Name:

Vorname:

Matrikelnummer:

Klausur-ID:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders 23.09.2022

Klausur Algorithmen II

Aufgabe 1.	Kleinaufgaben	12 Punkte
Aufgabe 2.	Kürzeste Wege: A* Algorithmus	10 Punkte
Aufgabe 3.	Predecessor Queries	9 Punkte
Aufgabe 4.	Flussalgorithmen	10 Punkte
Aufgabe 5.	Stringology: LZ77 und Fibonacci-Wörter	9 Punkte
Aufgabe 6.	Geometrische Algorithmen: Stormtrooper	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 **14 Blätter**.
- Zum Bestehen der Klausur sind 30 Punkte hinreichend.

Lösungsvorschlag

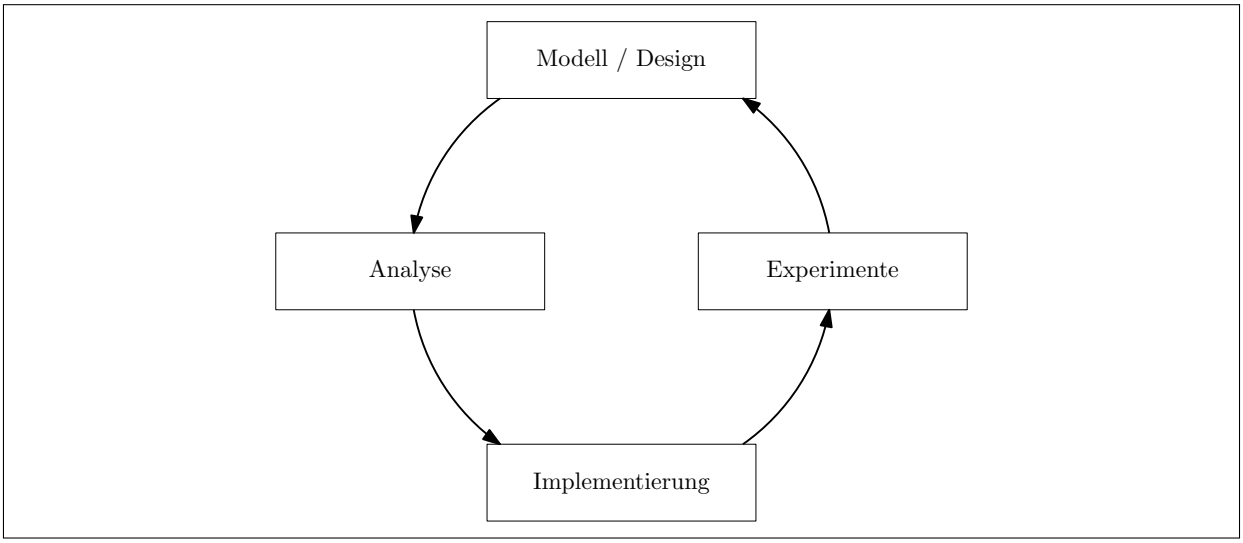
EK

ZK

Aufgabe 1. Kleinaufgaben [12 Punkte]

a. Beschriften Sie den abgebildeten Algorithm Engineering Kreis. [1 Punkt]

Lösung



b. Sei $f(n,k)$ die Laufzeit eines Algorithmus mit Eingabegröße $n \in \mathbb{N}$. Die Laufzeit hängt weiter von einem Parameter $k \in \mathbb{N}$ ab. Geben Sie an, welche der folgenden Laufzeiten ein Problem *fixed-parameter tractable* (FPT) machen. Begründen Sie Ihre Antwort kurz. [3 Punkte]

Lösung

1. $f_1(n,k) = n^{\log(k)}$

   Der Algorithmus macht das Problem nicht FPT, weil die Laufzeit nicht als Produkt zweier Funktionen $f(n)$, $p(k)$ geschrieben werden kann.

2. $f_2(n,k) = 2^k \cdot n^2$

   Der Algorithmus macht das Problem FPT, weil die Laufzeit als $\mathcal{O}(f(n)p(k))$ mit $f(n) = n^2$ Polynom und $p(k) = 2^k$ berechenbar geschrieben werden kann.

