Klausur Algorithmen II am 12.09.2017

Aufgabe	Erstkorrektur	Zweitkorrektur
1 a-d	Y. Akhremtsev	S. Witt
1 e-f	D. Hespe	S. Witt
2 a und b	Y. Akhremtsev	S. Witt
2 c	J. Speck	S. Witt
3	J. Speck	M. Axtmann
4	D. Hespe	M. Axtmann
5	M. Axtmann	D. Funke
6	M. Axtmann	D. Hespe

Es zählt grundsätzlich die Zweitkorrektur.

Notenschema

Note	ab Punktzahl
1	**38**
1,3	**36**
1,7	**34**
2	**32**
2,3	**30**
2,7	**28**
3	**26**
3,3	**24**
3,7	**22**
4	**20**

Name:

Vorname:

Matrikelnummer:

Klausur-ID:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders 12.09.2017

Klausur Algorithmen II

Aufgabe 1.	Kleinaufgaben	13 Punkte
Aufgabe 2.	Flussalgorithmen	10 Punkte
Aufgabe 3.	Make und Makefiles	8 Punkte
Aufgabe 4.	Approximationsalgorithmen: Konjunktive Normalform	8 Punkte
Aufgabe 5.	Geometrische Algorithmen: Schnitte und Polygone	11 Punkte
Aufgabe 6.	Stringalgorithmen: Textrekonstruktion	10 Punkte

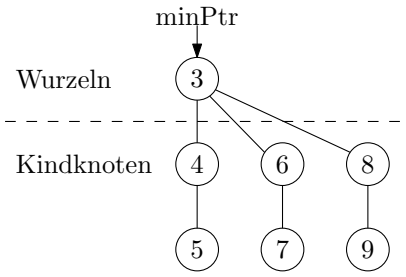
Bitte beachten Sie:

- 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 **18 Blätter**.
- Zum Bestehen der Klausur sind 20 Punkte hinreichend.

Lösungsvorschlag

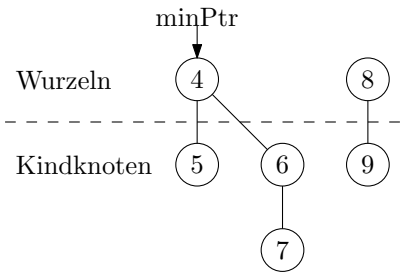
Aufgabe 1. Kleinaufgaben [13 Punkte]

a. Gegeben sei ein Pairing Heap mit unten eingezeichnetem Zustand. Führen Sie auf diesem Pairing Heap die Operation `deleteMin()` aus und zeichnen Sie den Zustand des Pairing Heaps nach dieser Operation.

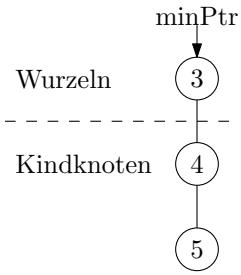


[1 Punkte]

Lösung



b. Gegeben sei ein leerer Pairing Heap. Geben Sie eine Folge von Operationen an, die daraus den unten eingezeichneten Zustand erzeugt.

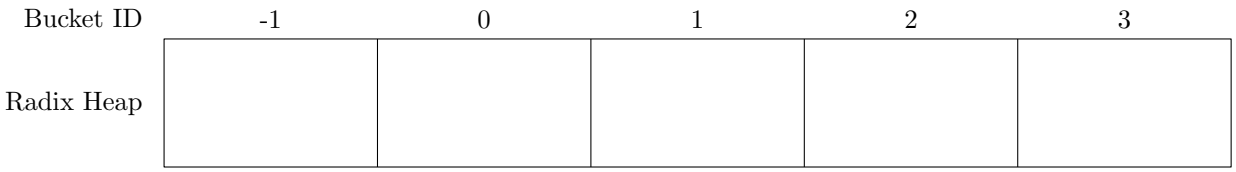


[2 Punkte]

Lösung

```
insert(1), insert(2), insert(3), insert(4), insert(5), deleteMin(),
deleteMin()
```

c. Gegeben sei ein Radix Heap. Im Moment ist 1010_b das kleinste gespeicherte Element und es können Elemente im Bereich $1010_b - 10000_b$ gespeichert werden ($C = 6$). Geben Sie für alle erlaubten Elemente an, in welchem Bucket diese gespeichert werden, solange keine `deleteMin()` Operation ausgeführt wird.



