPT = \frac{|E|}{2} = \frac{|E_1| + |E_2|}{2} \leq \max(|E_1|,|E_2|) = |E'| \leq |E| = OPT$$

Daraus folgt, dass `ApproxMaximumAcyclicSubgraph(G = (V, E))` ein Approximationsalgorithmus mit Approximationsfaktor $\frac{1}{2}$ ist.

d. Gegeben sei ein Graph mit n Knoten und $m = \Theta(n \log n)$ Kanten ohne negative Zyklen. Die Ausführungszeit eines parallelen Algorithmus zur Berechnung *aller* kürzesten Wege in diesem Graphen auf p Prozessoren sei $\Theta(\frac{n^3 \log p}{p})$.
Geben Sie die Arbeit, sowie den Speedup und die Effizienz gegenüber dem sequentiellen Knotenpotential Algorithmus zur Berechnung aller kürzesten Wege aus der Vorlesung an.

[3 Punkte]

Lösung

- Die Arbeit des Algorithmus ist $W(p) = p \cdot T(p) = \Theta(n^3 \log p)$
- Die Worst-case Laufzeit des sequentiellen Algorithmus für $m = \Theta(n \log n)$ Kanten ist $\Theta(n^2 \log n)$.

  Damit ist der relative Speedup $S(p) = \frac{T_{\text{seq}}}{T_{\text{par}}(p)} = \frac{\Theta(n^2 \log n)}{\Theta(\frac{n^3 \log p}{p})} = \Theta(\frac{p \log n}{n \log p})$.
- Für die Effizienz folgt $E(p) = S(p)/p = \Theta(\frac{\log n}{n \log p})$.

e. Gegeben sei ein Roboter, der sich auf einem Punkt $s \in \mathbb{Z}$ einer Linie befindet. Ziel des Roboters ist es einen weiteren Punkt $t \in \mathbb{Z}$ auf der Linie zu finden, der eine Distanz $n \geq 1$ von s entfernt ist. Hierbei ist n dem Roboter nicht bekannt. Der Roboter ist weiterhin nur in der Lage sich schrittweise in eine der beiden Richtungen (links oder rechts) auf der Linie zu bewegen und festzustellen, ob er sich auf dem Punkt t befindet.
Gegeben sei der folgende Algorithmus, den der Roboter verwendet, um t zu erreichen:

Algorithmus 2 `FindTarget(s, t)` für Startpunkt s und Zielpunkt t

```
dir ← links
i ← 1
while true do
    dist ← 2^i
    for j ← 1; j ≤ dist; j ← j + 1 do
        Gehe einen Schritt in Richtung dir
        if t erreicht then return
    Gehe zu s zurück                          ▷ Benötigt ebenfalls dist Schritte
    if dir = links then
        dir ← rechts
    else
        dir ← links
    i ← i + 1
```

Geben Sie den kompetitiven Factor für den Algorithmus `FindTarget(s, t)` an, wenn ein optimaler Algorithmus n Schritte benötigt. Begründen Sie Ihre Antwort.
Hinweis: Verwenden Sie $\sum_{k=1}^{n} q^k = q\frac{q^n-1}{q-1}$. [3 Punkte]

Lösung

Der Algorithmus verdoppelt in jeder Iteration den überprüften Weg und wechselt dabei jedesmal die Richtung. Somit wird in Iteration i eine Strecke von 2^i überprüft. Nach spätestens $\lfloor \log n \rfloor + 1$ Iterationen ist der von s zurückgelegte Weg also $\geq n$. Dabei ist es jedoch möglich, dass der Roboter sich in dieser Iteration in die falsche Richtung bewegt und in der nächsten Iteration noch die n Schritte bis zu t gehen muss. Da der Roboter zusätzlich bei jeder Iteration, in der t nicht erreicht wird, wieder zu s zurückkehren muss, ergibt sich ein Gesamtweg von

$$2 \cdot \sum_{i=1}^{\lfloor \log n \rfloor + 1} 2^i + n \leq 2 \cdot \sum_{i=1}^{\log n + 1} 2^i + n = 9n - 4 \leq 9n.$$

Somit besitzt der Algorithmus einen kompetitven Faktor von 9 (bzw. $9 - \frac{4}{n}$).

Lösungsvorschlag

Aufgabe 2. FPT Algorithmen: d-Bounded Degree Deletion [10 Punkte]

Gegeben sei ein ungerichteter Graph $G = (V,E)$ ohne Mehrfachkanten und Schlingen mit n Knoten und m Kanten, sowie $k \in \mathbb{N}$. Das Ziel des Problems d-`Bounded-Degree Deletion` ist es zu entscheiden, ob eine Menge S von maximal k Knoten existiert, deren Entfernen aus G den maximalen Grad des Graphen auf $\leq d$ reduziert. Im Folgenden bezeichnen wir eine Probleminstanz (G,k), für die ein solches S existiert, als *gültig*.
Für d-`Bounded-Degree Deletion` lassen sich die folgenden zwei Reduktionsregeln R1 und R2 angeben:

R1: Existiert ein Knoten v mit Grad $\delta(v) > d+k$, dann entferne v aus G und verringere k um eins ($k := k-1$).