3. $f_3(n,k) = n^{1+\frac{1}{k}}$

   Der Algorithmus macht das Problem FPT, weil er insbesondere auch in $\mathcal{O}(f(n)p(k))$ mit $f(n) = n^2$ und $p(k) = 1$ liegt.

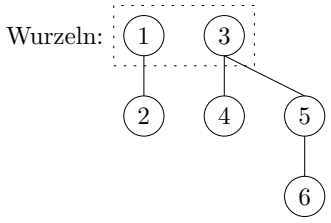
c. Gegeben sei ein vergleichsbasierter, paralleler Sortieralgorithmus für n Objekte und $p = n$ Prozessoren. Der Algorithmus habe den absoluten Speedup $S(p) = \mathcal{O}\left(\sqrt{p}\right)$. Geben Sie dessen Laufzeit und Effizienz an. [2 Punkte]

Lösung

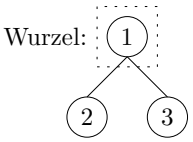
- $E(p) = S(p)/p = \mathcal{O}\left(\sqrt{p}/p\right) = \mathcal{O}\left(1/\sqrt{p}\right)$
- $T_{\text{best}} = \mathcal{O}(n \log n)$
- $S(p) = T_{\text{best}}/T(p) \iff T(p) = T_{\text{best}}/S(p) = \mathcal{O}\left((n \log n)/\sqrt{p}\right) = \mathcal{O}\left(\sqrt{p} \log p\right)$

d. Geben Sie jeweils eine Folge der Heap-Operationen `insert`(x), `deleteMin`$()$ und/oder `decreaseKey`(h, x') an, die einen leeren Fibonacci-Heap in die jeweilige Waldstruktur überführt. Falls Ihre Lösung von einer bestimmten (zulässigen) Union-by-Rank Reihenfolge abhängt, geben Sie außerdem an, welche Knoten oder Teilbäume miteinander vereinigt werden. [4 Punkte]

Lösung



1. `insert`$(0), \ldots,$ `insert`(6)
2. `deleteMin`$()$
3. vereinige 5 mit 6, 3 mit 4, 3 – 4 mit 5 – 6, 1 mit 2



1. `insert`$(0), \ldots,$ `insert`(3)
2. $h \leftarrow$ `insert`(4)
3. `deleteMin`$()$
4. vereinige 4 mit 3, 2 mit 1, 3 – 4 mit 1 – 2
5. `decreaseKey`$(h, 0)$
6. `deleteMin`$()$

e. Ein randomisierter Algorithmus habe Laufzeit $T(n)$. Der Algorithmus schlägt bei 50% der Ausführungen fehl und gibt dann am Ende einen Fehlercode zurück. Wie kann man ihn umwandeln, sodass er immer erfolgreich ist? Um welchen Faktor ist der umgewandelte Algorithmus erwartet langsamer? Begründen Sie kurz.
Hinweise: $\sum_{k=0}^{\infty} q^k = \frac{1}{1-q}$ und $\sum_{k=1}^{\infty} kq^k = \frac{q}{(1-q)^2}$ für $|q| < 1$. [2 Punkte]

Lösung

Führe bei einem Fehler den Algorithmus nochmal aus. Dies ist der standard-Trick, um einen Monte Carlo Algorithmus in einen Las Vegas Algorithmus zu überführen. Die Wahrscheinlichkeit, dass nach k Ausführungen immernoch alle fehlgeschlagen sind, ist $(1/2)^k$. Die erwartete Laufzeit ist also $\sum_{k=0}^{\infty} T(n)(1/2)^k = 2T(n)$. Dadurch ergibt sich ein Faktor von 2, welcher der umgewandelte Algorithmus langsamer ist.

Aufgabe 2. Kürzeste Wege: A* Algorithmus [10 Punkte]

a. Sei $G = (V,E)$ ein gerichteter Graph mit Kostenfunktion $c : E \to \mathbb{R}$. Für einen Landmark $\ell \in V$ und einen Zielknoten $t \in V$ setzen wir wie in der Vorlesung $f_\ell(v) := \mu(v,\ell) - \mu(t,\ell)$, wobei $\mu(u,v)$ die Länge des kürzesten Weges zwischen u und v in G bezeichnet.
Zeigen Sie, dass f_ℓ eine gültige Potentialfunktion für den A* Algorithmus ist, indem Sie die drei dafür in der Vorlesung geforderten Eigenschaften angeben und nachrechnen. [3 Punkte]

Lösung

1. Konsistenz: für $(u,v) \in E$ gilt

$$c(u,v) + f_\ell(v) = c(u,v) + \mu(v,\ell) - \mu(t,\ell) \geq \mu(u,\ell) - \mu(t,\ell) = f_\ell(u),$$

   denn $\mu(u,\ell) \leq c(u,v) + \mu(v,\ell)$.