[2 Punkte]

Lösung

Bucket ID	-1	0	1	2	3
Bucket Heap	1010_b	1011_b		1100_b 1101_b 1110_b 1111_b	10000_b

d. Gegeben sei ein Graph mit n Knoten und $m = \Theta(n \log n)$ Kanten. Die Ausführungszeit eines parallelen Algorithmus zur parallelen Suche kürzester Wege in diesem Graph auf p Prozessoren sei

$$\Theta\left(\frac{n^{\frac{3}{2}} \log n}{p}\right).$$

Geben Sie den Speedup $S(p)$ (Beschleunigung) und die Effizienz $E(p)$ gegenüber dem sequentiellen Algorithmus von Dijkstra mit Fibonacci Heaps an. Wie muss die Eingabegröße mit der Prozessorzahl asymptotisch steigen, damit der Speedup $S(p) = s$ erreicht wird? Nehmen Sie immer den schlimmsten Fall an und rechnen Sie im Θ-Kalkül.

[3 Punkte]

Lösung

Der relative *Speedup* eines parallelen Algorithmus T_{par} ausgeführt auf p Prozessoren gegenüber einem sequentiellen Algorithmus T_{seq} ist definiert als

$$S(p) = \frac{T_{\text{seq}}}{T_{\text{par}}(p)}.$$

Die (relative) *Effizienz* ist definiert als

$$E(p) = \frac{S(p)}{p} = \frac{T_{\text{seq}}}{pT_{\text{par}}(p)}.$$

Die Worst-Case Laufzeit von Dijkstra's Algorithmus für $m = \Theta(n \log n)$ Kanten ist in $\Theta(n \log n)$. Damit ist der relative Speedup

$$S(p) = \frac{\Theta(n \log n)}{\Theta\left(\frac{n^{\frac{3}{2}} \log n}{p}\right)} = \Theta\left(\frac{p}{\sqrt{n}}\right)$$

und die Effizienz

$$E(p) = \frac{\Theta\left(\frac{p}{\sqrt{n}}\right)}{p} = \Theta\left(\frac{1}{\sqrt{n}}\right).$$

Aus Speedup s

$$S(p) = \Theta\left(\frac{p}{\sqrt{n}}\right) \stackrel{!}{=} s$$

folgt

$$n = \Theta\left(\left(\frac{p}{s}\right)^2\right).$$

e. Sei $f(n,k)$ die Laufzeit eines Algorithmus mit Eingabegröße n. Die Laufzeit hängt weiter von einem Parameter k ab. Geben Sie an, welche der folgenden Laufzeiten ein Problem *fixed-parameter-tractable* (FPT) machen. Begründen Sie Ihre Antwort jeweils kurz.

[3 Punkte]

Lösung

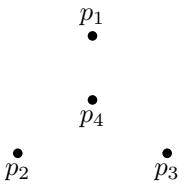
Ein Problem ist FPT, falls ein Algorithmus das Problem in $\mathcal{O}(f(k) \cdot p(n))$ löst, wobei f eine berechenbare Funktion, k der Parameter, p ein beliebiges Polynom und n die Eingabelänge ist.

1. $f_1(n,k) = n^{2.718} \log_{1+\frac{1}{k}}(n)$:
   Das beschriebene Problem ist fixed-parameter-tractable. $f_1(n,k) = n^{2.718} \log_{1+\frac{1}{k}}(n) = n^{2.718} \log_2(n) \cdot \frac{1}{\log_2(1+\frac{1}{k})}$ ist polynomiell in n ($n^{2.718} \log_2(n)$) und davon unabhängig ist $\frac{1}{\log_2(1+\frac{1}{k})}$ nur abhängig von k.

2. $f_2(n,k) = 2^n \cdot n^{\frac{1}{4}}$:
   Das Problem ist nicht fixed-parameter-tractable. Die Laufzeit hängt zwar nicht vom Parameter k ab, die Laufzeit $2^n \cdot n^{\frac{1}{4}}$ ist nicht polynomiell in n.

3. $f_3(n,k) = n^4 \cdot n^k$:
   Das Problem ist nicht fixed-parameter-tractable. $f_3(n,k)$ ist durch $n^4 n^k$ zwar polynomiell in n, aber dies ist nicht unabhängig von k.

f. Gegeben sei folgende Punktmenge $P := \{p_1, p_2, p_3, p_4\}, p_i \in \mathbb{R} \times \mathbb{R}$:



Für diese Punktmenge soll die kleinste einschließende Kugel berechnet werden. Der Algorithmus aus der Vorlesung hat schon für die Punkte $\{p_1, p_2, p_3\}$ die kleinste einschließende Kugel berechnet. Welche Schritte führt der Algorithmus noch aus, um die kleinste einschließende Kugel für P zu berechnen? Wo müsste der Punkt p_4 liegen, sodass der Algorithmus mehr Berechnungen ausführen würde?