R2: Existiert ein Knoten v für den gilt: $\forall w \in N[v] : \delta(w) \leq d$, dann entferne v aus G.
$N[v] = \{u : \{u,v\} \in E\} \cup \{v\}$ ist die geschlossene Nachbarschaft von $v \in V$.

a. Begründen Sie die Korrektheit der Reduktionsregeln R1 und R2. [2 Punkte]

Lösung

- R1: Würde dieser Knoten nicht Teil einer optimalen Lösung sein, müssten mindestens $k+1$ seiner Nachbarn enthalten sein. Dies ist ein Widerspruch zur Voraussetzung, das k Knoten ausreichen. Somit ist ein solcher Knoten immer Teil einer optimalen Lösung und kann daher entfernt und k um eins reduziert werden.
- R2: Keine Lösung minimaler Kardinalität würde einen solchen Knoten enthalten und die Nachbarn des Knoten haben auch nach dessen Entfernen noch einen Grad $\leq d$.

b. Gegeben sei eine Instanz (G,k) für die gilt, dass jeder Knoten $v \in V$ mindestens Grad $d+k+1$ besitzt. Begründen Sie, dass eine solche Instanz (G,k) von d-`Bounded-Degree Deletion` niemals gültig ist. [2 Punkte]

Lösung

Da jeder Knoten $v \in V$ mindestens Grad $d+k+1$ besitzt, folgt zunächst $|V| \geq d+k+2$. Sei nun einer der Knoten $v \in V$ nicht in S enthalten, müssen mehr als k seiner Nachbarn in S enthalten sein, damit v nach Entfernen von S einen Grad $\leq d$ besitzt. Damit wäre jedoch $|S| > k$ und daher (G,k) nicht gültig. Folglich müssen alle $|V| \geq d+k+2$ Knoten in S enthalten sein. Damit ist jedoch ebenfalls $|S| > k$ und daher (G,k) nicht gültig.

c. Zeigen Sie, dass eine gültige Instanz (G,k) von d-`Bounded-Degree Deletion` nicht mehr als $n(k+d)$ Kanten besitzt.
Hinweis: Führen Sie einen Widerspruchsbeweis mit Hilfe von Teilaufgabe **b**.
[3 Punkte]

Lösung

Annahme: G enthält mehr als $n(k+d)$ Kanten.
Beweis Entferne zunächst alle Knoten mit Grad $\leq d+k$. Hierdurch werden maximal $n(d+k)$ Kanten entfernt. Nach der Annahme muss nun ein induzierter Subgraph von G mit minimalem Grad $d+k+1$ übrig bleiben. Nach Teilaufgabe **b** ist eine solche Instanz jedoch niemals gültig. Somit kann auch (G,k) nicht gültig sein im Widerspruch zur Voraussetzung.

d. Geben Sie einen Algorithmus an, der für eine gegebene Instanz (G,k) die Reduktionsregeln R1 und R2 in $\mathcal{O}(n(k+d))$ Zeit vollständig (d.h. so lange bis keine der Regeln mehr möglich ist) anwendet oder ausgibt, dass die Instanz nicht gültig ist. Begründen Sie die Laufzeit Ihres Algorithmus.
Hinweis: Gehen Sie davon aus, dass das Entfernen eines Knoten v in $\mathcal{O}(\delta(v))$ Zeit möglich ist und $\delta(v)$ in konstanter Zeit bestimmt werden kann. [3 Punkte]

Lösung

Zunächst zählen wir die Kanten von G. Sobald mehr als $n(k+d)$ Kanten gezählt werden, kann der Algorithmus nach Teilaufgabe **b** abbrechen und "ungültig" ausgeben. Mittels der Knotengrade wird nun über alle Knoten iteriert und überprüft, ob ein Knoten Grad $> k+d$ besitzt. Entsprechende Knoten bzw. inzidente Kanten werden entfernt und der Grad der Nachbarn dekrementiert. Dies ist linear in der Größe des Graphen und somit in $\mathcal{O}(n(k+d))$ Zeit möglich. Nun wird erneut über alle Knoten iteriert und mittels der gespeicherten Grade überprüft, ob die Bedingung für R2 gegeben ist (auch in Linearzeit möglich). Da eine Anwendung von R2 keine weiteren Knoten mit Grad $> k+d$ erzeugt, terminiert der Algorithmus danach.