2. Nicht überschätzen: es gilt

$$f_\ell(v) = \mu(v,\ell) - \mu(t,\ell) \leq \mu(v,t),$$

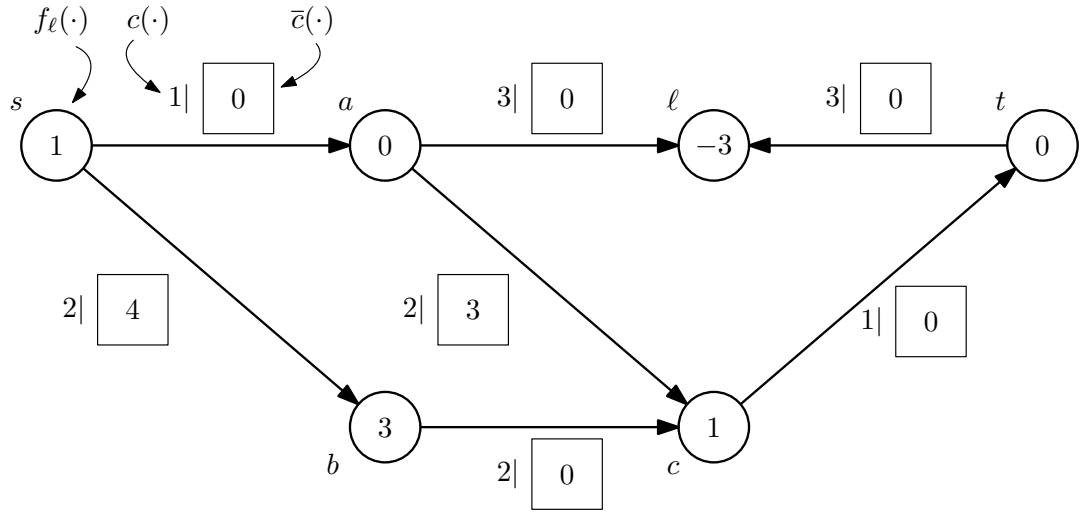
   denn $\mu(v,\ell) \leq \mu(v,t) + \mu(t,\ell)$ (Dreiecksungleichung).

3. Zuletzt gilt

$$f_\ell(t) = \mu(t,\ell) - \mu(t,\ell) = 0.$$

b. Betrachten Sie den folgenden Graphen mit Startknoten s, Zielknoten t und Landmark ℓ. Geben Sie die Potentialfunktion f_ℓ, die reduzierte Kostenfunktion $\bar{c}$ sowie die Reihenfolge an, in der der A* Algorithmus die Knoten *scannt* (also mit einer `deleteMin()` Operation aus der Prioritätswarteschlange entfernt). [3 Punkte]

Lösung



Knotenreihenfolge: s, a, ℓ, c, t

c. Sei $G = (V, E)$ ein gerichteter Graph mit Kostenfunktion $c : E \to \mathbb{R}$ und seien $f, g : V \to \mathbb{R}$ zwei gültige A* Potentialfunktionen. Für eine kürzeste Wege Anfrage von einem Startknoten s zu einem Zielknoten t seien V_f (bzw. V_g) die Knoten, die der A* Algorithmus mit Potentialfunktion f (bzw. g) scannt. Gleiche Prioritäten werden nach kleinerer Knoten ID aufgelöst. Zeigen Sie:

$$\text{wenn } f(v) \geq g(v) \text{ für alle } v \in V, \text{ dann folgt } V_f \subseteq V_g.$$

[3 Punkte]

Lösung

Sei $v \in V_f$, also $\mu_f(s,v) \leq \mu_f(s,t)$. Daraus folgt $\mu(s,v) + f(v) \leq \mu(s,t)$, also $\mu(s,v) + g(v) \leq \mu(s,t)$, also $\mu_g(s,v) \leq \mu_g(s,t)$ und damit $v \in V_g$. Im Gleichheitsfall stimmt die Folgerung auch, da das Tie-Breaking gleich ist.

d. Zeigen Sie, dass der A* Algorithmus mit nicht-negativer Potentialfunktion (also $f(v) \geq 0$ für alle $v \in V$) nie mehr Knoten scannt als Dijkstra's Algorithmus. [1 Punkt]

Lösung

Folgt aus Teilaufgabe c mit f gegeben, $g \equiv 0$.