[2 Punkte]

Lösung

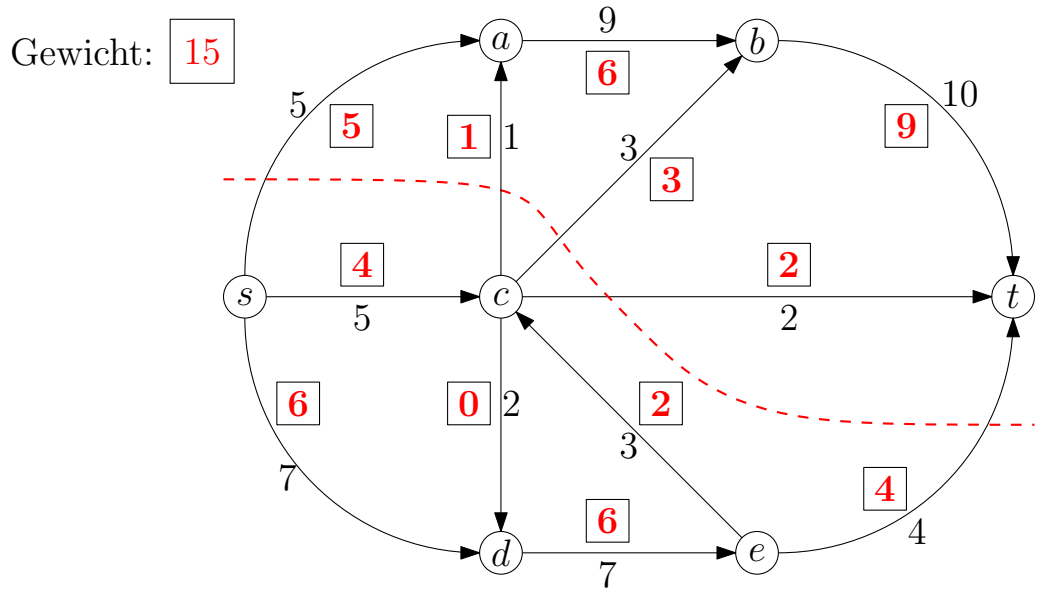
Bisher wurde die kleinste einschließende Kugel K für die Punkte $\{p_1, p_2, p_3\}$ berechnet. Nun prüft der Algorithmus, ob p_4 in K liegt. Da diese Überprüfung erfolgreich ist, gibt der Algorithmus K zurück und terminiert.
Würde p_4 außerhalb von K liegen, so würde p_4 auf der einschließenden Kugel liegen. Unter dieser Voraussetzung würde der Algorithmus rekursiv eine neue kleinste einschließende Kugel K' berechnen.

Lösungsvorschlag

Aufgabe 2. Flussalgorithmen [10 Punkte]

a. Betrachten Sie das unten stehende Flussnetzwerk. Jede Kante ist mit ihrer Kapazität beschriftet. Geben Sie einen maximalen *st*-Fluss an, indem Sie die Flusswerte in die Kästchen schreiben. Geben Sie den Wert des Flusses an **und** zeigen Sie dass der Fluss maximal ist.
[3 Punkte]

Lösung



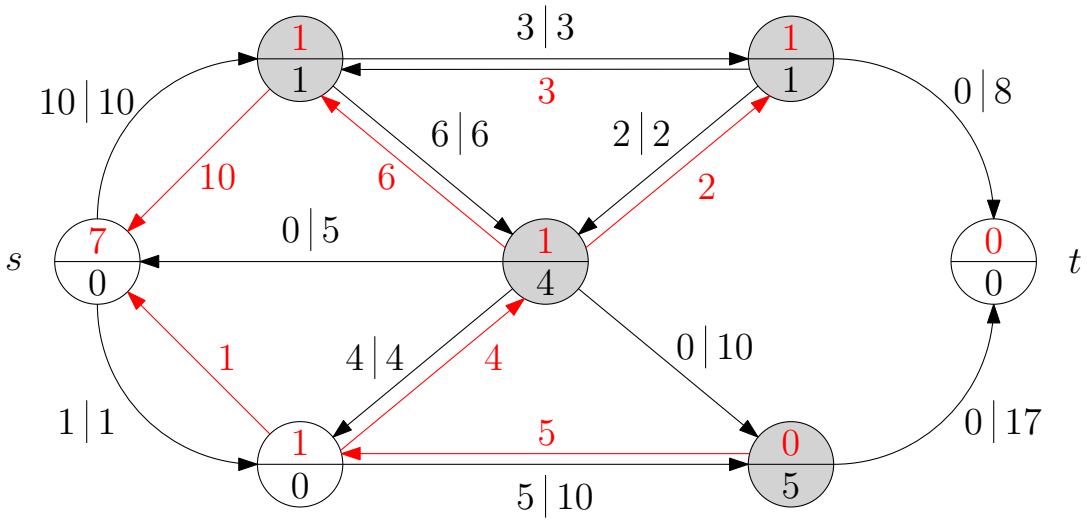
Der *st*-Fluss ist oben (fett) eingetragen. Der Fluss hat den Wert 15. Dieser Fluss ist nach dem Max-Flow-Min-Cut-Theorem maximal, da es einen Schnitt mit Gewicht 15 gibt (siehe gestrichelte Linie in der Zeichnung).

