Karlsruher Institut für Technologie
Institut für Theoretische Informatik

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

3. 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: Kompression*)

a) Bestimmen Sie die *Lempel-Ziv-Faktoren* $f_i = (l_i, p_i)$ des Textes $T_1 =$ `abbbababbbb$`.

b) Gegeben seien folgende *Lempel-Ziv-Faktoren*. Bestimmen Sie den Text T_2, aus dem sie entstanden sind.

$$(0, \texttt{a}), (5, 1), (0, \texttt{b}), (6, 1), (7, 7), (0, \texttt{\$})$$

c) Angenommen, jedes Zeichen kann in 1 byte, sowie jeder *Lempel-Ziv-Faktor* in 2 byte repräsentiert werden. Wie viel Prozent des Speicherplatzes von T_2 konnte somit eingespart werden?

Musterlösung:

a) Die Lempel-Ziv-Faktoren lauten:

f_1	`a`	(0, `a`)
f_2	`b`	(0, `b`)
f_3	`bb`	(2, 2)
f_4	`ab`	(2, 1)
f_5	`abbb`	(4, 1)
f_6	`b`	(1, 2)
f_7	`$`	(0, `$`)

b) $T_2 =$ `aaaaaabaaaaaabaaaaaa$`

f_1	`a`	(0, `a`)
f_2	`aaaaa`	(5, 1)
f_3	`b`	(0, `b`)
f_4	`aaaaaa`	(6, 1)
f_5	`baaaaaa`	(7, 7)
f_6	`$`	(0, `$`)

c) $|T_1| = 21$ benötigt 21 byte. $|f_i| = 6$ benötigt 12 byte. Dadurch ergibt sich eine Einsparung von $1 - 12/21 \approx 42\%$.

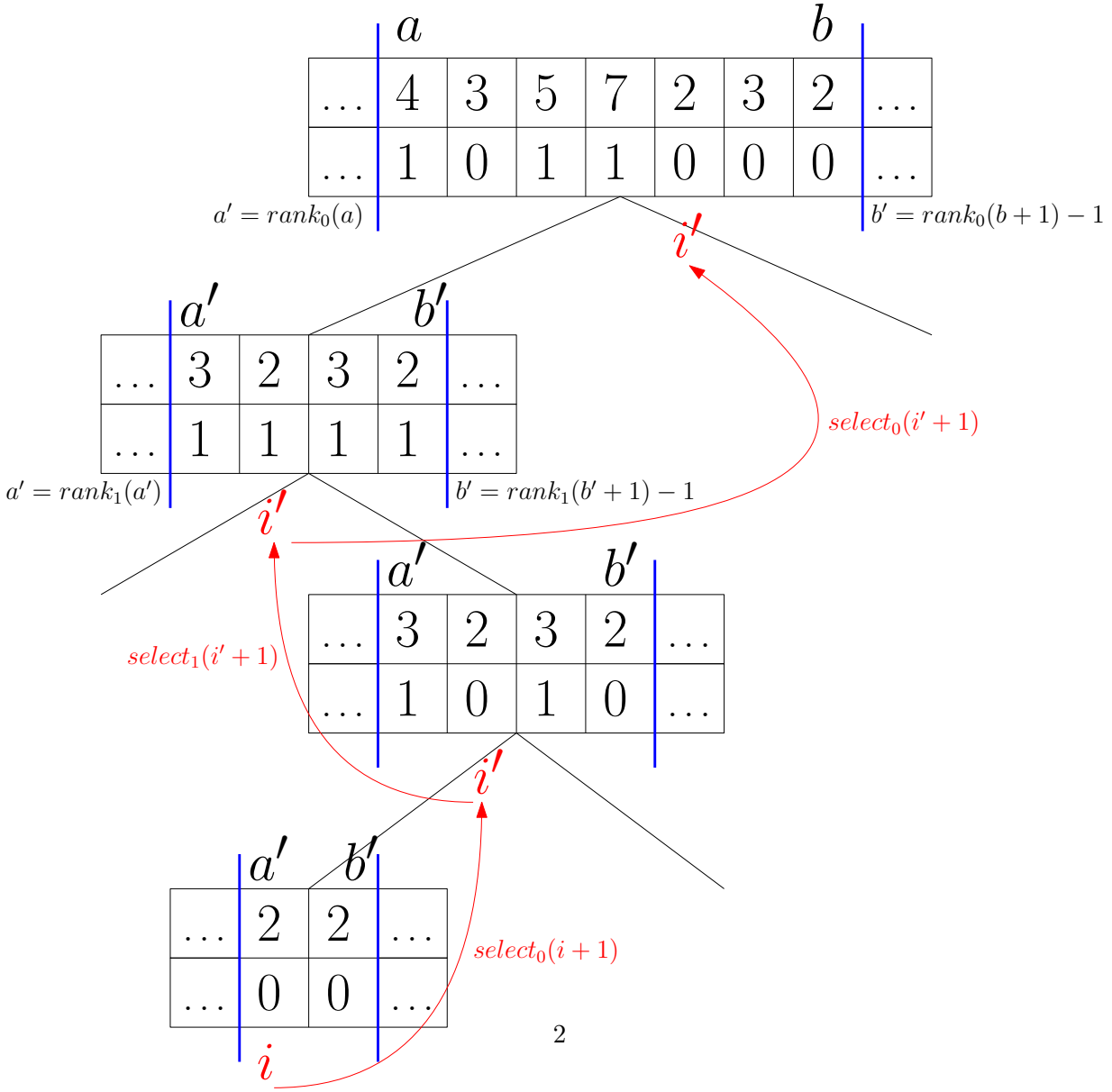
Aufgabe 2 (*RMQ in Wavelet Trees*)

