Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Hans-Peter Lehmann, Daniel Seemaier

2. Übungsblatt zu Algorithmen II im WS 2021/2022

http://algo2.iti.kit.edu/AlgorithmenII_WS21.php
{sanders, hans-peter.lehmann, daniel.seemaier}@kit.edu

Musterlösungen

Aufgabe 1 (*Rechnen: Monotone ganzzahlige* Priority Queues)

Bei einer Ausführung von *Dijkstra's Algorithmus* wird folgender Ausschnitt an *Priority Queue* Operationen protokolliert:

...

- `insert(a, 06 [00110] )` (Parameter: Knotenbezeichnung, Distanz [Distanz binär])
- `insert(b, 10 [01010] )`
- `insert(c, 07 [00111] )`
- `deleteMin()`
- `deleteMin()`
- `insert(d, 12 [01100] )`
- `deleteMin()`
- `insert(e, 16 [10000] )`

...

Zusätzlich wissen Sie, dass das maximale Kantengewicht im Graphen $C = 6$ beträgt und dass vor der ersten protokollierten Operation das letzte enthaltene Element aus der *Priority Queue* entfernt wurde. Dieses hatte den Wert $min = 5$.

a) Führen Sie die Operationen auf einer *Bucket Queue* aus. Geben Sie den Zustand der Datenstruktur nach jeder Operation an.

b) Wieviele *Buckets* werden für eine Ausführung auf einem *Radix Heap* benötigt? Führen Sie die Operationen auf einem *Radix Heap* aus. Geben Sie den Zustand der Datenstruktur und den Wertebereich der *Buckets* nach jeder Operation an.

Musterlösung:

a) *Bucket Queue*:

`insert(a, 06 [00110] ):`

0	1	2	3	4	5	6
						(a, 6)

min = 5

`insert(b, 10 [01010] ):`

0	1	2	3	4	5	6
			(b, 10)			(a, 6)

min = 5

`insert(c, 07 [00111] ):`

(für monotone *Priority Queues* nur wichtig, dass Elemente aus $[min, min + C]$ stammen!)

0	1	2	3	4	5	6
(c, 7)			(b, 10)			(a, 6)

min = 5

`deleteMin():`

0	1	2	3	4	5	6
(c, 7)			(b, 10)			

min = 6

`deleteMin():`

0	1	2	3	4	5	6
			(b, 10)			

min = 7

`insert(d, 12 [01100] ):`

0	1	2	3	4	5	6
			(b, 10)		(d, 12)	

min = 7

`deleteMin():`

0	1	2	3	4	5	6
					(d, 12)	

min = 10

`insert(e, 16 [10000] ):`

0	1	2	3	4	5	6
		(e, 16)			(d, 12)	

min = 10

Musterlösung:

b) *Radix Heap*:
(Es werden $K+2$ *Buckets* benötigt: $B[-1]$, $B[0]$, …, $B[K]$; mit $K = 1 + \lfloor \log_2 C \rfloor = 3$ ergeben sich 5 *Buckets*.)

`insert(a, 06 [00110] )`:

-1	0	1	2	3
		(a, 06 [00110])		
5	–	6–7	–	8–11

min = 5 [00101]

`insert(b, 10 [01010] )`:

-1	0	1	2	3
		(a, 06 [00110])		(b, 10 [01010])
5	–	6–7	–	8–11

min = 5 [00101]

`insert(c, 07 [00111] )`:

-1	0	1	2	3
		(a, 06 [00110]) (c, 07 [00111])		(b, 10, [01010])
5	–	6–7	–	8–11

min = 5 [00101]

`deleteMin()`:
($B[1]$ ist der erste gefüllte *Bucket*; min wird auf das kleinste enthaltene Element (6) gesetzt; die Elemente in $B[1]$ werden neu verteilt und anschließend das Element in $B[-1]$ entfernt)

-1	0	1	2	3
	(c, 07 [00111])			(b, 10 [01010])
6	7	–	–	8–12

min = 6 [00110]

`deleteMin()`:

-1	0	1	2	3
				(b, 10 [01010])
7	–	–	–	8–13

min = 7 [00111]

`insert(d, 12 [01100] )`:

-1	0	1	2	3
				(b, 10 [01010]) (d, 12 [01100])
7	–	–	–	8–13

min = 7 [00111]

`deleteMin()`:

-1	0	1	2	3
			(d, 12 [01100])	
10	11	–	12–15	16

min = 10 [01010]

