Name:

Vorname:

Matrikelnummer:

Klausur-ID:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders 15.03.2022

Klausur Algorithmen II

Aufgabe 1.	Kleinaufgaben	10 Punkte
Aufgabe 2.	Parallele Algorithmen: Sortieren auf Hyperwürfeln	9 Punkte
Aufgabe 3.	Externe Algorithmen: LevelDB	12 Punkte
Aufgabe 4.	Approximationsalgorithmen: Minimale S-Schnitte	10 Punkte
Aufgabe 5.	Stringology: Textkompression mit Suffix-Bäumen	11 Punkte
Aufgabe 6.	Geometrische Algorithmen: Rasensprenger	8 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 **17 Blätter**.
- Zum Bestehen der Klausur sind 20 Punkte hinreichend.

Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [10 Punkte]

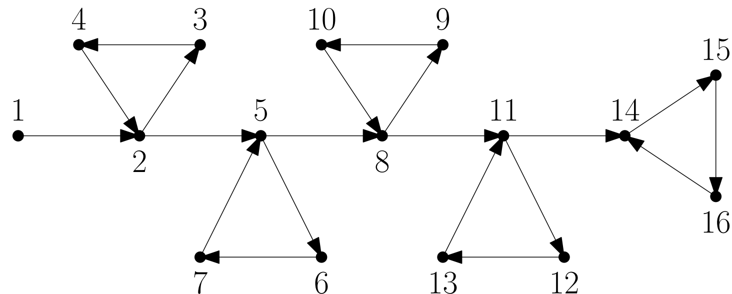
a. Ein Algorithmus habe Laufzeit $\mathcal{O}(f(n,\varepsilon))$. Sein berechnetes Ergebnis weiche um den Faktor $\rho(n,\varepsilon)$ vom optimalen Wert ab. Geben Sie für die folgenden Fälle an, ob ein PTAS, FPTAS, APX oder keins davon vorliegt. Begründen Sie Ihre Antworten kurz. [4 Punkte]

1. $f(n,\varepsilon) = n^{1/\varepsilon}$, $\rho(n,\varepsilon) = 2+\varepsilon^2$
2. $f(n,\varepsilon) = n^{1+\varepsilon}$, $\rho(n,\varepsilon) = 1+\varepsilon$
3. $f(n,\varepsilon) = n + \frac{1}{\varepsilon^{42}}$, $\rho(n,\varepsilon) = 1-\varepsilon$
4. $f(n,\varepsilon) = n + \log(n)^{1/\varepsilon}$, $\rho(n,\varepsilon) = 1+\varepsilon^2$

Lösung

1. APX, da Approximationsalgorithmus mit konstanter Approximationsgüte (für fixes ε), aber keine $(1+\varepsilon)$ Approximation.
2. FPTAS, da $(1+\varepsilon)$ Approximation und polynomiell in n und $1/\varepsilon$.
3. FPTAS, da $(1-\varepsilon)$ Approximation und polynomiell in n und $1/\varepsilon$.
4. PTAS, da $(1+\tilde{\varepsilon})$ Approximation (mit $\tilde{\varepsilon} := \sqrt{\varepsilon}$), aber nicht polynomiell in $1/\tilde{\varepsilon}$.

b. Der Algorithmus aus der Vorlesung zur Bestimmung von starken Zusammenhangskomponenten (SCC) verwendet 2 Stacks. Zur Erinnerung: `oNodes` speichert alle Knoten in offenen Komponenten und `oReps` speichert die Repräsentanten der offenen Komponenten. Was ist jeweils die maximale Anzahl von Elementen in `oNodes` und `oReps`, während der folgende Graph abgearbeitet wird? Wie viele SCCs werden gefunden? Die Knoten sind mit ihrer `dfsNum` beschriftet. [3 Punkte]



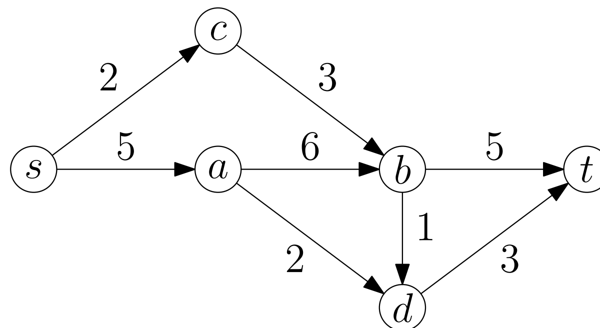
Lösung

max $|$`oReps`$| =$ 8	max $|$`oNodes`$| =$ 16	$|\mathrm{SCC}| =$ 6

Durch *schlaues Hinsehen* lässt sich das Ergebnis ohne Ausführung des Algorithmus erkennen. Bei jedem der dreieckigen SCCs besitzt der Repräsentant eine weitere ausgehende Kante, die nach der Abarbeitung des Dreiecks betrachtet wird. Dadurch finden bis zur Betrachtung des letzten Knotens keine "backtrack"-Operationen auf Repräsentanten statt. Es liegen also alle 16 Knoten auf `oNodes`.