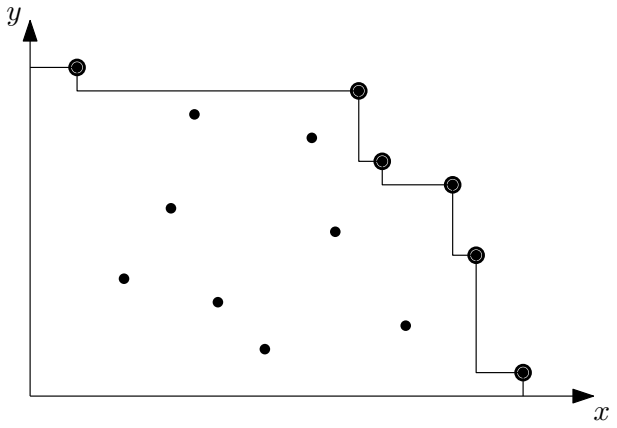
Lösungsvorschlag

Aufgabe 3. Geometrische Algorithmen: Pareto-optimale Punkte [9 Punkte]

Gegeben sei eine Menge von Punkten $S = \{\{x_1, y_1\}, \ldots \{x_n, y_n\}\}$, mit $x_i, y_i \in \mathbb{R}^+$ für $1 \leq i \leq n$. Es seien alle x- sowie y-Koordinaten der Punkte paarweise verschieden. Ein Punkt (x_i, y_i) *dominiert* einen anderen Punkt (x_j, y_j), wenn $x_i > x_j$ und $y_i > y_j$. Im Folgenden suchen wir die Menge P aller Punkte, so dass kein Punkt aus P einem anderen Punkt aus S dominiert wird. Die Punkte aus P heißen *Pareto-optimal*.

a. Markieren sie in der unten abgebildete Menge an Punkten S alle Pareto-optimalen Punkte. [1 Punkte]

Lösung



b. Geben Sie einen Sweepline-Algorithmus an, der alle Pareto-optimalen Punkte in $\mathcal{O}(n \log n)$ Zeit findet. Begründen Sie die Laufzeit Ihres Algorithmus. [3 Punkte]

Lösung

Wir sortieren und durchlaufen alle Punkte nach absteigender x-Koordinate. Die Punkte werden nun in dieser Reihenfolge verarbeitet. Zur Ausgabe der Pareto-optimalen Punkte P speichern wir uns in jedem Schritt den Punkt mit der bisherigen maximalen y-Koordinate. Dieser Punkt ist immer in P enthalten, da sich kein anderer Punkt oberhalb und rechts davon befinden kann. Wird ein neuer Punkt verarbeitet, überprüfen wir, ob er vom aktuell gespeicherten Punkt aus P dominiert wird. Ist dies der Fall, wird der Punkt entfernt bzw. übersprungen. Wird der Punkt nicht vom aktuellen Maximum dominiert, so ist er Teil von P und wird als neues Maximum gespeichert. Das alte Maximum wird ausgegeben.
Analyse: Sortieren nach x-Koordinate ist in $\mathcal{O}(n \log n)$ Zeit möglich. Das Durchlaufen der Punkte benötigt $\mathcal{O}(n)$ Zeit. Das Speichern bzw. Vergleichen mit dem Maximum ist in $\mathcal{O}(1)$ Zeit möglich. Daher ergibt sich eine Gesamtlaufzeit von $\mathcal{O}(n \log n)$.
Alternativ kann auch in die andere Richtung gesweept werden. Hier muss dann allerdings ein Stack zur Verwaltung der Pareto-optimalen Punkte verwendet werden.

c. Geben Sie einen rekursiven Divide-and-Conquer Algorithmus an, der alle Pareto-optimalen Punkte in $\mathcal{O}(n \log n)$ Zeit findet. Begründen Sie die Laufzeit Ihres Algorithmus.
Hinweis: Überlegen Sie sich zunächst, wie Sie das Problem in gleich große Teilprobleme zerlegen können und bestimmen Sie dann einen geeigneten Punkt, den Sie zu P hinzufügen können.
[5 Punkte]