b. Gegeben sei der folgende unvollständige Flussgraph mit Quelle s und Senke t. Er beschreibt einen Zwischenzustand des *preflow-push* Algorithmus. Die Knoten sind mit ihrem momentanen Überschuss im unteren Halbkreis beschriftet, die Kanten sind mit ihrem aktuellen Fluss (links) und ihrer Kapazität (rechts) beschriftet.

Weisen Sie allen Knoten gültige Level zu, markieren Sie die aktiven Knoten und zeichnen Sie die Residualkanten mit deren Kapazitäten ein.

[3 Punkte]

Lösung



Die Knoten sind mit ihrem Level im oberen Halbkreis rot beschriftet und die aktiven Knoten sind grau hinterlegt. Residualkanten sind als rote Rückwärtskanten eingezeichnet und mit ihrer Kapazität beschriftet.

c. Gegeben sei ein Flussnetzwerk $N = (E, V, s, t, c)$ mit einem Kreis K. Beweisen Sie, dass es mindestens einen maximalen st-Fluss $f : E \to \mathbb{R}$ in N gibt mit einer Kante e auf dem Kreis K für die $f(e) = 0$ gilt. Beachten Sie, dass Sie folgende Eigenschaften für f zeigen müssen:

1. Der Fluss über jede Kante ist zulässig.
2. Der eingehende Fluss von jedem Knoten ist gleich seinem ausgehenden Fluss.
3. Es gibt keinen Fluss mit einem höheren Flusswert.

[4 Punkte]

Lösung

Gegeben sei ein maximaler Fluss f auf dem Flussnetzwerk N. Gilt $f(e) = 0$ für eine der Kanten e auf dem Kreis K, so erfüllt f schon die gestellte Anforderung.
Sei e' die Kante im Kreis K mit dem geringsten Fluss in f. Wir definieren uns auf Basis des Flusses f ein neue Funktion f', indem wir den Fluss über die Kanten in K um jeweils $f(e')$ verringern. Offensichtlich gilt nun $f'(e') = 0$.

f' ist ein st-Fluss:

- Der eingehende Fluss $\sum_{(v,\cdot)\in E} c(v,u)$ und der ausgehende Fluss $\sum_{(\cdot,v)\in E} c(u,v)$ für alle Knoten außerhalb vom Kreis K bleibt gleich. Da jeder Knoten v auf dem Kreis K genau eine eingehende Kreiskante und eine ausgehende Kreiskante hat, reduziert sich sowohl der eingehende Fluss von v als auch der ausgehende Fluss von v um genau $f(e')$. Da in f der eingehende Fluss gleich dem ausgehenden Fluss von v ist, ist somit auch in f' der eingehende Fluss gleich dem ausgehenden Fluss von v. Der Fluss über jede Kante im Kreis ist nicht negativ, da wir den Fluss über diese Kanten nur um den Fluss der leichtesten Kante im Kreis verringert haben. Da wir den Fluss über die Kanten nicht erhöhen, überschreitet der Fluss f', genauso wie der Fluss f, die Kapazität keiner Kante.

Der Fluss f' ist maximal:

- Da s keine Eingehenden Kanten hat, gilt $u \neq s$ und $v \neq s$. Somit ändert sich der ausgehende Fluss von s nicht. Da der Fluss f maximal ist, ist auch der Fluss f' ebenfalls maximal.

Lösungsvorschlag

Aufgabe 3. Make und Makefiles [8 Punkte]

Das Tool *make* ist ein Build-Management-Tool, um Dateien mithilfe anderer Dateien zu erzeugen. Die Erzeugungsvorschrift `ErzeugeA` beschreibt, wie eine Datei `A` aus einer Liste vorhandener Dateien $\{A_1,\ldots,A_n\}$ erzeugt wird. Um eine Datei *A* mit einer Erzeugungsvorschrift erfolgreich erzeugen zu können, müssen alle Dateien $A_i \in \{A_1,\ldots,A_n\}$ schon existieren, oder durch eine eigene Erzeugungsvorschrift erfolgreich erzeugt worden sein. Im folgenden Beispiel benötigt die Datei `A` die Dateien `B` und `C`; die Datei `B` benötigt die Dateien `C` und `D`; die Datei `C` benötigt die Dateien `E`. Beachten Sie, dass es für die Dateien `D` und `E` keine Erzeugungsvorschrift gibt, diese Dateien also schon existieren müssen. Nehmen Sie in allen Teilaufgaben an, dass alle Dateien ohne Erzeugungsvorschrift schon existieren.

```
A : B C
  ErzeugeA
B : C D
  ErzeugeB
C : E
  ErzeugeC
```

a. Gegeben sei folgendes Makefile:

```
A : B
  ErzeugeA
B : C
  ErzeugeB
C : D E
  ErzeugeC
D : F
  ErzeugeD
E : A
  ErzeugeE
```