Nach Abarbeitung jedes der ersten 4 Dreiecke bleibt jeweils nur noch dessen Repräsentant auf dem Stack `oReps`. Dieser wird aber, da er eine weitere ausgehende Kante hat, noch nicht vom Stack entfernt. Kurz vor dem Betrachten der letzten Kante (ausgehend von Knoten 16) liegen also der Startknoten, die Repräsentanten der 5 Dreiecke und die beiden Nicht-Repräsentanten des letzten Dreiecks auf `oReps`.

c. Betrachten Sie das unten abgebildete Flussnetzwerk. Die Kanten sind mit ihren Kapazitäten beschriftet. Berechnen Sie mit dem Ford Fulkerson Algorithmus den maximalen Fluss von der Quelle s zur Senke t. Geben Sie dabei für jeden Schritt den augmentierenden Pfad in Form einer Knotenliste und den Fluss, den Sie über diesen Pfad schieben, an. Geben Sie zusätzlich den Wert des maximalen Flusses nach der Ausführung an. [3 Punkte]



Lösung

Mögliche augmentierende Pfade:

- $p_1 = s \to a \to b \to t$. Fluss: 5.
- $p_2 = s \to c \to b \to d \to t$. Fluss: 1.
- $p_2 = s \to c \to b \to a \to d \to t$. Fluss: 1.

Maximaler Fluss: 7.

Lösungsvorschlag

Aufgabe 2. Parallele Algorithmen: Sortieren auf Hyperwürfeln [9 Punkte]

Gegeben seien $p = 2^d$ nachrichtengekoppelte Prozessoren, die in einem Hyperwürfel verbunden sind. Zur Erinnerung: In einem Hyperwürfel sind genau die Prozessoren miteinander verbunden, deren Prozessor-Indizes sich in genau einem Bit unterscheiden. Zur Vereinfachung können Sie davon ausgehen, dass ein Austausch von Nachrichten der Länge l zwischen zwei verbundenen Prozessoren die Zeit $\mathcal{O}(l)$ benötigt.

a. Geben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(dl)$ an, der eine Nachricht der Länge l vom Prozessor 0 an alle $p = 2^d$ Prozessoren verteilt (Broadcast). Begründen Sie die Laufzeit. [2 Punkte]

Lösung

Die Verbreitung der Nachricht funktioniert wie beim schrittweisen Aufbau eines Hyperwürfels. Im Schritt $i \in \{1,...,d\}$ senden die Prozessoren $\texttt{0}^{d-i}\texttt{0}\star^{i-1}$ ihre Nachricht an Prozessoren $\texttt{0}^{d-i}\texttt{1}\star^{i-1}$ (also an ihre Kopie im nächst größeren Hyperwürfel). Jedes Senden einer Nachricht benötigt Zeit $\mathcal{O}(l)$ und es gibt d Schritte, also insgesamt $\mathcal{O}(dl)$.

b. Seien $n \gg p$ verschiedene Objekte gegeben, die vergleichsbasiert sortiert werden sollen. Anfangs seien die Objekte gleichmäßig und **zufällig** auf die p Prozessoren verteilt. Geben Sie eine Adaption des Quicksort-Algorithmus für Hyperwürfel an. Der Algorithmus soll die Laufzeit $\mathcal{O}\left(\frac{n}{p}\log\left(\frac{n}{p}\right) + \log(p)\left(\log(p) + \frac{n}{p}\right)\right)$ besitzen. Begründen Sie, warum Ihr Algorithmus die Laufzeit besitzt.
Hinweis: Sind die Objekte zufällig auf die Prozessoren verteilt, ist der lokale Median auf einem der Prozessoren eine gute Approximation für den globalen Median aller Prozessoren. Sie dürfen in dieser Aufgabe annehmen, dass bei der Partitionierung an einem Median, der auf einem der Prozessoren bestimmt wurde, keine Last-Imbalancen auftreten. [4 Punkte]

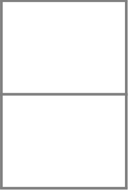
Lösung

Verwende $\log(p)$ Iterationen, welche die Prozessoren rekursiv in jeweils zwei Sub-Hyperwürfel ($\texttt{xyz0}\star\star\star$ und $\texttt{xyz1}\star\star\star$) aufspalten. In jeder Iteration bestimmt der erste Prozessor des Sub-Hyperwürfels ($\texttt{xyz0000}$) seinen lokalen Median. Der Median kann z.B. mit Quickselect in Zeit $\mathcal{O}\left(\frac{n}{p}\right)$ bestimmt werden.
Der Median wird dann als Pivot innerhalb des Sub-Hyperwürfels verteilt. Dies entspricht einem Broadcast und benötigt Zeit $\mathcal{O}(\log(p))$. Jeder Prozessor klassifiziert jetzt lokal in große und kleine Objekte, was Zeit $\mathcal{O}\left(\frac{n}{p}\right)$ benötigt. Dann tauschen jeweils die 2 Nachbarn in der aktuell aufzuspaltenden Dimension ihre Objekte aus: Die Objekte, die größer sind als der Pivot, werden an den Prozessor geschickt, der in der Dimension eine 1 stehen hat. Die Objekte, die kleiner sind als der Pivot, werden an den Prozessor mit der 0 geschickt.
Durch die Vereinfachung brauchen die Nachrichten Zeit $\mathcal{O}\left(\frac{n}{p}\right)$. Zudem sind (da der lokale Median nahe des echten Medians ist) die Prozessoren wieder alle gleichmäßig ausgelastet. Im Gegensatz zum allgemeinen, parallelen Quicksort tritt also keine Imbalance auf und die Prozessoren können anhand der nächst kleineren Dimension in die Rekursion starten.
Am Ende muss nur noch lokal in $\mathcal{O}\left(\frac{n}{p}\log\left(\frac{n}{p}\right)\right)$ sortiert werden. Es ergibt sich eine Laufzeit von $\mathcal{O}\left(\frac{n}{p}\log\left(\frac{n}{p}\right) + \log(p)\left(\log(p) + \frac{n}{p}\right)\right)$.