Lösungsvorschlag

Aufgabe 3. Predecessor Queries [9 Punkte]

Gegeben sei ein aufsteigend sortiertes Array A aus n unterschiedlichen Ganzzahlen $\in [0,U]$. Die Operation $predecessor(x)$ gibt die größte Zahl im Array zurück, die noch kleiner ist als x. Die Operation $predecessorIndex(x)$ gibt den Array-Index zurück, an dem diese Zahl steht. Für Parameter kleiner als alle Array-Einträge sind die beiden Funktionen undefiniert. In dieser Aufgabe beginnen Array-Indizes bei 1.

a. Gegeben sei das Array $A = [1,4,8,10]$. Geben Sie die Werte der folgenden Operationen an. [1 Punkt]

Lösung

$predecessor(6) = 4, predecessorIndex(7) = 2$

Der Arbeitsspeicher sei in Seiten der Größe B aufgeteilt. Zusätzlich gibt es einen Cache, der deutlich schneller als der Arbeitsspeicher zugegriffen werden kann. Beim ersten Zugriff auf eine Seite liegt diese noch nicht im Cache (Cache-Fault), wodurch der Zugriff langsam ist. Danach kann deutlich schneller auf die Seite zugegriffen werden. Gehen Sie im Folgenden davon aus, dass vor jeder Anfrage an Ihre Datenstruktur der Cache geleert wird.

b. Das Array A sei linear im Arbeitsspeicher abgelegt. Beschreiben Sie einen Algorithmus, der *predecessor* und *predecessorIndex* für eine Zahl mit $\mathcal{O}(\log n)$ Cache-Faults bestimmen kann. Begründen Sie die Anzahl der Cache-Faults kurz. [2 Punkte]

Lösung

Führe binäre Suche auf dem Array aus. Bei jedem Zugriff tritt ein Cache-Fault auf, also insgesamt $\mathcal{O}(\log(n/B)) \in \mathcal{O}(\log(n))$ Cache-Faults.

c. Beschreiben Sie eine Datenstruktur, die *predecessor* und *predecessorIndex* für eine Zahl mit $\mathcal{O}(\log_B n)$ Cache-Faults bestimmen kann. B sei hier eine Eingabegröße, also im $\mathcal{O}$-Kalkül nicht irrelevant. Zur Vereinfachung sei n eine Potenz von B, also $\log_B n \in \mathbb{N}$. Die Datenstruktur darf maximal $\mathcal{O}(n)$ Platz benötigen. Begründen Sie den Platzbedarf und die Anzahl der Cache-Faults kurz.
Hinweis: $\sum_{k=0}^{a} q^k = \frac{1-q^{a+1}}{1-q}$ für $q \neq 1$. [4 Punkte]

Lösung

Speichere in einem zweiten Array A' jeweils den ersten Array-Eintrag von A für jede Seite. A' hat also die Größe n/B. Ein Predecessor-Query auf A' würde ergeben, auf welcher Seite man nach dem eigentlichen Predecessor suchen muss. Führe nun diese Idee rekursiv aus, also erstelle ein Array A'', das jeweils die ersten Array-Einträge jeder Seite von A' enthält, usw.
Dadurch ergibt sich implizit ein B-ärer Suchbaum der Tiefe $\mathcal{O}(\log_B n)$. Für eine Anfrage wird der Baum genau einmal bis zu einem Blatt traversiert. In jeder Ebene ergibt sich genau ein Cache-Fault, da die Ebene darüber für jede Page ein "Sample" abgespeichert hat. Insgesamt ist also die Anzahl der Cache-Faults pro Anfrage $\mathcal{O}(\log_B n)$. Der Speicherplatz ergibt sich aus der Summe der Größen aller Hilfs-Arrays:

$$\sum_{k=0}^{\log_B n} B^k = \frac{1-Bn}{1-B} = n\frac{B-1/n}{B-1} \leq 2n \in \mathcal{O}(n)$$

d. Sie haben nun Zugriff auf ein Bit-Array der Länge U, in dem für jede Zahl $x \in A$ das Bit an Position x auf `1` gesetzt ist. Alle anderen Bits sind auf `0` gesetzt. Zusätzlich haben Sie Zugriff auf eine rank und select Datenstruktur auf dem Bit-Array, die Anfragen in konstanter Zeit beantwortet. Geben Sie an, wie Sie damit die Operationen *predecessor* und *predecessorIndex* in konstanter Zeit beantworten können. [2 Punkte]

Lösung

$predecessorIndex(x) = rank_1(x)$
$predecessor(x) = select_1(rank_1(x))$

Lösungsvorschlag

EK

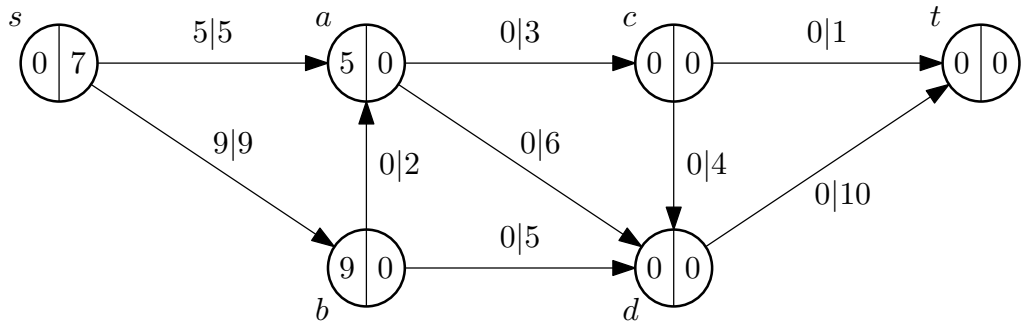
ZK

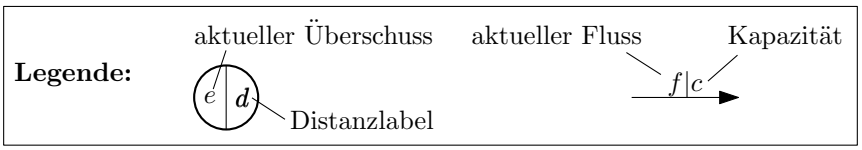
Aufgabe 4. Flussalgorithmen [10 Punkte]

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 aktuellen Überschuss und ihrem Distanzlabel beschriftet. Die Kanten sind mit ihrem aktuellen Fluss und ihrer Kapazität beschriftet.
Berechnen Sie mittels des *generic preflow-push* Algorithmus einen maximalen Fluss f, indem Sie eine Operationenfolge mit folgenden beiden Operationen angeben:

- `push`$((u,v),x)$: schiebe x Fluss über die Kante (u,v).
- `relabel`(v,d): setze Distanzlabel von Knoten v auf d.

Sie dürfen den Algorithmus abbrechen, sobald der Fluss bei t maximal ist. [4 Punkte]





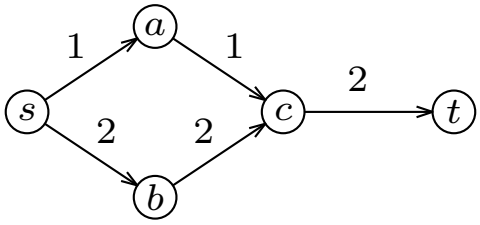
Lösung

1. `relabel`$(b,1)$	6. `push`$((a,d),6)$	11. `relabel`$(c,1)$
2. `push`$((b,d),5)$	7. `push`$((a,c),1)$	12. `push`$((c,t),1)$
3. `push`$((b,a),2)$	8. `relabel`$(d,1)$	13. ...
4. `relabel`$(a,1)$	9. `push`$((d,t),10)$	

b. Gegeben sei ein Flussnetzwerk $G = (V, E, c, s, t)$ mit maximalem Fluss f.
Beweisen oder widerlegen Sie: für jede saturierte Kante $e \in E$ existiert ein minimaler st-Schnitt $E' \subseteq E$ mit $e \in E'$. [3 Punkte]

Lösung

Gegenbeispiel:



Der maximale Fluss

$$f(e) = \begin{cases} 1, & e \in \{sa, sb, ac, bc\} \\ 2, & e = ct \end{cases}$$

saturiert die Kanten sa und ac, der eindeutig kleinste Schnitt besteht aber nur aus der Kante ct.

c. Gegeben sei eine Cuckoo-Hashtabelle $\mathscr{H}$ der Größe m mit den beiden Hashfunktionen $h_1, h_2 : \mathscr{U} \to \{1, \ldots, m\}$, die Elemente aus einem Universum $\mathscr{U}$ auf die Positionen $1, \ldots, m$ der Hashtabelle abbilden. Die Hashtabelle speichert an jeder Position maximal ein Element. Weiter sei $S \subseteq \mathscr{U}$ eine endliche Menge der Größe n.
Geben Sie einen Algorithmus mit Laufzeit $\mathscr{O}\big((n+m)\sqrt{n+m}\big)$ an, der entscheidet, ob alle Elemente in S gleichzeitig in $\mathscr{H}$ eingefügt werden können. Falls keine solche Zuordnung existiert, soll der Algorithmus das ausgeben; andernfalls soll er eine konkrete Zuordnung der Elemente auf die Hashtabellenpositionen ausgeben. Begründen Sie die Laufzeit Ihres Algorithmus kurz.
Hinweis: Modellieren Sie das Problem als Flussproblem. [3 Punkte]

Lösung

Wir modellieren das Problem als Matching im ungewichteten, bipartiten Graphen, indem jedes Element sowie jeder Platz in der Hashtabelle als Knoten abgebildet wird. Eine mögliche Zuordnung eines Elements zu einem Hashtabellenplatz wird als Kante vom Element zum Hashtabellenplatz dargestellt. Ein kardinalitätsmaximales Matching kann nun zum Beispiel mit Dinitz Algorithmus berechnet werden, indem zusätzlich eine Kante von einer Quelle s zu jedem Element und von jedem Hashtabellenplatz zu einer Senke t eingefügt wird. Alle Kanten bekommen Kapazität 1. Da das Flussnetzwerk also uniforme Kantengewichten hat und jeder Knoten Eingangs- oder Ausgangsgrad 1 hat, beträgt die Laufzeit von Dinitz Algorithmus gerade $\mathscr{O}\big((N+M)\sqrt{N}\big)$, wobei N die Anzahl der Knoten (hier $n+m$) und M die Anzahl der Kanten (hier $\leq 2n$) ist. Dadurch ergibt sich die geforderte Laufzeit $\mathscr{O}\big((n+m)\sqrt{n+m}\big)$. Wenn das maximale Matching kleiner als n ist, existiert keine geforderte Zuordnung und der Algorithmus gibt *Keine Zuordnung* aus. Andernfalls repräsentieren die Kanten im Matching die geforderte Zuordnung.

Lösungsvorschlag

Aufgabe 5. Stringology: LZ77 und Fibonacci-Wörter [9 Punkte]

Sei $n \in \mathbb{N}$ und $\Sigma = \{\texttt{a}, \texttt{b}\}$, dann ist das n-te *Fibonacci-Wort* $F_n \in \Sigma^*$ definiert als

$$F_n = \begin{cases} \texttt{a} & n = 1 \\ \texttt{ab} & n = 2 \\ F_{n-1}F_{n-2} & n > 2 \end{cases}$$

Die ersten vier Fibonacci-Wörter lauten: $F_1 = \texttt{a}$, $F_2 = \texttt{ab}$, $F_3 = \texttt{aba}$ und $F_4 = \texttt{abaab}$.

a. Geben Sie das Suffix-Array und das LCP-Array für F_5 an. Beachten Sie, dass Fibonacci-Wörter kein Sentinel (\$) haben. [3 Punkte]

Lösung

T	a	b	a	a	b	a	b	a
SA (1-basiert)	8	3	6	1	4	7	2	5
SA (0-basiert)	7	2	5	0	3	6	1	4
LCP	0	1	1	3	3	0	2	2

b. Markieren Sie die LZ77-Faktoren von F_6 und F_7.