Können alle Dateien des Makefiles erfolgreich erzeugt werden? Begründen Sie Ihre Antwort.
[2 Punkte]

Lösung

Nein, das Makefile kann nicht erfolgreich ausgeführt werden. `A` kann beispielsweise nicht erzeugt werden, da es über `B` und `C` von `E` abhängt, Ziel `E` aber wiederum von Ziel `A` abhängt. Es existiert also eine zyklische Abhängigkeit.

b. Gegeben sei ein Makefile, das erfolgreich ausgeführt werden kann. Geben Sie einen Algorithmus an, welcher die Erzeugungsreihenfolge der Ziele des Makefiles berechnet, sodass die Ausführung des Makefiles erfolgreich ist. Die Laufzeit von Ihrem Algorithmus soll asymptotisch linear mit der Anzahl an eingegebenen Dateien und der Anzahl an Abhängigkeiten steigen.

[2 Punkte]

Lösung

Erzeuge aus dem Makefile einen Abhängigkeitsgraph. Jedes Ziel ist ein Knoten. Für jede Quelle q eines Ziels z existiert eine Kante vom Quellknoten v_q zum Zielknoten v_z. Berechne mittels Tiefensuche die *finish-time* (bzw. topologische Sortierung). Die Ziele müssen in aufsteigender finish-time (bzw. absteigender topologischer Sortierung) ausgeführt werden.

c. Geben Sie einen Algorithmus an, welcher die Korrektheit eines Makefiles prüft. Die Laufzeit von Ihrem Algorithmus soll asymptotisch linear mit der Anzahl an eingegebenen Dateien und der Anzahl an Abhängigkeiten steigen.

[1 Punkte]

Lösung

Berechne den Abhängigkeitsgraph aus der Lösung der letzten Teilaufgabe. Traversiere diesen Abhängigkeitsgraph mittels Tiefensuche. Falls eine Rückwärtskante gefunden wird, so gebe “False” zurück. Wird keine Rückwärtskante gefunden, so wird “True” zurückgegeben.
Alternative Lösung: Berechne alle SCCs auf dem Abhängigkeitsgraph. Falls es nur triviale SCCs gibt (SCC besteht nur aus einem Knoten), so gebe “True” zurück. Sonst gebe “False” zurück.

d. Gegeben sei ein Makefile, für den nicht notwendigerweise alle Dateien erzeugt werden können. Geben Sie einen Algorithmus an, welcher alle Paare A und B von Dateinamen ausgibt, die jeweils direkt oder indirekt gegenseitig voneinander abhängen. Die Laufzeit von Ihrem Algorithmus soll asymptotisch linear mit der Anzahl an eingegebenen Dateien, der Anzahl an ausgegebenen Paaren und der Anzahl an Abhängigkeiten steigen. Beispiel: Der Algorithmus soll für das unten angegebene Makefile die Liste $((B,C),(B,D),(C,D))$ ausgeben.

```
A : B
  ErzeugeA
B : C
  ErzeugeB
C : D
  ErzeugeC
D : B
  ErzeugeD
```

[3 Punkte]

Lösung

Berechne den Abhängigkeitsgraph aus der Lösung der vorletzten Teilaufgabe. Berechne auf diesem Graph alle starken Zusammenhangskomponenten. Berechne für jede nicht-triviale SCC (SCC bestehend aus mehr als einem Knoten) alle Paare von Dateien der SCC und gebe diese jeweils aus.

Lösungsvorschlag

Aufgabe 4. Approximationsalgorithmen: Konjunktive Normalform [8 Punkte]

Gegeben sei eine konjunktive Normalform (KNF) K mit n Klauseln und drei Literalen in jeder Klausel. Die konjunktive Normalform enthält in dieser Aufgabe keine Negation. Beispiel:

$$K = (l_1 \vee l_2 \vee l_3) \wedge (l_4 \vee l_5 \vee l_3) \wedge (l_5 \vee l_6 \vee l_7)$$

Ein **Lösungsset** S einer KNF K ist ein (Teil-)Set von Literalen aus K für die gilt: Falls alle Literale aus S wahr sind, so ist K wahr. Ein Lösungsset einer KNF K ist minimal, falls es kein kleineres Lösungsset von K gibt.

a. Geben Sie ein minimales Lösungsset für die oben angegebene KNF K an.