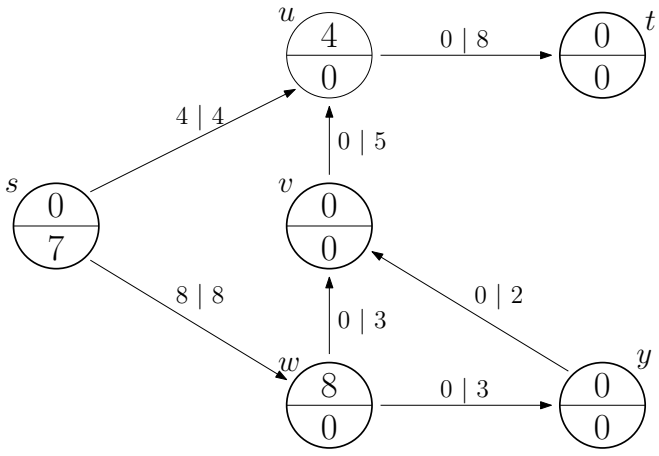
Lösung

Der Algorithmus beginnt mit der Berechnung des Punktes dessen x-Koordinate dem Median über alle x-Koordinaten entspricht. Danach wird die Menge der Punkte anhand des Medians in eine linke Hälfte S_l und rechte Hälfte S_r partitioniert. Sei q der Punkt mit der maximalen y-Koordinate in S_r. Wir stellen fest, das q sowohl eine größere x-Koordinate als alle Punkte in S_l, als auch eine größere y-Koordinate als alle Punkte in S_r besitzt. Somit wird q von keinem anderen Punkt dominiert und wird zur Menge der Pareto-optimalen Punkte P hinzugefügt. Danach entfernen wir alle Punkte in S_l und S_r, die von q dominiert werden, sowie q selbst. Zuletzt werden die Pareto-optimalen Punkte für beiden Hälften S_l und S_r rekursiv bestimmt. Sollte während des Algorithmus eine der Hälften leer sein oder nur aus einem Punkt bestehen wird die Rekursion abgebrochen und der verbleibende Punkt zur Menge der Pareto-optimalen Punkte hinzugefügt.
Analyse: Innerhalb eines Rekursionsaufrufs wird zunächst in $\mathcal{O}(n)$ Zeit der Median der x-Koordinaten bestimmt. Das Partitionieren der Punkte in zwei Hälften, sowie das bestimmen des Punktes mit maximaler y-Koordinate in der rechten Hälfte benötigen ebenfalls $\mathcal{O}(n)$ Zeit. Danach werden alle Punkte aus S_l und S_r in $\mathcal{O}(n)$ Zeit mit q verglichen und ggf. entfernt. Die Laufzeit eines Rekursionsaufrufs ist somit $\mathcal{O}(n)$. Durch jeden Aufruf wird einer bzw. kein Punkt (wenn S leer) zur Menge der Pareto-optimalen Punkte hinzugefügt. Zusätzlich wird die Menge der Punkte für jeden Rekursionsaufruf halbiert. Somit ergibt sich eine Rekursionstiefe von $\mathcal{O}(\log n)$, was zu einer Gesamtlaufzeit von $\mathcal{O}(n \log n)$ führt.

Lösungsvorschlag

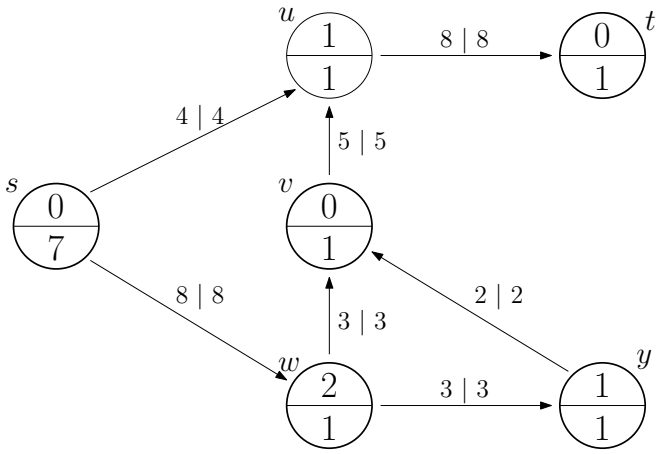
Aufgabe 4. Flussalgorithmen: Partielle Ordnungen [9 Punkte]

a. Gegeben sei das folgende Flussnetzwerk mit Quelle s und Senke t. Es beschreibt den Zustand des preflow-push Algorithmus nach der Initialisierung. Die Knoten sind mit ihrem momentanen Überschuss im oberen und ihrem Distanzlabel im unteren Halbkreis beschriftet. Die Kanten sind jeweils mit ihrem aktuellen Flusswert (links) und Kapazität (rechts) beschriftet. Ihre Aufgabe ist es, mittels des *generic preflow-push* Algorithmus, die Kante (u,t) zu *saturieren*. Geben Sie dafür eine Folge von gültigen `push(a,b)` und `relabel(a)` Operationen an. Für eine `push(a,b)`-Operation geben Sie bitte an, wie viel Fluss Sie über die Kante (a,b) schieben und für eine `relabel(a)`-Operation geben Sie bitte an, auf welchen Wert sich das Distanzlabel von Knoten a ändert.