`insert(e, 16 [10000] )`:

-1	0	1	2	3
		3	(d, 12 [01100])	(e, 16 [10000])
10	11	–	12–15	16

min = 10 [01010]

Aufgabe 2 (*Analyse: Laufzeit von Dijkstra's Algorithmus*)

Gegeben sei ein gerichteter Graph $G = (V,\ E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtungsfunktion $c : E \to \mathbb{R}_0^+$.

a) Beweisen Sie die Behauptung aus der Vorlesung, dass für $m = \Omega(n \log n \log\log n)$ Dijkstra's Algorithmus mit einem *binary heap* eine durchschnittliche Laufzeit von $O(m)$ besitzt.

b) Eine spezielle *Priority Queue* habe folgende Laufzeiteigenschaften:

- `insert`: $O(\log n)$
- `decreaseKey`: $O(1)$
- `deleteMin`: $O(\sqrt{m})$

(ob eine Datenstruktur mit diesen Eigenschaften existiert und Dijkstra's Algorithmus mit ihr korrekt arbeitet, ist eine andere Frage, aber wir nehmen für diese Aufgabe an es ginge :-))

Geben Sie eine kleinste obere Schranke für die Laufzeit von Dijkstra's Algorithmus unter Verwendung dieser *Priority Queue* an. Unter welcher Bedingung an das Verhältnis der Anzahl Knoten n zu Kanten m wird die Laufzeit linear in der Eingabegröße? Die Eingabe erfolgt in Form einer Adjazenzliste.

Musterlösung:

a) Für die durchschnittliche Laufzeit von Dijkstra's Algorithmus mit einem *binary heap* gilt:

$$O(m + n \log \frac{m}{n} \log n)$$

Zu zeigen ist, ob diese Laufzeit in $O(m)$ liegt für die gegebene Wahl von $m = \Omega(n \log n \log \log n)$. Wähle den kleinstmöglichen Wert für m. Falls die Aussage diesen Wert gilt, gilt Sie sicher auch für alle größeren m. Eingesetzt und umgeformt ergibt sich:

$$\begin{aligned}
& O(n \log n \log \log n + n \log \frac{n \log n \log \log n}{n} \log n) \\
\overset{\text{kürzen}}{=} \quad & O(n \log n \log \log n + n \log(\log n \log \log n) \log n) \\
\overset{\log ab = \log a + \log b}{=} \quad & O(n \log n \log \log n + n \log n \log \log n + n \log \log \log n \log n) \\
\overset{\log \log \log n = O(\log \log n)}{=} \quad & O(n \log n \log \log n) \\
= \quad & O(m)
\end{aligned}$$

Damit liegt die Laufzeit in $O(m)$.

Für eine geringere Abhängigkeit, z.B. $m = O(n)$ würde der zweite Term den ersten im O-Kalkül dominieren und die Umformung würde nicht zu $O(m)$ führen.

b) Allgemein gilt für die Laufzeit von Dijkstra's Algorithmus:

$$O(m + m \cdot T_{decreaseKey}(n) + n \cdot (T_{deleteMin}(n) + T_{insert}(n)))$$

Mit den angegebenen Laufzeiten eingesetzt ergibt sich:

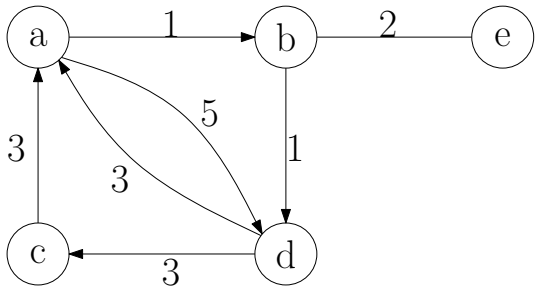
$$O(m + n\sqrt{m} + n \log n)$$

Unter den Forderungen $n \cdot \sqrt{m} = O(m)$ und $n \log n = O(m)$ ist die Laufzeit linear in m. Dies lässt sich umformen zu $\Omega(n) = \sqrt{m}$ und $\Omega(n \log n) = m$. Damit ergibt sich $m = \Omega(n^2)$.