c. Gehen Sie nun davon aus, dass Sie $n \gg p$ Objekte gegeben haben, die bereits global aufsteigend sortiert sind. Die n/p kleinsten Objekte liegen also auf Prozessor 0, die nächsten auf Prozessor 1, usw. Für eine Anwendung wird aber nun eine absteigende Sortierung benötigt. Geben Sie einen Algorithmus für Hyperwürfel an, der die Objekt-Reihenfolge umkehrt. Der Algorithmus soll die Laufzeit $\mathcal{O}\left(\frac{n}{p} + \log(p)\frac{n}{p}\right)$ besitzen. Begründen Sie, warum Ihr Algorithmus die Laufzeit besitzt. [3 Punkte]

Lösung

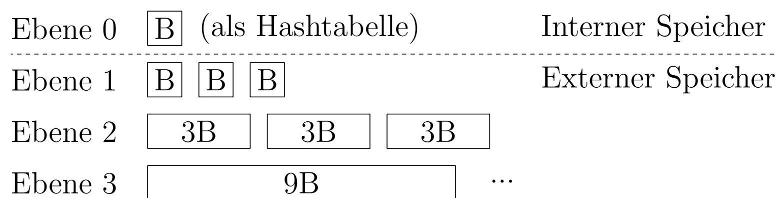
Die Reihenfolge der Objekte muss zunächst lokal invertiert werden. Dies ist durch einen linearen Scan und Vertauschen von jeweils zwei Objekten in Zeit $\mathcal{O}\left(\frac{n}{p}\right)$ möglich.
Um die globale Ordnung umzukehren, muss Prozessor i all seine Objekte mit Prozessor $p-i$ austauschen. Es müssen also gerade die Prozessoren ihre Ojekte austauschen, deren Prozessor-Indizes sich in jedem Bit unterscheiden (also im Hyperwürfel genau gegenüber).
Wir können also die $\log(p)$ Bits der Prozessor-Indizes der Reihe nach durchgehen und jeder Prozessor vertauscht seine Liste mit der dessen Nachbarn, der sich genau in diesem Bit unterscheidet. Jeder Austausch dauert Zeit $\mathcal{O}\left(\frac{n}{p}\right)$, also haben wir insgesamt eine Laufzeit von $\mathcal{O}\left(\frac{n}{p} + \log(p)\frac{n}{p}\right)$. Dadurch bewegt sich jede Liste in jedem Schritt in Richtung ihres Ziels und außerdem kommt es nicht zu Stau-Situationen, denn jeder Prozessor hat immer nur eine Liste.



Lösungsvorschlag

Aufgabe 3. Externe Algorithmen: LevelDB [12 Punkte]

LevelDB ist ein externer *Key-Value-Store*, in dem Objekte mittels eines Schlüssels eingefügt und wieder abgerufen werden können. Eine vereinfachte Version von LevelDB besteht aus mehreren Ebenen. Ebene 0 ist eine Hashtabelle im internen Speicher, in der bis zu B Objekte gespeichert sind. Die restlichen Ebenen liegen im externen Speicher. Ebene $i > 0$ speichert bis zu r *Runs*, wobei $r \geq 2$ ein Eingabeparameter ist. Ein Run in Ebene i ist eine sortierte Liste von $B \cdot r^{(i-1)}$ Objekten. Nachfolgende Illustration zeigt ein Beispiel für $r = 3$ und drei Ebenen im externen Speicher.
Beim Einfügen eines neuen Objekts wird zunächst versucht, es in der internen Hashtabelle zu speichern. Speichert die Hashtabelle bereits B Objekte, werden vor dem Einfügen die bereits enthaltenen Objekte nach ihren Schlüsseln sortiert und als Run in der ersten Ebene abgelegt. Speichert eine Ebene bereits r Runs und ein neuer soll darin abgelegt werden, werden zuerst alle Objekte der Ebene mit einem externen r-Wege Merge-Algorithmus sortiert und als r-mal größerer Run in der nächst höhere Ebene abgelegt.



a. Sei eine LevelDB-Datenbank mit k Ebenen im externen Speicher gegeben. Es wird nun ein neues Objekt eingefügt. Wie viele r-Wege Merge-Operationen müssen dafür im schlimmsten Fall ausgeführt werden? Begründen Sie kurz. [1 Punkt]

Lösung

Sind alle k externen Ebenen und die interne Hashtabelle bereits voll, kommt es beim Einfügen in jeder Ebene zu einer Merge-Operation. Die gesamte Anzahl der r-Wege Merge-Operationen ist also k.

b. Es werden n Objekte in eine leere LevelDB-Datenbank eingefügt. Zeigen Sie, dass die Datenbank nun $\mathcal{O}(\log_r(n/B))$ Ebenen besitzt.
Hinweis: $\sum_{i=0}^{k} x^i = \frac{x^{k+1}-1}{x-1}$ [3 Punkte]