[3 Punkte]

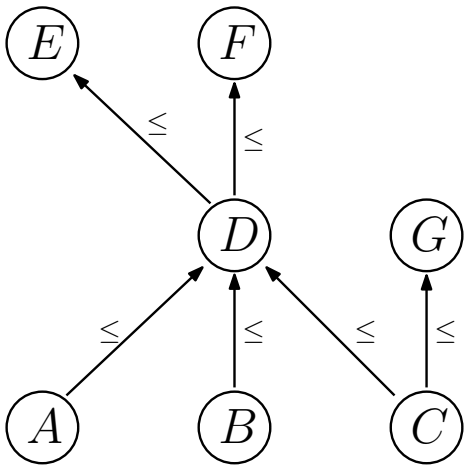
Lösung



1. `relabel(w)` $\rightarrow d(w) = 1$
2. `push(w,y)` $\rightarrow f(w,y) = 3$
3. `push(w,v)` $\rightarrow f(w,v) = 3$
4. `relabel(y)` $\rightarrow d(y) = 1$
5. `push(y,v)` $\rightarrow f(y,v) = 2$
6. `relabel(v)` $\rightarrow d(v) = 1$
7. `push(v,u)` $\rightarrow f(v,u) = 5$
8. `relabel(u)` $\rightarrow d(u) = 1$
9. `push(u,t)` $\rightarrow f(u,x) = 8$

b. Gegeben sei ein gerichteter azyklischer Graph $G = (V,E)$ der eine partielle Ordnung $\leq$ auf der Menge V repräsentiert. Hierfür sei $u \leq v$ $(u,v \in V)$ genau dann, wenn ein Pfad von u nach v in G existiert. Zwei Elemente $u,v \in V$ sind *vergleichbar*, falls entweder $u \leq v$ oder $v \leq u$. Andernfalls sind sie nicht miteinander vergleichbar.
Finden Sie für den unten abgebildeten Graphen G eine maximale Anzahl von vergleichbaren Paaren (x,y) (mit $x \neq y$ und $x \leq y$), sodass jedes Element $v \in V$ maximal einmal als Paar $(v,\cdot)$ und maximal einmal als Paar $(\cdot,v)$ in Ihrer Lösungsmenge vorkommt. [2 Punkte]

Partielle Ordnung für eine Menge $V = \{A,B,C,D,E,F,G\}$. A und F sind vergleichbar ($A \leq F$), jedoch nicht A und C.



Lösung

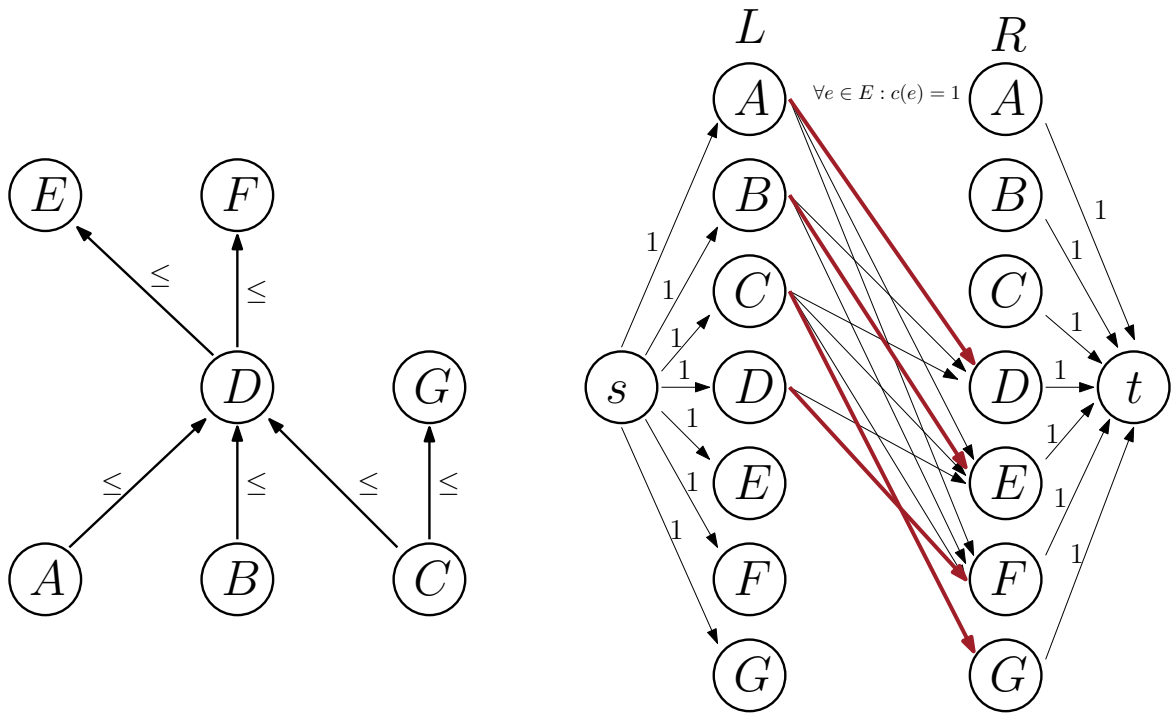
Es gibt ingesamt vier gültige Lösungen:

1. $T = \{\ (A,D),\ (B,E),\ (C,G),\ (D,F)\ \}$
2. $T = \{\ (A,D),\ (B,F),\ (C,G),\ (D,E)\ \}$
3. $T = \{\ (A,E),\ (B,D),\ (C,G),\ (D,F)\ \}$
4. $T = \{\ (A,F),\ (B,D),\ (C,G),\ (D,E)\ \}$

c. Geben Sie einen Algorithmus an, der mit Hilfe eines maximalen Fluss in einem bipartiten Graphen das Problem aus Teilaufgabe **b** für beliebige gerichtete azyklische Graphen $G = (V, E)$ löst. Begründen Sie, warum Ihre Lösung maximal ist.

[4 Punkte]

Lösung



Wir konstruieren ein Flussnetwerk $G' = (V', E', c, s, t)$ wie folgt:

1. $V' = L \cup R \cup \{s, t\}$ mit $L = R = V$
2. $E' = \{(u, v) \text{ mit } u \in L \text{ und } v \in R \mid \forall u, v \in V : u \leq v\} \cup \{(s, u) \mid u \in L\} \cup \{(v, t) \mid v \in R\}$
3. Für alle $(u, v) \in E'$ mit $u \in L$ und $v \in R$ ist die Kapazität $c(u, v) = 1$
4. Für alle $(s, u) \in E'$ mit $u \in L$ ist die Kapazität $c(s, u) = 1$
5. Für alle $(v, t) \in E'$ mit $v \in R$ ist die Kapazität $c(v, t) = 1$

G' ist ein bipartiter Graph. Ein maximaler (s, t)-Fluss induziert ein maximales bipartites Matching M in G', sodass jedes Element in L und R maximal einmal als Paar vorkommt. Da M maximal ist entspricht M einer maximalen Lösung für Teilaufgabe **b**.

Lösungsvorschlag

Aufgabe 5. Randomisierte Algorithmen: Multi-Hashing [8 Punkte]

Im Folgenden wollen wir mit Hilfe eines Arrays t der Größe m und einer endlichen Menge H von unabhängigen Hashfunktionen der Form $h : U \to \{1,\ldots,m\}$ (für alle $h \in H$) eine Hashtabelle konstruieren. Die Hashtabelle soll die Operationen `Insert` und `Contains` unterstützen. Die `Insert`-Operation sei durch folgenden Pseudocode gegeben:

Algorithmus 3 `Insert(u)` für ein $u \in U$

```
for all h ∈ H do
    if t[h(u)] = ⊥ then                    ▷ ⊥ = Leeres Element
        t[h(u)] ← u
        return true                        ▷ Element u wurde erfolgreich eingefügt
return false                               ▷ Element u wurde nicht eingefügt
```

Gehen Sie davon aus, dass die Größe der Menge H hinreichend groß gewählt ist, so dass für alle `Insert(u)` Operationen eine Hashfunktion $h \in H$ existiert mit $t[h(u)] = \perp$, falls ein $i \in \{1,\ldots,m\}$ existiert mit $t[i] = \perp$. Des Weiteren seien alle Einträge von t mit dem leeren Element $\perp$ initialisiert.

a. Zeigen Sie, dass $\mathcal{O}(\frac{m}{m-n})$ eine obere Schranke für die erwartete Laufzeit von `Insert(u)` ist, wobei n die Anzahl der zuvor eingefügten Elemente ist (mit $n < m$). Nehmen Sie dabei an, dass für alle $h \in H$ gilt: $\forall x,y \in U : P(h(x) = h(y)) = \frac{1}{m}$.
Hinweis: Sie benötigen entweder $\sum_{k=0}^{\infty} k \cdot q^{k-1} = \frac{1}{(1-q)^2}$ oder $\sum_{k=0}^{\infty} q^i = \frac{1}{1-q}$ [3 Punkte]