Aufgabe 3 (*Einführung+Analyse: Bidirektionaler Dijkstra*)

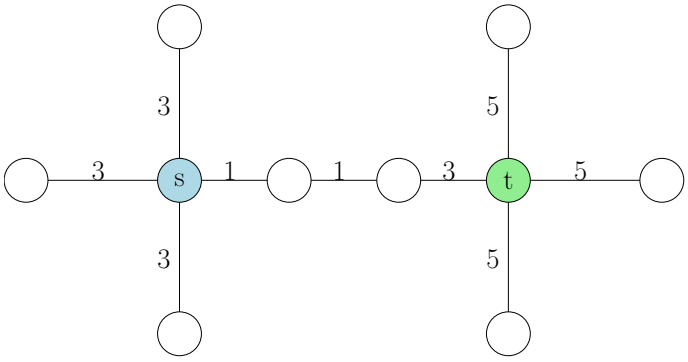
Gegeben sei –wie üblich– ein gerichteter Graph $G = (V,\ E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtungsfunktion $c : E \to \mathbb{R}_0^+$. Gesucht ist der kürzeste Pfad $p = \langle s, \ldots, t \rangle$ zwischen zwei Punkten $s,\ t \in V$.
Eine bidirektionale Suche löst dieses Problem wie folgt: Es werden zwei unidirektionale Suchen mit Dijkstra's Algorithmus gestartet. Die *Vorwärtssuche* beginnt bei Knoten s und operiert auf dem normalen Graphen G, auch *Vorwärtsgraph* genannt. Die *Rückwärtssuche* beginnt bei Knoten t und operiert auf dem *Rückwärtsgraph* $G^r = (V, E^r)$ mit Kantengewichtungsfunktion c^r. Dieser Graph entsteht aus G durch Umkehrung aller Kanten. Der Algorithmus scannt abwechselnd einen Knoten in der Vorwärtssuche und in der Rückwärtssuche, beginnend mit der Vorwärtssuche.
Wird während des Scans von Knoten u Kante $(u,\ v)$ relaxiert, so wird überprüft, ob die Distanz $d_{\text{forward}}[v] + d_{\text{backward}}[v]$ kleiner ist als die momentan minimale gefundene Distanz von s nach t und diese gegebenenfalls angepasst ($d_{\text{forward}}[v]$ gibt die bisher kürzeste gefundene Distanz von s nach v in der Vorwärtssuche und $d_{\text{backward}}[v]$ die bisher kürzeste gefundene Distanz von v nach t in der Rückwärtssuche an).
Sobald ein Knoten in einer Richtung gescannt werden soll, der bereits in der anderen Richtung gescannt worden ist, kann die Suche beendet werden (*Abbruchbedingung*). Die aktuelle minimale gefundene Distanz ist dann die tatsächliche minimale Distanz zwischen s und t.

a) Zeichnen Sie den Rückwärtsgraph G^r zum angegebenen Graphen. Geben Sie die Kantengewichte $c(a, d)$, $c^r(a, d)$ sowie $c(b, e)$, $c^r(b, e)$ an.



(Kante (b, e) ist eine bidirektionale [bzw. ungerichtete] Kante)

b) Geben Sie an, in welcher Reihenfolge der unten angegebene Graph durchlaufen wird.

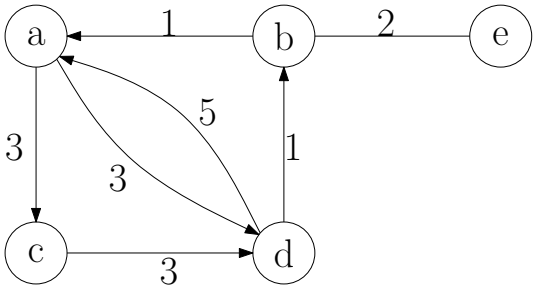


c) Zeigen Sie, dass die Abbruchbedingung korrekt ist.

d) Wann kann es passieren, dass die Suche nach dem Scan von Knoten u beendet wird, dieser aber nicht Teil des kürzesten Weges ist? Geben Sie ein Beispiel an.