$F_6 = \texttt{abaababaabaab}$ $F_6 = \texttt{abaababaabaab}$

$F_7 = \texttt{abaababaabaababaababa}$ $F_7 = \texttt{abaababaabaababaababa}$

[2 Punkte]

Lösung

- $F_6 = \texttt{a}\,|\,\texttt{b}\,|\,\texttt{a}\,|\,\texttt{aba}\,|\,\texttt{baaba}\,|\,\texttt{ab}$
- $F_7 = \texttt{a}\,|\,\texttt{b}\,|\,\texttt{a}\,|\,\texttt{aba}\,|\,\texttt{baaba}\,|\,\texttt{ababaaba}\,|\,\texttt{ba}$

c. Zeigen Sie, dass für die Anzahl $z(F_n)$ an LZ77-Faktoren von F_n gilt

$$z(F_n) \leq n.$$

[4 Punkte]

Lösung

Wir möchten zeigen, dass $z(F_n) \leq n$. Dies tun wir per vollständiger Induktion.

Induktionsanfang: Für $F_1 = a$ und $F_2 = ab$ ist offensichtlich, dass die Behauptung gilt.

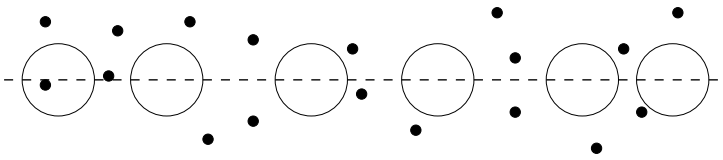
Induktionsvoraussetzung: $z(F_n) \leq n$

Induktionsbeweis: Es gilt, dass $z(F_{n+1}) = z(F_nF_{n-1})$. Wir wissen zudem, dass $z(F_n) \leq n$. Da $F_n = F_{n-1}F_{n-2}$ ein Präfix von F_{n+1} ist, können wir zudem das Suffix F_{n-1} von F_{n+1} durch einen Faktor ersetzten (falls notwendig). Es kommt also nie mehr als ein weiterer Faktor hinzu, wenn wir n um eins erhöhen. Somit gilt die Ungleichung.

Lösungsvorschlag

Aufgabe 6. Geometrische Algorithmen: Stormtrooper [10 Punkte]

Der Superbösewicht Doktor Meta möchte die Weltherrschaft übernehmen. Dazu heuert er eine Armee aus n Stormtroopern an. Für ein erstes Schießtraining werden n Zielscheiben mit jeweils Radius 1 Meter auf der gleichen Höhe an der Wand befestigt. Die Zielscheiben sind gegeben durch ihre Position auf der x-Achse und können sich berühren, aber nicht überlappen. Jeder Stormtrooper gibt nun 3 Schüsse ab. Die Einschusslöcher können als Punkte mit deren jeweiligen Koordinaten modelliert werden. Ein Schuss auf den Rand einer Zielscheibe zählt als Treffer. Die Zielscheiben und die Einschusslöcher seien in zwei nach x-Koordinate sortierten Listen gegeben. Die untenstehende Grafik illustriert beispielhaft die Situation.



a. Beschreiben Sie einen Algorithmus, der für eine sortierte Liste an Zielscheiben Z und eine sortierte Liste an Schüssen S in Zeit $\mathcal{O}(n)$ berechnet, wie viele der Schüsse auf eine der Zielscheiben getroffen haben. Sie haben eine Funktion $istTreffer(z,s)$ gegeben, die für eine Zielscheibe z und einen Schuss s in konstanter Zeit entscheidet, ob dieser getroffen hat. [2 Punkte]

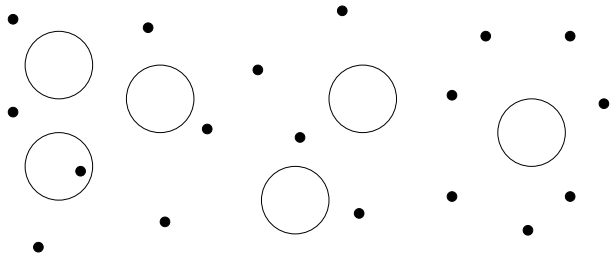
Lösung

Gehe analog vor zum Merging von zwei sortierten Listen. Benutze beide Listen als Queues, bei denen jeweils das erste Objekt betrachtet werden kann. Solange noch beide Queues nicht-leer sind, prüfe folgendes:

- Ist die x-Koordinate des ersten Schusses kleiner als der linke Rand der ersten Zielscheibe, entferne den Schuss aus der Queue.
- Sonst: Ist der rechte Rand der ersten Zielscheibe kleiner als der erste Schuss, entferne die Zielscheibe aus der Queue.
- Sonst: Möglicher Treffer. Falls es ein Treffer ist (Funktion *istTreffer*), inkrementiere die Gesamtzahl der Treffer. Entferne dann in jedem Fall den Schuss aus der Queue.