Lösung

Lösung 1:
Die Wahrscheinlichkeit für eine Hash-Kollision einer Hashfunktion $h \in H$ in einer Hashtabelle der Größe m mit n belegten Plätzen ist $p = \frac{n}{m}$. Sei X eine Zufallsvariable die die Anzahl an benutzten Hashfunktionen angibt bis zum ersten Mal ein freier Platz während der `Insert` Operation gefunden wird, dann gilt $P(X = i) = p^{i-1}(1-p)$. Für die erwartete Laufzeit ergibt sich dann

$$E(X) = \sum_{k=0}^{\infty} k \cdot P(X = k) = (1-p) \cdot \sum_{k=0}^{\infty} k \cdot p^{k-1} = \frac{1}{1-p} = \mathcal{O}(\frac{m}{m-n})$$

Lösung 2:
Die Wahrscheinlichkeit für eine Hash-Kollision einer Hashfunktion $h \in H$ in einer Hashtabelle der Größe m mit n belegten Plätzen ist $p = \frac{n}{m}$. Sei X_i eine Zufallsvariable die angibt, ob die i-te Hashfunktion benutzt wurde. Die Wahrscheinlichkeit von $P(X_i = 1)$ ist $\left(\frac{n}{m}\right)^i$. Die Anzahl an benutzten Hashfunktion für eine `Insert(u)` Operation wird durch die Zufallsvariable $X = X_0 + X_1 + \ldots$ bestimmt. Für die erwartete Laufzeit ergibt sich dann

$$E(X) = \sum_{i=0}^{\infty} E(X_i) = \sum_{i=0}^{\infty} \left(\frac{n}{m}\right)^i = \mathcal{O}(\frac{m}{m-n})$$

b. Geben Sie einen Pseudocode für die Operation `Contains(u)` an. Die Funktion soll `true` zurückgeben, falls $u \in U$ zuvor in die Hashtabelle eingefügt worden ist und anderfalls `false`. [1 Punkt]

Lösung

Algorithmus 4 `Contains(u)` für ein $u \in U$

```
for all h ∈ H do
    if t[h(u)] = u then
        return true
    else if t[h(u)] = ⊥ then
        return false
return false
```

c. Geben Sie die erwartete Laufzeit für `Contains(u)` aus Teilaufgabe **b** an, falls nach dem Element gesucht wird, dass durch die i-te `Insert(u)` Operation eingefügt worden ist. [1 Punkt]

Lösung

Die Laufzeit für das Suchen nach dem Element, das als i-tes eingefügt worden ist, ist gleich seiner erwarteten Laufzeit für `Insert(u)` $\Rightarrow \mathcal{O}(\frac{m}{m-i})$

d. Gegeben Sei ein zufällig ausgewähltes Element u das zuvor in die Hash-Tabelle eingefügt worden ist. Zeigen Sie, dass $\mathcal{O}((H_m - H_{m-n})\frac{m}{n})$ eine obere Schranke für die erwartete Laufzeit von `Contains(u)` ist.
Hinweis: $H_n = \sum_{i=1}^{n} \frac{1}{i}$ [3 Punkte]

Lösung

Sei X eine Zufallsvariable die die Anzahl an benutzten Hashfunktionen angibt bis das gesuchte Element $u \in U$ gefunden wird. Die erwartete Laufzeit um u zu finden, wird durch seine erwartete Laufzeit zum Einfügezeitpunkt festgelegt (siehe Teilaufgabe **c**). Die erwartete Laufzeit, um ein beliebiges Element zu finden ist also der Durchschnitt über die Summe aller erwarteten Einfügelaufzeiten:

$$E(X) = \frac{1}{n}\sum_{k=0}^{n-1} T_{\texttt{Insert}}(k) = \frac{1}{n}(\frac{m}{m} + \frac{m}{m-1} + \ldots + \frac{m}{m-(n-1)})$$
$$= \frac{m}{n}(\frac{1}{m-n+1} + \frac{1}{m-n+2} + \ldots + \frac{1}{m}) = \frac{m}{n}(H_m - H_{m-n})$$

Lösungsvorschlag

Aufgabe 6. Stringalgorithmen: LCP-Interval Bäume [11 Punkte]

a. Geben Sie für den Text „*babbaba*\$“ das Suffix-Array und inverse Suffix-Array, sowie den LCP-Array an.