Musterlösung:

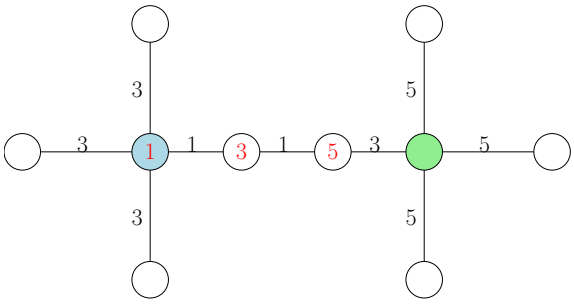
a) Rückwärtsgraph G^r:



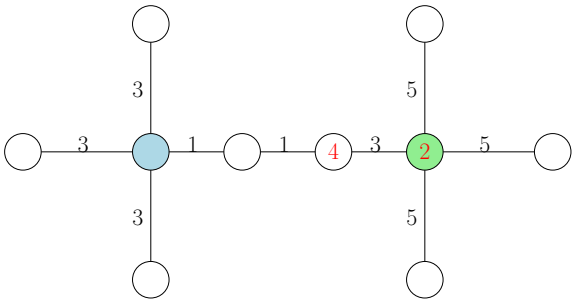
Es sind einfach alle Pfeile umgedreht worden.

Kantengewichte:
$c(a,d) = 5$, $c^r(a,d) = 3$
$c(b,e) = 2$, $c^r(b,e) = 2$
Allgemein gilt $c(u,v) = c^r(v,u)$.

b) Vorwärtssuche:



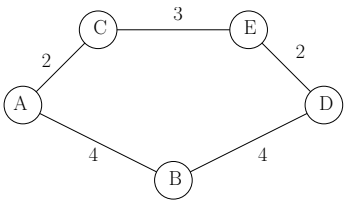
Rückwärtssuche:



Die Zahlen in den Knoten geben die Reihenfolge an, in der Sie gescannt worden sind. Sobald die Vorwärtssuche den Knoten mit Nummer 5 scannt, ist die Suche beendet, da er schon in Rückwärtsrichtung gescannt wurde.

c) Nehmen wir an, es existiere ein Knoten u, der in beiden *Queues* gelöscht wurde, aber $d(s,t) < d(s,u) + d(u,t)$ sei noch nicht bekannt. Da die Knoten in streng monotoner Reihenfolge gescannt werden, sind in der Vorwärtssuche bereits alle Knoten v mit $d(s,v) < d(s,u)$ gescannt worden. Gleiches gilt für Knoten v mit $d(v,t) < d(u,t)$ in der Rückwärtssuche. Betrachten wir den kürzesten Pfad $p = \{s = n_1, \ldots, n_k = t\}$. Weiterhin betrachten wir den Knoten mit maximalem i, so dass $d(s,n_i) < d(s,u)$ sowie den Knoten mit minimalem j, so dass $d(n_j,t) < d(u,t)$. Da $d(s,t)$ noch nicht bekannt ist, muss gelten: $i < j - 2$ (sonst wäre die Distanz bekannt). Folglich existiert aber ein Knoten n_x in p mit $d(s,n_x) \geq d(s,u)$ sowie $d(n_x,t) \geq d(u,t)$. Damit wäre aber auch $d(s,t) \geq d(s,u) + d(u,t) > d(s,t)$, was ein Widerspruch ist.

d) Der abgebildete Graph ist ein mögliches Beispiel.



Die Vorwärtssuche bearbeitet die Knoten in der Reihenfolge A,C,B,E,D. Die Rückwärtssuche bearbeitet die Knoten in der Reihenfolge D,E,B,C,A. Nach drei abwechselnden Schritten wurde B folglich in beiden Suchräumen gescannt. Der kürzeste Weg folgt aber der Route A,C,E,D.

Aufgabe 4 (*Rechnen+Analyse: Suche in Strings*)

a) Zur Ausführung des KMP-Algorithmus muss zunächst ein sogenanntes *border-array* berechnet werden. Geben Sie das *border-array* für das Suchmuster $P =$ `abacababc` an.

b) Führen Sie den KMP-Algorithmus auf dem Text $T =$ `abacababbabacababc` mit obigem Suchmuster durch.

c) Wie oft muss der KMP-Algorithmus ein Muster P der Länge $|P|$ maximal an einen Text T der Länge $|T|$ anlegen, falls P nicht in T vorkommt? Wie oft minimal? Geben Sie das Ergebnis in Abhängigkeit von $|P|$ und $|T|$ an. Geben Sie außerdem jeweils ein Beispiel für P und T an.

d) Zeigen oder widerlegen Sie:
Das *border-array* kann keine drei aufeinanderfolgenden Einträge enthalten, die jeweils um eins kleiner sind als ihr Vorgänger, falls der erste von diesen drei Einträgen auf ein Suffix der Größe $k \geq 3$ verweist, welches gleichzeitig echtes Präfix ist.