Lösung

Eine komplett gefüllte LevelDB-Datenbank mit k Ebenen enthält folgende Anzahl an Elementen:

$$n = B + \sum_{i=1}^{k} Br^{i-1}r = B + Br \cdot \sum_{i=1}^{k} r^{i-1} = B + Br \cdot \sum_{i=0}^{k-1} r^i = B + Br \cdot \frac{r^{k-1+1}-1}{r-1}$$
$$\iff r^k = \frac{(n-B)(r-1)}{Br} + 1$$
$$\iff k = \log_r\left(\frac{(n-B)(r-1)}{Br} + 1\right) = \log_r\left(\left(\frac{n}{B} - \frac{B}{B}\right) \cdot \frac{r-1}{r} + 1\right)$$
$$\leq \log_r\left(\left(\frac{n}{B} - \frac{B}{B} + 2\right) \cdot \frac{r-1}{r}\right) = \log_r\left(\frac{n}{B} + 1\right) + \log_r\left(\frac{r-1}{r}\right)$$
$$\leq \log_r\left(\frac{n}{B} + 1\right) + \log_r(1) \in \mathcal{O}\left(\log_r \frac{n}{B}\right)$$

Anmerkung: Die Basis r im Logarithmus ist im O-Kalkül nicht irrelevant, weil r ein Eingabeparameter ist. Eine alternative Darstellung wäre $\log_r(n/B) = \log(n/B)/\log(r)$

c. Geben Sie einen Algorithmus an, der ein Objekt mit einem bestimmten Schlüssel aus der Datenbank abruft. Der Algorithmus darf maximal sublinear viele I/O-Zugriffe in der Anzahl der gespeicherten Objekte n ausführen. Jeder I/O-Zugriff kann einen Block der Größe B aus dem externen Speicher lesen. [3 Punkte]

Lösung

Durchsuche zunächst die interne Hashtabelle in Zeit $\mathcal{O}(1)$. Extern könnte ein angefragtes Objekt in jedem Run jeder Ebene stehen. Es müssen also alle Runs durchsucht werden. Innerhalb eines Runs kann binäre Suche verwendet werden. Dadurch werden pro Run nur logarithmisch viele I/O-Zugriffe gebraucht, was die gesamte Laufzeit sublinear macht.

d. Gegeben sei ein Bit-Array der Länge n, das mit Nullen gefüllt ist. Es werden nun n unabhängig zufällige Bits auf 1 gesetzt (einige Positionen möglicherweise mehrmals). Zeigen Sie, dass der erwartete Anteil an Bits, die am Ende noch 0 sind, für $n \to \infty$ gegen $1/e$ konvergiert.
Hinweis: $e^x = \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n$ [2 Punkte]

Lösung

Betrachten wir ein beliebiges aber festes Bit im Array. Beim Setzen eines zufälligen Bits ist die Wahrscheinlichkeit, dass dieses feste Bit nicht gesetzt wird, $(1 - 1/n)$. Setzen wir also n Bits, ist die Wahrscheinlichkeit, dass es nie gesetzt wird, $(1 - 1/n)^n \to 1/e$. Ist jedes Bit unabhängig voneinander mit Wahrscheinlichkeit $1/e$ gesetzt, können wir insgesamt auch einen Anteil von $1/e$ nicht gesetzten Bits erwarten.

e. Nehmen Sie nun an, dass im internen Speicher für jeden Run einer LevelDB-Datenbank genug Platz für ein Bit-Array ist. Die Länge des Bit-Arrays entspricht der Anzahl an Objekten im Run. Wie können Sie das Bit-Array benutzen, um die erwartete Anzahl I/O-Zugriffe für den Abruf eines Objekts um einen Faktor von etwa $1/e$ zu reduzieren? Beschreiben Sie insbesondere, wie der Abruf abläuft. [3 Punkte]

Lösung

Wir nehmen uns pro Run eine Hashfunktion, die von einem Schlüssel auf eine Position im jeweiligen Bit-Array abbildet. Beim Mergen von Runs wird für jedes enthaltene Objekt das Bit gesetzt, das die Hashfunktion auf dessen Schlüssel ausgibt. Es treten zwischen den Objekten des Runs Kollisionen auf und das Bit-Array besitzt noch einen Anteil von erwartet $1/e$ Nullen (siehe Teilaufgabe c).
Bei der Anfrage eines Schlüssels wird wieder die gleiche Hashfunktion ausgeführt. Ist das entsprechende Bit nicht gesetzt, kann man sich sicher sein, dass das Objekt nicht in dem Run gespeichert ist. Der Run muss also extern nicht durchsucht werden. Ist das Bit gesetzt, ist entweder das Objekt im entsprechenden Run gespeichert oder es gab eine Kollision der Hashfunktion. Der Run muss also wie in Teilaufgabe b durchsucht werden. Da im Array $1/e$ der Positionen 0 sind, kann bei jedem Abruf die Anzahl der durchsuchten Runs und damit die I/O-Zugriffe um $1/e$ rediziert werden. Der Run, in dem ein Objekt enthalten ist, muss auf jeden Fall durchsucht werden, aber die Reduktion der I/O-Operationen konvergiert für $n \to \infty$ trotzdem zum Faktor $1/e$.
Diese probabilistische Datenstruktur zum Filtern von Anfragen ist auch bekannt unter dem Namen *Bloom Filter*.