In jeder Iteration wird entweder eine Zielscheibe oder ein Schuss entfernt. Die Überprüfungen sind in konstanter Zeit möglich. Dadurch ergibt sich die lineare Laufzeit.

Die Zielscheiben seien nun frei auf der Wand verteilt. Sie sind definiert durch ihre x- und y-Position. Sie können sich weiterhin nicht überlappen. Die untenstehende Grafik illustriert beispielhaft die Situation.

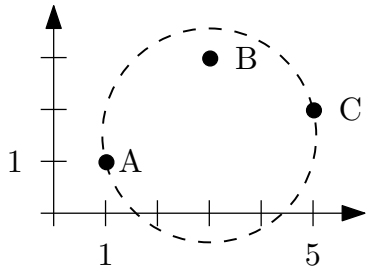


b. Beschreiben Sie einen Algorithmus, der für eine Liste an Zielscheiben Z und eine Liste an Schüssen S in Zeit $\mathcal{O}(n \log n)$ berechnet, wie viele der Schüsse auf eine der Zielscheiben getroffen haben. Sie haben eine Funktion *istTreffer*(z,s) gegeben, die für eine Zielscheibe z und einen Schuss s in konstanter Zeit entscheidet, ob dieser getroffen hat. [4 Punkte]

Lösung

Verwende einen Sweepline-Algorithmus mit 3 Event-Typen: Start einer Zielscheibe, Ende einer Zielscheibe, und Schuss. Die Events werden in sortierter x-Reihenfolge verarbeitet. Die Sweepline selbst hält eine Liste der aktiven Zielscheiben sortiert nach ihrer y-Koordinate. Bei einem Start- oder End-Event wird die jeweilige Zielscheibe in $\mathcal{O}(\log n)$ eingefügt oder gelöscht. Bei einem Schuss-Event müssen nun alle aktiven Zielscheiben auf *istTreffer* getestet werden, deren y-Koordinate einen Abstand von maximal 1 Meter vom Schuss haben. Da sich die Zielscheiben nicht überlappen können, gibt es maximal 4 Kandidaten, welche durch Suche in der sortierten Sweepline in Zeit $\mathcal{O}(\log n)$ gefunden werden können. Die Schnitt-Berechnung (sowie möglicherweise Erhöhung der Treffer-Anzahl) ist dann in konstanter Zeit möglich.
Insgesamt ergeben sich $2n$ Zielscheiben-Events und $3n$ Schuss-Events, also $\mathcal{O}(n)$ Events. Die Abarbeitung jedes Events benötigt Zeit $\mathcal{O}(\log n)$, wodurch sich (auch inklusive Sortieren der Events) die geforderte Laufzeit von $\mathcal{O}(n \log n)$ ergibt.

Doktor Meta ist enttäuscht von der Leistung seiner Stormtrooper. Die Stormtrooper schießen zwar schlecht, aber sie schießen zuverlässig immer an die gleiche Position — egal wo die Zielscheiben hängen. Um den Kauf der Stormtrooper dennoch vor seinen Investoren rechtfertigen zu können, möchte er eine große Zielscheibe kaufen, sodass alle Schüsse diese treffen.



c. Führen Sie schrittweise den *Smallest Enclosing Ball* Algorithmus (SEB) aus der Vorlesung mit den Punkten aus der obenstehenden Grafik aus. Für die rekursiven Aufrufe ist bereits die nötige Struktur vorgegeben und alle Aufruf-Parameter bis zum ersten *return* sind bereits eingetragen. Tragen Sie jeweils die restlichen verwendeten Parameter sowie den Rückgabewert in die Kästchen ein. Kennzeichnen Sie leere Mengen explizit mit ∅. Wählen Sie als zufälligen Punkt immer den mit lexikographisch kleinstem Namen.

[4 Punkte]

Lösung