[4 Punkte]

Lösung

Indices	0	1	2	3	4	5	6	7
Text	b	a	b	b	a	b	a	\$
Suffix-Array	7	6	4	1	5	3	0	2
Inverse Suffix-Array	6	3	7	5	2	4	1	0
LCP-Array	-1	0	1	2	0	2	3	1

Im Folgenden sei ein Intervall $[i, j]$ eines LCP-Arrays ein LCP-Intervall mit Wert ℓ, wenn es folgende Eigenschaften erfüllt:

1. $LCP[i] < \ell$
2. $LCP[k] \geq \ell$ für alle k mit $i+1 \leq k \leq j$
3. $LCP[k] = \ell$ für mindestens ein k mit $i+1 \leq k \leq j$
4. $LCP[j+1] < \ell$

Für einen LCP-Intervall $[i, j]$ mit Wert ℓ schreiben wir $([i, j],\ \ell)$. Ein LCP-Intervall $L_1 = ([i, j],\ \ell)$ *umschließt* ein anderes LCP-Intervall $L_2 = ([p,q],\ m)$, falls $i \leq p < q \leq j$ und $m > \ell$ (*Notation:* $L_1 \prec L_2$). Ein LCP-Intervall L_2 ist ein Kind von L_1, falls $L_1 \prec L_2$ und es kein anderes LCP-Intervall L_3 gibt, für das gilt $L_1 \prec L_3 \prec L_2$. Wir nennen den darüber definierten Baum einen *LCP-Intervall Baum*.

Beispiel: Das Intervall $[1,3]$ aus Teilaufgabe **a** ist ein LCP-Intervall mit Wert 1, da

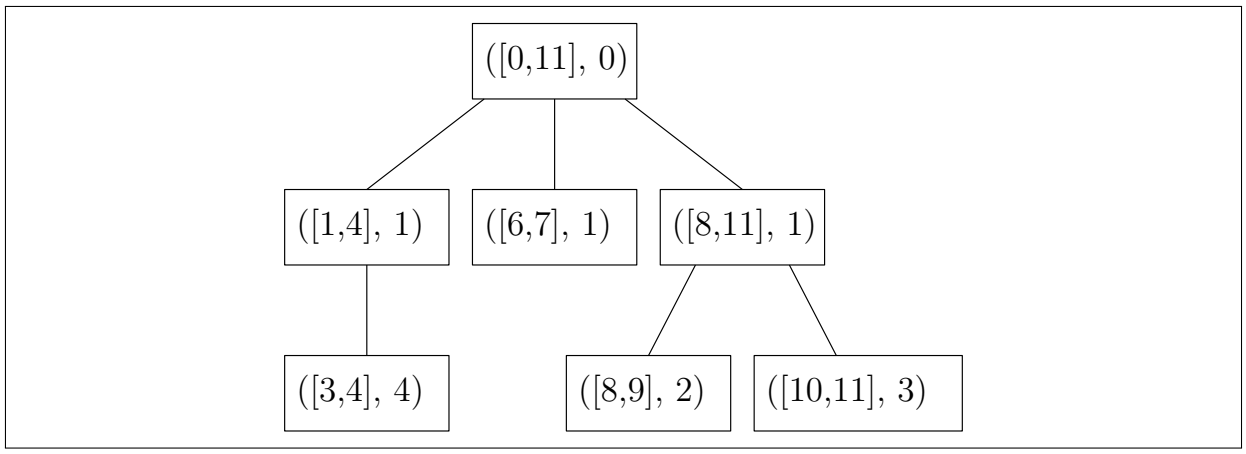
1. $LCP[1] = 0 \Rightarrow LCP[1] < 1$
2. $LCP[k] \geq 1$ für alle k mit $2 \leq k \leq 3$
3. $LCP[2] = 1$
4. $LCP[4] = 0 \Rightarrow LCP[4] < 1$

b. Vervollständigen Sie für den Text „*mississippi*\$“ den *LCP-Intervall Baum*. [4 Punkte]

LCP-Array des Text „mississippi\$“

Indices	0	1	2	3	4	5	6	7	8	9	10	11	
LCP-Array	-1	0	1	1	4	0	0	1	0	2	1	3	-1

Lösung



c. Vervollständigen Sie den folgenden Algorithmus zu einem Linearzeitalgorithmus zur Berechnung aller LCP-Intervalle eines beliebigen LCP-Array. [3 Punkte]

Lösung

Algorithmus 5 `FindLCPIntervals(`LCP`)` mit $LCP[0] = -1$ und $LCP[n] = -1$ ($|T| = n$)

$L \leftarrow \emptyset$ ▷ Soll am Ende alle LCP-Intervalle enthalten
s.push($(0,\ 0)$) ▷ $(lcp,\ start)$ $\Rightarrow$ LCP-Wert und Startposition des LCP-Intervals
▷ s ist ein Stack

for $k = 1, \ldots, n$ **do**
  $i \leftarrow k - 1$
  while $LCP[k] < s$.top().lcp **do**
    [lcp] $\leftarrow$ s.top().lcp
    [i] $\leftarrow$ s.top().start
    s.pop()
    $L \leftarrow L \cup \{$ ([[i] , [k-1]], [lcp]) $\}$
  if $LCP[k] > s$.top().lcp **then**
    s.push(([LCP[k]] , [i]))
return L