Lösungsvorschlag

Aufgabe 4. Approximationsalgorithmen: Minimale S-Schnitte [10 Punkte]

Gegeben sei ein ungerichteter Graph $G = (V, E, c)$ mit Kantengewichten $c : E \to \mathbb{R}_{>0}$. Sei ferner $S = \{s_1, \dots, s_k\} \subseteq V$ eine Knotenteilmenge, deren Elemente wir als *Terminalknoten* bezeichnen. Ein *S-Schnitt* in G ist eine Kantenteilmenge $E' \subseteq E$ mit der Eigenschaft, dass nach Entfernung von E' aus G alle Terminalknoten voneinander getrennt sind (d. h. jede Zusammenhangskomponente von $G - E'$ enthält maximal einen Terminalknoten). Ein *minimaler S-Schnitt* E^{OPT} ist ein S-Schnitt mit minimalem Gewicht $c(E^{\mathrm{OPT}}) := \sum_{e \in E^{\mathrm{OPT}}} c(e)$ unter allen S-Schnitten. Betrachten Sie für die Berechnung eines S-Schnittes den folgenden Approximationsalgorithmus.

Algorithmus 1 `S-Schnitt`$(G = (V, E, c), S)$

- $E_1, \dots, E_k \leftarrow \emptyset$
- **for** $s_i \in S$ **do**
  - // Berechne minimalen Schnitt zwischen s_i und $S \setminus \{s_i\}$:
  - $t_i \leftarrow$ `neuerKnoten()`
  - $G_i \leftarrow (V \cup \{t_i\}, E \cup \{\{s_j, t_i\} \mid s_j \in S \setminus \{s_i\}\})$ // verbinde $S \setminus \{s_i\}$ zu neuem Knoten t_i ...
  - $c_i \leftarrow \begin{cases} c(e), & e \in E \\ \infty, & \text{sonst} \end{cases}$ // ... mit Kantengewicht ∞
  - $E_i \leftarrow$ `minimalerSTSchnitt`(G_i, c_i, s_i, t_i)
- // s_i-t_i-Schnitte E_i bis auf einen ergeben S-Schnitt:
- $j \leftarrow \arg\max_i c(E_i)$ // E_j ist schwerster s_i-t_i-Schnitt
- **return** $E_1 \cup \dots \cup E_{j-1} \cup E_{j+1} \cup \dots \cup E_k$ // Vereinigung aller E_i außer E_j

a. Beschreiben Sie kurz, wie Sie die Operation `minimalerSTSchnitt` mit Hilfe eines Algorithmus zur Berechnung maximaler Flüsse implementieren können. [2 Punkte]

Lösung

Betrachte den Residualgraphen eines maximalen Flusses. Seien S alle von s erreichbaren Knoten. Dann ist $(S, V \setminus S)$ ein minimaler Schnitt.

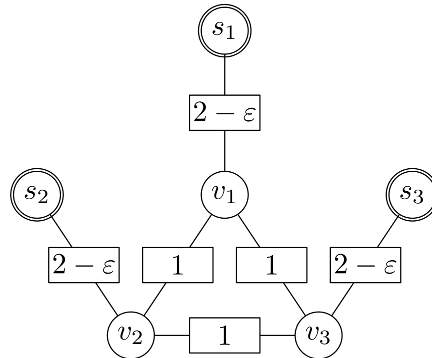
b. Geben Sie die Laufzeit des obigen Approximationsalgorithmus an, wenn die Operation `minimalerSTSchnitt` mit dem Arbitrary Preflow Push Algorithmus implementiert wird. [1 Punkt]

Lösung

Der dominierende Teil ist die Berechnung der k minimalen s-t-Schnitte: $\mathcal{O}\left(kn^2m\right)$.

c. Betrachten Sie den folgenden Graphen mit den markierten Terminalknoten $S = \{s_1, s_2, s_3\}$. Gewichten Sie die Kanten des Graphens derart, dass der Approximationsalgorithmus **unabhängig der berechneten minimalen s_i-t_i-Schnitte** keinen minimalen S-Schnitt berechnet. Geben Sie zusätzlich einen konkreten S-Schnitt in Form einer Kantenliste an, den der Algorithmus für Ihren gewichteten Graphen berechnen könnte, sowie einen minimalen S-Schnitt. [3 Punkte]

Lösung



Der Approximationsalgorithmus schneidet zum Beispiel s_1v_1 und s_2v_2 (Gewicht: $4-2\varepsilon$). Ein minimaler S-Schnitt schneidet v_1v_2, v_2v_3, v_1v_3 (Gewicht: 3).

d. Sei $G = (V,E)$ ein ungerichteter Graph mit Terminalknoten $S = \{s_1, \ldots, s_k\} \subseteq V$ und sei $E^{\text{OPT}} \subseteq E$ ein minimaler S-Schnitt in G. Für jeden Terminalknoten $s_i \in S$ induziert E^{OPT} einen s_i-t_i-Schnitt (mit t_i wie im Algorithmus) $E_i^{\text{IND}} \subseteq E^{\text{OPT}}$, der aus den Kanten besteht, die s_i von $S \setminus \{s_i\}$ trennen. Zeigen Sie, dass gilt:

$$\sum_{i=1}^{k} c(E_i^{\text{IND}}) = 2 \cdot c(E^{\text{OPT}}).$$

[2 Punkte]

Lösung

Für $s_i \in S$ sei V_i die Zusammenhangskomponente von $G - E^{\text{OPT}}$, in der s_i liegt. Sei $e = uv \in E^{\text{OPT}}$. Dann liegen u und v jeweils in einer Menge V_i und V_j, denn andernfalls wäre $E^{\text{OPT}} - e$ ebenfalls ein S-Schnitt mit kleinerem Gewicht. Also liegt e in genau zwei induzierten s-t-Schnitten, woraus die Behauptung folgt.