Beispiel: $\text{border}[10] = 3,\ \text{border}[11] = 2,\ \text{border}[12] = 1$

Kein Beispiel: $\text{border}[10] = 2,\ \text{border}[11] = 1,\ \text{border}[12] = 0$

Musterlösung:

a) Es ergibt sich folgendes *border-array*:

P	a	b	a	c	a	b	a	b	c
border[]	-1	0	0	1	0	1	2	3	2

b) Das Suchmuster wird insgesamt 4 mal angelegt:

T	a	b	a	c	a	b	a	b	b	a	b	a	c	a	b	a	b	c	
	a	b	a	c	a	b	a	b	c										Stelle 1
							a	b	a	c	a	b	a	b	c				Stelle 7
									a	b	a	c	a	b	a	b	c		Stelle 9
										a	b	a	c	a	b	a	b	c	Stelle 10

c) Das Muster muss maximal $|T| - |P| + 1$ mal angelegt werden. Dieser Fall tritt z.B. auf, falls das erste Zeichen von P nicht in T auftaucht. Das Muster muss minimal $\lceil |T|/(|P| - 1) \rceil$ mal angelegt werden. Dieser Fall tritt z.B. auf, falls P aus einer Folge paarweise unterschiedlicher Zeichen besteht und T aus Konkatenationen von P ohne das letzte Zeichen.

d) Zu zeigen oder widerlegen ist die Existenz von drei Einträgen:

$\text{border}[i] = k,\ \text{border}[i+1] = k-1, \text{border}[i+2] = k-2.$

Einträge dieser Art können nicht existieren, da in diesem Fall gelten würde:

für $\text{border}[i+0] = k-0$: $P_{k-2} = P_{i-3},\ P_{k-1} = P_{i-2},\ P_k = P_{i-1}$,
für $\text{border}[i+1] = k-1$: $P_{k-2} = P_{i-1},\ P_{k-1} = P_i$ und
für $\text{border}[i+2] = k-2$: $P_{k-2} = P_{i+1}$.

Damit würde sich ergeben:

$P_{k-2} = P_{i-3} = P_{i-1} = P_{i+1} = P_k$,
$P_{k-1} = P_{i-2} = P_i$.

In diesem Fall wäre aber $\text{border}[i+2] = k$, da
$P_{k-2} = P_{i-1} = P_{i-3},\ P_{k-1} = P_i = P_{i-2}$ und $P_k = P_{i+1} = P_{i-1}$.
Andernfalls wäre auch $\text{border}[i+0] \neq k$.

Aufgabe 5 (*Rechnen+Analyse: LCP-Array*)

Gegeben sei die Zeichenkette $s =$ `salsadipp$`.

a) Geben Sie den Suffixbaum für s an.

b) Geben Sie das Suffixarray für s an.

c) Geben Sie das LCP-Array für s an.

Im Folgenden sei ein String T sowie dessen Suffixarray SA[·] und dessen LCP-Array LCP[·] gegeben.

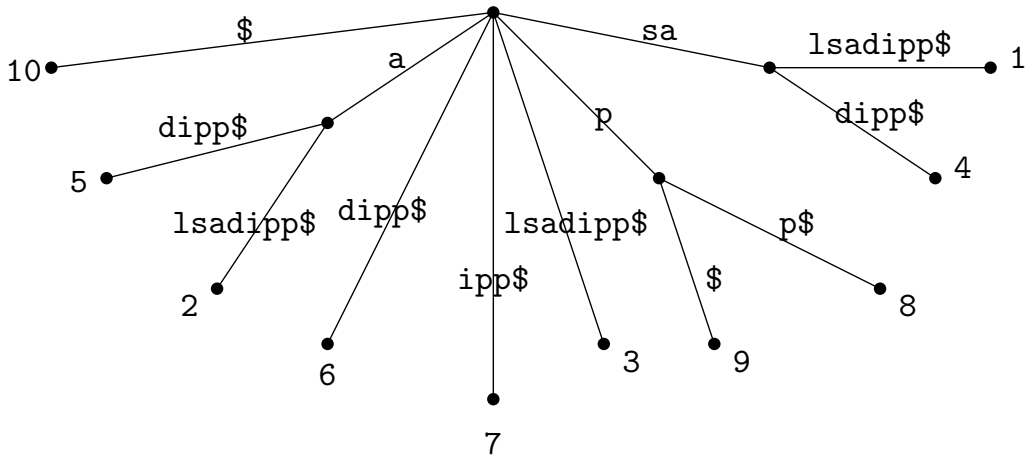
d) Wie kann der längste sich wiederholende Substring in T effizient bestimmt werden?
(der Substring darf sich dabei selbst überlappen)

e) Wie viele paarweise unterschiedliche Substrings kann ein String der Länge n maximal besitzen? Wie kann die tatsächliche Anzahl für einen konkreten String T bestimmt werden?