[1 Punkte]

Lösung

Es gibt kein Literal, welche im allen drei Klauseln vorkommt. Die KNF K ist aber wahr, falls l_3 und l_5 wahr sind. Somit ist $\{l_3, l_5\}$ ein minimales Lösungsset.

b. Der unten angegebene Algorithmus berechnet für eine KNF mit n Klauseln und c verschiedenen Literalen aus der Menge $\{1, \ldots, c\}$ ein Lösungsset in $\mathcal{O}(n)$ Zeit. Ist das Ergebnis eine 4-Approximation eines minimalen Lösungssets? Beweisen Sie Ihre Antwort.

Algorithm 1 K: `Liste von Klauseln;` c: `Anzahl an Literalen`

```
X[1...c] ← false
for all Klauseln k ∈ K do    ▷ Klausel ist durch ein Array mit drei Ganzzahlen repräsentiert
    if not X[k[1]] and not X[k[2]] and not X[k[3]] then
        X[k[1]] ← true
R : Liste von Literalen
for i in 1,...,c do
    if X[i] == true then
        R.add(i)
return R
```

[2 Punkte]

Lösung

Der angegebene Algorithmus berechnet keine 4-Approximation. Gegeben sei die KNF

$$K = (l_1 \vee l_2 \vee l_{11}) \wedge (l_3 \vee l_4 \vee l_{11}) \wedge (l_5 \vee l_6 \vee l_{11}) \wedge (l_7 \vee l_8 \vee l_{11}) \wedge (l_9 \vee l_{10} \vee l_{11}).$$

Das minimale Lösungsset $\{l_{11}\}$ besteht aus einem Literal. Der Algorithmus berechnet für die KNF das Lösungsset $\{l_1, l_3, l_5, l_7, l_9\}$, welches jedoch aus fünf Literalen besteht und somit eine 5-Approximation ist.

c. Geben Sie einen Algorithmus an, welcher in $\mathcal{O}(n)$ Zeit eine 3-Approximation eines minimalen Lösungssets berechnet. Beweisen Sie, dass Ihr Algorithmus eine 3-Approximation berechnet.
Hinweis: Sie müssen den Algorithmus aus der letzten Teilaufgabe nur geringfügig ändern.

[5 Punkte]

Lösung

Algorithm 2 K: `Liste von Klauseln;` c: `Anzahl an Literalen`

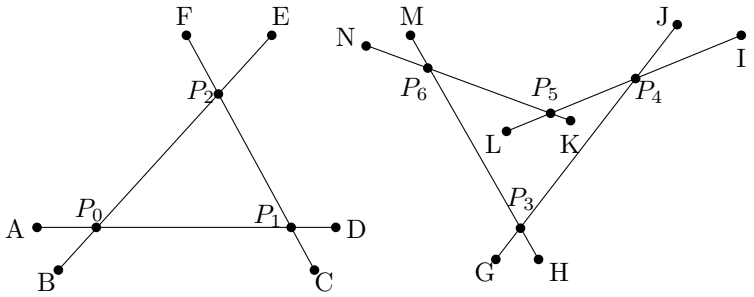
```
1:  X[1...c] ← false
2:  for all Klauseln k ∈ K do   ▷ Klausel ist durch ein Array mit drei Ganzzahlen repräsentiert
3:      if not X[k[1]] and not X[k[2]] and not X[k[3]] then
4:          X[k[1]] ← true
5:          X[k[2]] ← true
6:          X[k[3]] ← true
7:  R : Liste von Literalen
8:  for i in 1,...,c do
9:      if X[i] == true then
10:         R.add(i)
11: return R
```

Sei A die Menge an Klauseln für welche die Bedingung in Zeile 3 wahr ist und sei s die Anzahl an Klauseln in A. Es gibt keine zwei Klauseln in A, welche das gleiche Literal enthalten, da wir in X die bisher gesehenen Literale verfolgen und die Bedingung in Zeile 3 pro Literal höchstens einmal wahr werden kann. Somit besteht das minimale Lösungsset von K aus **mindestens** s Literalen. Die Lösung von unserem Algorithmus besteht aus genau $3s$ Literalen, da wir pro Klausel in A drei Literale in die Lösungsmenge aufnehmen. Somit berechnet der Algorithmus eine $3s/s = 3$-Approximation.