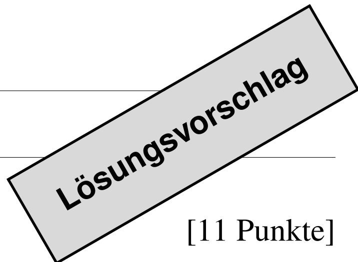
e. Zeigen Sie, dass ein vom Approximationsalgorithmus berechneter S-Schnitt maximal doppelt so schwer ist wie ein minimaler S-Schnitt.
Hinweis: Verwenden Sie Teilaufgabe d. [2 Punkte]

Lösung

Die von einem optimalen S-Schnitt E^{OPT} induzierten s_i-t-Schnitte E_i^{IND} sind mindestens so schwer wie die vom Approximationsalgorithmus berechneten minimalen s_i-t-Schnitte E_i^{ALG}. Damit folgt für einen vom Approximationsalgorithmus berechneten S-Schnitt E^{ALG}:

$$c(E^{\text{ALG}}) \leq \sum_{i=1}^{k} c(E_i^{\text{ALG}}) \leq \sum_{i=1}^{k} c(E_i^{\text{IND}}) = 2c(E^{\text{OPT}}),$$

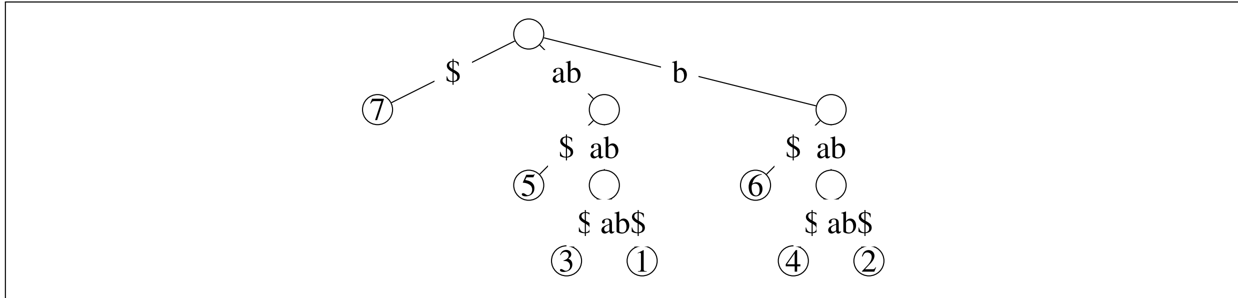
wobei die letzte Identität aus Teilaufgabe d stammt.



Aufgabe 5. Stringology: Textkompression mit Suffix-Bäumen [11 Punkte]

a. Erstellen Sie den Suffix-Baum für den Text $T =$ `ababab$`. Beschriften Sie die Kanten mit den expliziten Kantenbeschriftungen und sortieren Sie diese lexikographisch.
Hinweis: Ein Suffix-Baum ist ein kompakter Trie. [2 Punkte]

Lösung



b. Welcher Text wird durch die folgende LZ77-Faktorisierung kodiert?

$$(0,a)(0,b)(0,c)(5,1)(0,\$)$$

[2 Punkte]

Lösung

1. $(0,a) \Rightarrow T =$ `a`
2. $(0,b) \Rightarrow T =$ `ab`
3. $(0,c) \Rightarrow T =$ `abc`
4. $(5,1) \Rightarrow T =$ `abcabcab`
5. $(0,\$) \Rightarrow T =$ `abcabcab$`

c. Gegeben sei ein Suffix-Baum für einen Text der Länge n und ein Pattern der Länge m. Geben Sie einen Algorithmus an, der das *erste* Vorkommen des Patterns im Text findet. Sie dürfen Zeit $\mathcal{O}(n)$ für Vorverarbeitung verwenden. Eine Anfrage muss dann in Zeit $\mathcal{O}(m)$ beantwortet werden. Sie können hierbei annehmen, dass Sie in konstanter Zeit die Kinderkante mit dem passenden Zeichen finden. In den Blättern sei der Index des jeweiligen Suffix gespeichert. [3 Punkte]

Lösung

Zunächst traversieren wir den Baum und speichern für jeden inneren Knoten v die Nummer c_v ab, die dem kleinsten Suffix in dem Teilbaum unter v entspricht. Diese Zahlen können wir mit einer Tiefensuche in $\mathcal{O}(n)$ Zeit bestimmen. Für jeden Knoten v gibt uns die Zahl c_v nun an, welches das – in Textreihenfolge – erste Vorkommen des Musters $\lambda(v)$ ist.
Bei einer Anfrage folgen wir Kanten, die dem Pattern entsprechen, so lange, bis wir entweder das Ende des Patterns oder das Ende des Baums erreicht haben. Erreichen wir das Ende des Baums, ist das Pattern nicht enthalten. Erreichen wir das Ende des Patterns, gibt es zwei Möglichkeiten. Sind wir an einem Knoten angekommen, ist c_v dieses Knotens die Textposition des ersten Vorkommens. Sind wir mitten in einer Kante, ist c_v vom Ende dieser Kante die Textposition des ersten Vorkommens.