Gegeben sei ein Universum von Zahlen $\mathcal{U}$ und ein Feld A von Zahlen aus $\mathcal{U}$. Geben sie einen Algorithmus an, mit dem sich unter Benutzung eines Wavelet Trees in $\log |\mathcal{U}|$ Zeit $\arg\min_i\{A[i] \mid i \in [a, b]\}$ für Parameter a, b berechnen lässt.

Musterlösung:
Sei WT der Wavelet Tree für unser Feld A.

1: **function** RMQ(WT, a, b)
2: **if** WT ist Blatt **then**
3: **return** a
4: **end if**
5: $a' \leftarrow rank_0(a)$
6: $b' \leftarrow rank_0(b+1) - 1$
7: **if** $b' - a' \geq 0$ **then** ▷ Minimum ist in linkem Teilbaum
8: $i' \leftarrow \mathsf{RMQ}(WT \rightarrow child_{left}, a', b')$
9: $i \leftarrow select_0(i'+1)$
10: **else** ▷ Minimum ist in rechtem Teilbaum
11: $i' \leftarrow \mathsf{RMQ}(WT \rightarrow child_{right}, a - a', b - b' - 1)$ ▷ $rank_1(x) = x - rank_0(x)$
12: $i \leftarrow select_1(i'+1)$
13: **end if**
14: **return** i
15: **end function**

Dieser Algorithmus ist leicht modifizierbar um das linkeste, rechteste oder mittlere Minimum auszugeben.



Aufgabe 3 (*Rechnen: Suffixarrays und DC3-Algorithmus*)

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

a) Geben Sie den Suffixbaum für s an.

b) Geben Sie das Suffixarray für s an.

In der Vorlesung haben Sie einen Linearzeitalgorithmus zur Konstruktion von Suffixarrays kennengelernt. Dieser ist unter dem Namen *DC3-Algorithmus* bekannt. Im Folgenden soll der Algorithmus Schritt für Schritt per Hand ausgeführt werden.
Die Suffixe von s werden zunächst in drei Sequenzen $C^k = \langle s_i \mid (i \bmod 3) = k \rangle$ für $k \in \{0, 1, 2\}$ aufgeteilt. Danach müssen die Sequenzen C^0 und $C^{12} = C^1 \cup C^2$ lexikographisch sortiert werden.

Sortierung von C^{12}:

c) Geben Sie die Tripelsequenzen $R^k = \langle s[i..i+2] \mid (i \bmod 3) = k \rangle$ für $k \in \{1, 2\}$ an. Für $i \geq |s|$ gelte $s[i] =$ \$ (Auffüllen mit zusätzlichen Abschlusszeichen).

d) Bestimmen Sie den *Rang* der Tripel von $R^{12} = R^1 \circ R^2$. Sortieren Sie dazu die Tripel und entfernen mehrfache Vorkommnisse. Die Position eines Tripels in dieser Sortierung gibt seinen Rang an.

e) Die berechneten Ränge definieren eine eindeutige Bezeichnung für jedes Tripel in R^{12}. Drücken Sie R^{12} mit Hilfe dieser Ränge aus. Diese Darstellung ergibt die Zeichenkette s^{12}. Muss der DC3-Algorithmus eine Rekursion ausführen?

f) Geben Sie das Suffixarray SA^{12} für s^{12} von Hand an (unabhängig, ob der DC3-Algorithmus eine Rekursion durchführt). Vergewissern Sie sich, dass SA^{12} eine Sortierung von C^{12} beschreibt.