Lösungsvorschlag

Aufgabe 5. Geometrische Algorithmen: Schnitte und Polygone [11 Punkte]

a. Ein berühmter Zeichner zeichnet Skizzen bestehend aus mehreren einfachen Polygonen, **die sich gegenseitig nicht berühren oder schneiden**.
Ein **einfaches Polygon** $O := (P_0, \ldots, P_n = P_0), P_i \in \mathbb{R} \times \mathbb{R}$ ist ein Streckenzug bestehend aus den n Strecken $\overline{P_iP_{i+1}}$, wobei sich der Streckenzug **selbst nicht berührt oder schneidet**. Anstatt den Strecken $\overline{P_iP_{i+1}}$ des Polygons zeichnet der Zeichner jeweils eine Linie $\overline{L_i}$ durch die Punkte P_i und P_{i+1}, welche über diese beiden Eckpunkte des Polygons herausragt. Die gezeichnete Linie $\overline{L_i}$ schneidet zwei andere gezeichnete Linien des gleichen Polygons an den Punkten P_i und P_{i+1}. Neben diesen Punkten hat die gezeichnete Linie $\overline{L_i}$ **keine anderen Schnittpunkte**. Hier ein Beispiel mit zwei Polygonen:



Geben Sie einen Algorithmus an, welcher in $\mathcal{O}(k \log k)$ Zeit aus k gezeichneten Linien die eigentlichen Polygone berechnet. Begründen Sie die Laufzeit kurz. Beispiel: Die gezeichneten Linien der oben eingezeichneten Skizze sind $\overline{AD}$, $\overline{BE}$, $\overline{CF}$, $\overline{GJ}$, $\overline{HM}$, $\overline{IL}$ und $\overline{KN}$ und zu berechnenden Polygone sind $O_1 = (P_0, P_1, P_2, P_0)$ und $O_2 = (P_3, P_4, P_5, P_6, P_3)$.

[5 Punkte]

Lösung

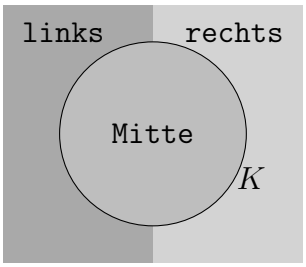
Wir führen den Algorithmus aus der Vorlesung zur Berechnung von Streckenschnitten aus, um die Schnittpunkte der gezeichneten Linien zu finden. Da jede gezeichnete Linie genau zwei Schnittpunkt hat, benötigt diese Berechnung $\mathcal{O}(k \log k)$ Zeit.
Mithilfe der berechneten Schnittpunkte bauen wir einen Graph auf. Jede gezeichnete Linie ist durch einen Knoten im Graph repräsentiert. Es existiert eine ungerichtete Kante zwischen zwei Knoten, falls sich die dazugehörigen gezeichneten Linien schneiden. Die Kanten werden mit dem Schnittpunkt beschriftet. Da jeder Knoten genau zwei Kanten hat, besitzt der Graph k Knoten und k Kanten. Dieser Graph kann in Form eines Adjazenzarrays in $\mathcal{O}(k)$ Zeit aufgebaut werden.
Der Graph besteht ausschließlich aus Kreisen. Es existiert im Graph für jedes Polygon ein Kreis, da sich die Polygone nicht gegenseitig oder selbst berühren oder schneiden. Mittels Tiefensuche traversieren wir einen Kreis nach dem anderen und lesen von jedem Kreis die Kantenbeschriftungen ein. Wir erzeugen für jeden Kreis einen Linienzug aus den eingelesenen Schnittpunkten (Kantenbeschriftungen), sortiert in DFS-Ordnung. Wir erweitern das Ende von jedem Linienzug um seinen ersten Punkt und geben den erweiterten Linienzug als Polygone aus.

b. Ein Kreis $K := (m, r)$ sei definiert durch seinen Mittelpunkt $m \in \mathbb{R} \times \mathbb{R}$ und seinen Radius $r \in \mathbb{R}$. Erweitern Sie einen Algorithmus aus der Vorlesung, um die k Schnittpunkte von n Kreisen zu berechnet. Der Algorithmus soll höchstens $\mathcal{O}((k+n)\log n)$ Zeit benötigen. Nehmen Sie folgende Eigenschaft der Eingabe an:

- Zwei Kreise schneiden sich entweder an zwei Punkten oder an keinem Punkt. Es gibt also keine zwei Kreise, die sich nur berühren.
- Es gibt keine zwei Kreise mit gleichem Mittelpunkt und Radius (keine gleichen Kreise).
- Der oberste Punkt eines Kreises und der unterste Punkt eines Kreises sind keine Schnittpunkte.