d. Gegeben sei ein Suffix-Baum für einen Text der Länge n. Geben Sie einen Algorithmus an, der die LZ77-Faktorisierung des Textes in Zeit $\mathcal{O}(n)$ berechnet. Sie können hierbei annehmen, dass Sie in konstanter Zeit die Kinderkante mit dem passenden Zeichen finden. In den Blättern sei der Index des jeweiligen Suffix gespeichert. Neben dem Suffix-Baum haben Sie noch Platz für die Ausgabe und maximal $2n$ Zahlen. [4 Punkte]

Lösung

Wir annotieren den Baum wie in der Pattern-Suche aus der vorherigen Teilaufgabe.
Nehmen wir nun an, dass wir den Text schon bis zur Position i faktorisiert haben. Wir möchten nun also den nächsten Faktor bestimmen, also den längsten Substring $f = T[i..i+\ell)$, der in $T[1..i+\ell)$ zweimal vorkommt. Hierfür folgen wir den Kanten, die dem Substing f entsprechen, so lange, bis wir einer Kante folgen würden, die zu einem Knoten v mit $c_v \geq i$ führt, oder wir kein passendes Zeichen mehr lesen können (was auch mitten auf der Kante passieren kann).
Sei w der letzte Knoten, den wir betreten haben und ℓ die String-Tiefe, bis zu der wir Zeichen vergleichen konnten. Hier müssen auch die Zeichen einbezogen werden, die auf der letzten Kante verglichen wurden. Nun unterscheiden wir zwei Fälle:

1. Wenn $\ell = 0$, dann ist w die Wurzel und wir erzeugen einen Faktor $(0, T[i])$ und setzen den Algorithmus an Textposition $i+1$ fort.

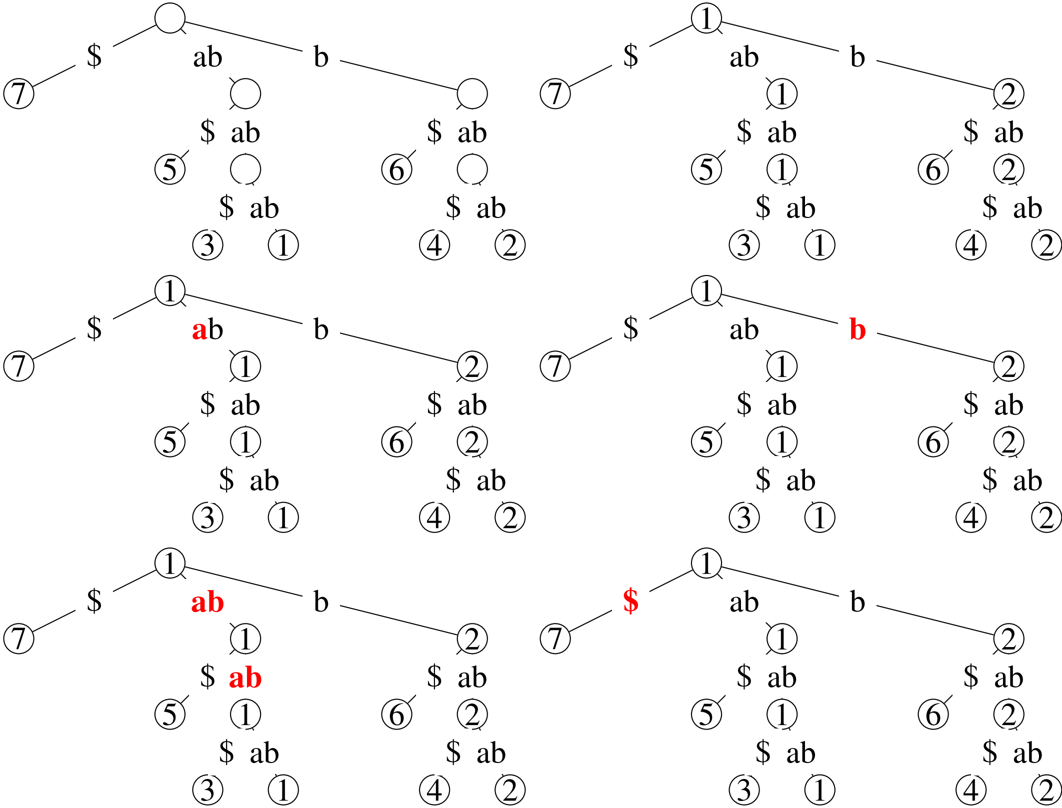
2. Wenn $\ell > 0$, dann haben wir einen neuen Faktor der Länge ℓ, der schon einmal an Textposition c_w vorkommt. Wir können also den Faktor (ℓ, c_w) erzeugen und den Algorithmus an Textposition $i+\ell$ fortsetzen.

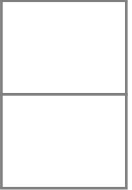
Da wir annehmen, dass wir die passenden Kinderkanten in konstanter Zeit finden können und jedes Zeichen in unserem Text genau einmal im Suffix-Baum suchen, hat unser Algorithmus eine Laufzeit von $\mathcal{O}(n)$. Zudem ist der Algorithmus korrekt, da er immer den längsten Faktor findet (wir suchen so lange wir können), der schon einmal vor der aktuellen Position vorkommt ($c_v < i$) und jedes Zeichen des Textes betrachtet wird. Somit sind alle Anforderungen der LZ77-Faktorisierung erfüllt.

(Beispiel auf dem nächsten Blatt)

Lösung

Beispiel.