f) Ein String lässt sich schlecht komprimieren, wenn er wenig Redundanz besitzt. Ein Maß dafür ist die Anzahl paarweise unterschiedlicher Substrings normiert auf die mögliche Gesamtanzahl unterschiedlicher Substrings. Geben Sie an, wie dieses Maß für T berechnet werden kann.

Musterlösung:

a) Der Suffixbaum für s als kompaktierter Trie:



Jeder Weg von der Wurzel zu einem Blatt gibt einen Suffix von s an. Der Wert am Blatt gibt den Index des Suffix an (d.h. die Stelle in der Zeichenkette an der der Suffix beginnt).

b) Das Suffixarray SA für s lautet:
SA $= \langle 10, 5, 2, 6, 7, 3, 9, 8, 4, 1 \rangle$.

Index i	SA$[i]$	$s_{\mathrm{SA}[i]}$
1	10	`$`
2	5	`adipp$`
3	2	`alsadipp$`
4	6	`dipp$`
5	7	`ipp$`
6	3	`lsadipp$`
7	9	`p$`
8	8	`pp$`
9	4	`sadipp$`
10	1	`salsadipp$`

In der rechten Spalte der Suffixtabelle sind die durch das Suffixarray indizierten Suffixe aufgetragen. Man sieht, dass sie durch das Suffixarray in alphabetischer Reihenfolge geordnet vorliegen.

c) Das LCP-Array LCP für s lautet:
LCP $= \langle 0, 0, 1, 0, 0, 0, 0, 1, 0, 2 \rangle$.

Der Eintrag LCP$[i]$ gibt die Länge des gemeinsamen Präfixes von $s_{\mathrm{SA}[i-1]}$ und $s_{\mathrm{SA}[i]}$ an. Damit ist LCP[1] nicht definiert. Nach Konvention setzt man normalerweise LCP$[1] = 0$.

Musterlösung:

d) Das Suffixarray SA[·] enthält alle Suffixe von T lexikographisch sortiert. Damit sind die Einträge mit dem längstem gemeinsamen Präfix –und damit mit dem längsten Substring– benachbart. Da jeder Eintrag LCP[i] im LCP-Array die Länge des gemeinsamen Präfixes von benachbarten Suffixen $s_{\mathrm{SA}[i-1]}$ und $s_{\mathrm{SA}[i]}$ angibt, genügt es, das Maximum von LCP[·] zu bestimmen, um die Position des längsten sich wiederholenden Substring zu erhalten.

e) Die Menge aller Substrings eines Strings ist durch die Menge aller Präfixe seiner Suffixe gegeben. In einem String mit maximal vielen unterschiedlichen Substrings besitzen keine zwei Suffixe ein gemeinsames Präfix. Damit steuert jedes Suffix s genau $|s|$ zur Menge der paarweise unterschiedlichen Substrings bei. Somit ist die gesuchte Anzahl $\sum_{i=1}^{n} |s_{SA[i]}| = \sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$.

   Die Anzahl paarweiser unterschiedlicher Substrings für einen konkreten String T ist gegeben durch $\#_{pds} = \sum_{i=1}^{n} |s_{\mathrm{SA}[i]}| - \mathrm{LCP}[i]$. Dies folgt aus folgender Überlegung:
   Sei LCP[j] = k. Dies bedeutet, dass $s_{\mathrm{SA}[i-1]}$ und $s_{\mathrm{SA}[i]}$ ein gemeinsames Präfix der Länge k besitzen. Das wiederum heißt, dass die k Suffixe dieses gemeinsamen Präfix nicht gezählt werden dürfen, da sie gemeinsam und nicht paarweise verschieden sind.

   Die Länge eines Suffix lässt sich mit Hilfe des Suffix-Arrays bestimmen zu $|s_{\mathrm{SA}[i]}| = n - SA[i]$.

f) Dies lässt sich direkt aus der vorherigen Teilaufgabe ableiten: $\frac{2}{n\cdot(n+1)} \sum_{i=1}^{n} P[i]$.