Folgende Methoden mit konstanter Ausführungszeit stehen Ihnen zur Verfügung:

- `KreisKreisSchnitt(`K_1`,` K_2`)`: Gibt die Schnittpunkte der Kreise K_1 und K_2 in einer Liste zurück. Falls es keine Schnittpunkte gibt, so wird eine leere Liste zurückgegeben.
- `PunktKreisOrdnung(`p`,` K`)`: Gibt `links` zurück, falls der Punkt p links vom Kreis K liegt, gibt `rechts` zurück, falls der Punkt p rechts vom Kreis K liegt und gibt `Mitte` zurück, falls der Punkt p im Kreis K liegt.



[6 Punkte]

Lösung

Wir erweitern den Algorithmus aus der Vorlesung zur Berechnung von Streckenschnitten, um Kreisschnitte zu berechnen. Dazu zerschneiden wir jeden Kreis an seinem Mittelpunkt vertikal in zwei Hälften und erweitern den Algorithmus zur Berechnung von Streckenschnitten, um Schnitte zwischen Halbkreisen zu berechnen.

- Anstatt den Beginn und das Ende jeder Linie als Event in die Prioritätsliste einzufügen, fügen wir den oberen und unteren Punkt jedes Halbkreises in die Prioritätsliste ein.
- Um den Schnittpunkt zwischen zwei Halbkreisen zu berechnen, schneiden wir die beiden dazugehörigen Kreise und geben nur diejenigen Schnitte zurück, welche auf den beiden betrachteten Halbkreisen liegen.
- Sei T die vergleichsbasierte Datenstruktur, welche die Linien sortiert nach ihrer relativen Ordnung auf der Sweepline verwaltet. Diese Datenstruktur speichert nun Halbkreise der Form $(m, r, \texttt{links})$ und $(m, r, \texttt{rechts})$. Wird beim Einfügen in T ein neuer Halbkreis $H = (m := (x,y), r, -))$ mit einem schon eingefügten Halbkreis $H' = (m := (x',y'), r', o')$ verglichen, so wird die Funktion $H < H'$ ausgeführt. Diese Funktion berechnet zuerst die Ordnung des oberen Punktes $(x, y+r)$ des Halbkreises H bezüglich des kompletten Kreises (m, r') von H' mit dem Aufruf $o :=\texttt{PunktKreisOrdnung}\,((x,y+r),(m,r'))$. Die folgende Tabelle definiert nun auf Grundlage von o und o' das Ergebnis der Funktion $H < H'$:

o'	o	$H < H'$
$-$	`links`	true
$-$	`rechts`	false
links	`Mitte`	false
rechts	`Mitte`	true

  Eine Lösung, welche behauptet, dass mit der Funktion `PunktKreisOrdnung` die Ordnung vom oberen Ende des neuen Halbkreises bezüglich einem anderen Halbkreis auf Höhe der aktuellen Sweepline berechnet werden kann, wird auch akzeptiert.

Neben diesen Anpassungen wird der Algorithmus aus der Vorlesung wie gewohnt ausgeführt.

Lösungsvorschlag

Aufgabe 6. Stringalgorithmen: Textrekonstruktion [10 Punkte]

a. Berechnen Sie für den Text „*bcbabcb*\$“ das Suffix-Array und das LCP-Array.

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

[4 Punkte]

Lösung

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

b. Gegeben sei das LCP-Array eines Textes T. Wie lässt sich die Anzahl an unterschiedlichen Symbolen in T berechnen, falls Ihnen ausschließlich das dazugehörige LCP-Array zur Verfügung steht?

[2 Punkte]

Lösung

Die Anzahl an unterschiedlichen Symbole im Text T ist durch die Anzahl an Einträgen mit dem Wert „0“ im LCP-Array definiert.

c. Gegeben sei das LCP-Array und das Suffix-Array von einem Ursprungstext T der Länge n. T enthält alle Zeichen des ASCII-Alphabets im Bereich von 36 (für „$“) bis 126 (für „∼“). Die anderen Zeichen des ASCII-Alphabets kommen in T nicht vor. Leider wurde der Ursprungstext T gelöscht. Geben Sie einen Algorithmus in Pseudocode an, der in $\mathcal{O}(n)$ Zeit den Text T rekonstruiert.

[4 Punkte]

Lösung

Der folgende Algorithmus berechnet den Text T mithilfe des LCP-Arrays LCP und des Suffix-Arrays SA:

Algorithm 3 `SA: Suffix-Array, LCP: LCP-Array, n: Länge des Textes`

```
T: Array der Länge n                          ▷ Initialisierte Text
s ← 35                                        ▷ Initialisiere aktuelles Symbol mit Terminalsymbol
for i ← 0 to n − 1 do                         ▷ Iteriere über alle Einträge im Text
    if LCP[i] == 0 then
        s = s + 1                             ▷ Suffix SA[i] beginnt mit neuem Zeichen s + 1
    T[SA[i]] ← s
return T
```