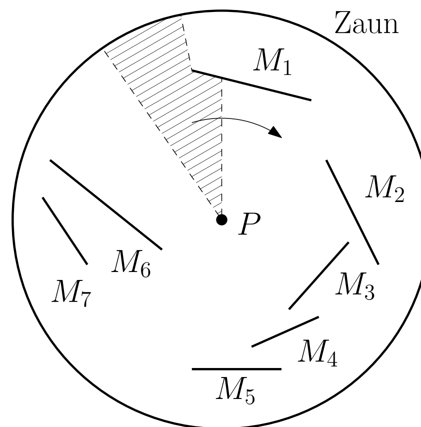




Lösungsvorschlag

Aufgabe 6. Geometrische Algorithmen: Rasensprenger [8 Punkte]

In dieser Aufgabe betrachten wir einen kreisförmig umzäunten Garten, in dem an Punkt P ein Rasensprenger platziert wurde. Der Wasserstrahl des Rasensprengers (schraffierte Fläche) dreht sich im Kreis. Da sich im Garten n Mauern befinden (Linien $M_1, \ldots, M_n$), kann nicht der ganze Garten bewässert werden.
Gehen Sie zur Vereinfachung davon aus, dass sich keine zwei Mauern berühren oder schneiden, und dass alle Endpunkte aller Mauern unterschiedliche Winkel zu P haben.



a. Geben Sie einen Algorithmus an, der in Laufzeit $\mathcal{O}(n \log n)$ alle Mauern ausgibt, die **nicht** nass werden.
Hinweis: Bedenken Sie mögliche Randfälle, die auftreten können. [4 Punkte]

Lösung

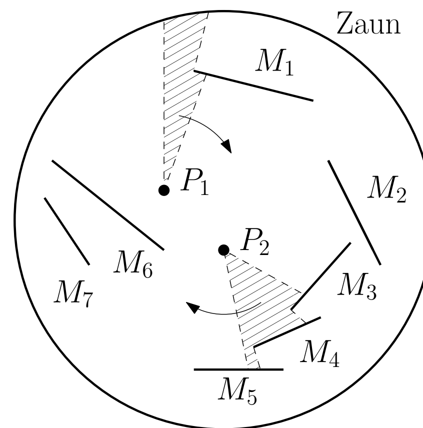
Das Problem kann mit einer Sweep Line gelöst werden, die um P rotiert. Die Sweep Line wird durch eine sortierte Liste T implementiert, die alle aktuell geschnittenen Mauerstücke sortiert nach ihrer Entfernung zu P speichert. In einer solchen sortierten Liste können Liniensegmente (Mauern) in Zeit $\mathcal{O}(\log n)$ eingefügt, gefunden und gelöscht werden. Die Start- und Endpunkte der Mauern werden in sortierter Reihenfolge bezüglich ihres Winkels zu P abgearbeitet. Bei einem Startpunkt wird das Mauerstück in T einsortiert, bei einem Endpunkt gelöscht. Mauerstücke, die während dieses Verfahrens irgendwann das erste Element in T waren, werden nass; die anderen nicht. Achtung: Durch die kreisförmige Abarbeitung kann es sein, dass Mauern die initiale Sweepline im Winkel 0° schneiden. Hier müssen die entsprechenden Start-Events schon vor Beginn des normalen Sweep-Algorithmus in die Sweepline eingefügt werden. Beim Winkel 360° ist der Algorithmus beendet, selbst wenn dann in der Sweepline T noch Elemente enthalten sind.

b. Erweitern Sie Ihren Algorithmus nun derart, dass alle Stücke des äußeren Zaunes ausgegeben werden, die **nicht** nass werden. Ihnen steht dazu eine Funktion anZaun(P,R) zur Verfügung, die einen Punkt R innerhalb des Gartens aus Sicht des Rasensprengers (an Position P) auf den Gartenzaun projiziert. [2 Punkte]

Lösung

Erweitere die Lösung zu Teilaufgabe a wie folgt: Wenn nach Entfernung eines Endpunktes R_1 aus T die Liste leer ist und als nächstes Startpunkt R_2 wieder eingefügt wird, gib Kreissegment anZaun(P,R_1) bis anZaun(P,R_2) als nass aus. Wenn die Sweepline zu Beginn keine Mauer schneidet, gib zusätzlich das Zaunsegment vom letzten Endpunkt bis zum ersten Startpunkt einer Mauer aus. Am Ende invertiere die Auswahl, um alle nicht nassen Zaunsegmente zu erhalten.

c. Im Garten werden nun $k \leq n$ Rasensprenger an Punkten $P_1, \ldots, P_k$ platziert (siehe nachfolgende Skizze für $k = 2$). Geben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(kn \log n)$ an, der erneut alle äußeren Zaunstücke berechnet, die **nicht** nass werden. Begründen Sie die Laufzeit Ihres Algorithmus. [2 Punkte]



Lösung

Wir wiederholen den Algorithmus von Teilaufgabe b für jeden der k Rasensprenger. Das benötigt Zeit $\mathcal{O}(kn \log n)$ und liefert bis zu kn Kreissegmente. Die Start- und Endpunkte der Segmente können mit einer PQ in Zeit $\mathcal{O}(kn \log k) = \mathcal{O}(kn \log n)$ zu einer sortierten Liste zusammengefasst werden. (Alternativ: alle Listen zusammenfassen und in Zeit $\mathcal{O}(kn \log(kn)) = \mathcal{O}(kn \log(n))$ sortieren.) Mit der sortierten Liste kann mit einem Greedy Algorithmus der Schnitt in Zeit $\mathcal{O}(kn)$ berechnet werden.