Sortierung von C^0:

g) Erstellen Sie eine Zuordnung *rank*, die jedem i mit $s_i \in C^{12}$ den Index von s_i in der sortierten Sequenz C^{12} zuweist. Für alle anderen i sei $rank(i) = 0$.

Formale Berechnung von *rank* mit Hilfe des Suffixarray SA^{12} nach:

$$\begin{array}{lcll} rank[3 \cdot (\mathrm{SA}^{12}[i]) & +1] & = & i \quad \mathrm{SA}^{12}[i] < |C^1| \\ rank[3 \cdot (\mathrm{SA}^{12}[i] - |C^1|) + 2] & & = & i \quad \mathrm{SA}^{12}[i] \geq |C^1| \end{array}$$

Alle anderen Werte von $rank[i]$ können gleich 0 gesetzt werden.

h) Erstellen Sie Tupel $(s[i], rank[i{+}1])$ f.a. $s_i \in C^0$ und sortieren diese lexikographisch. Vergewissern Sie sich, dass diese Sortierung einer Sortierung von C^0 entspricht.

Nachdem C^0 und C^{12} sortiert worden sind, kann das Suffixarray von s bestimmt werden:

i) Führen Sie eine Mischen-Operation auf C^0 und C^{12} aus. Die resultierende Sequenz wird mit C bezeichnet. Es gelten folgende Sortierkriterien:

$$s_i \leq s_j \iff \begin{cases} (s[i], \quad rank[i+1]) & \leq & (s[j], \quad rank[j+1]) & s_j \in C^1 \\ (s[i..i+1], rank[i+2]) & \leq & (s[j..j+1], rank[j+2]) & s_j \in C^2 \end{cases}$$

Vergewissern Sie sich, dass C lexikographisch sortiert ist und damit das Suffixarray induziert.

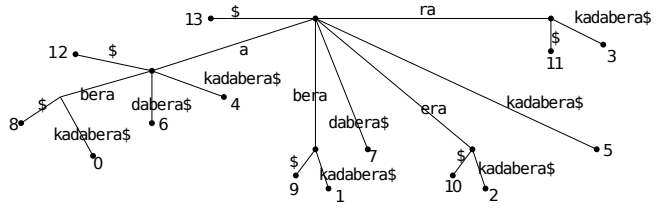
Notation:
- Alle Indizes fangen bei 0 an – analog zu Kapitel 9.3.6, auf dem die Aufgabe basiert.

- $s[i\ (..j)\ ]$: Zeichen an Stelle i (bis j) in s (z.B. $s[1..3] =$ `ber`)
- s_i: Suffix von s ab Stelle i (z.B. $s_2 =$ `erakadabera`)

Musterlösung:
Als Referenz zunächst Zeichenkette s mit ihren Indizes (beachten Sie das Abschlusszeichen `$`!):

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$s[i]$	a	b	e	r	a	k	a	d	a	b	e	r	a	$

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 $= \langle 13, 12, 8, 0, 6, 4, 9, 1, 7, 10, 2, 5, 11, 3 \rangle$.

Index i	SA$[i]$	$s_{\mathrm{SA}[i]}$
0	13	`$`
1	12	`a$`
2	8	`abera$`
3	0	`aberakadabera$`
4	6	`adabera$`
5	4	`akadabera$`
6	9	`bera$`
7	1	`berakadabera$`
8	7	`dabera$`
9	10	`era$`
10	2	`erakadabera$`
11	5	`kadabera$`
12	11	`ra$`
13	3	`rakadabera$`

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.

Musterlösung:

c) Die Tripelsequenzen lauten:

$R^1 = \langle \texttt{ber}, \texttt{aka}, \texttt{dab}, \texttt{era}, \texttt{\$\$\$} \rangle$
$R^2 = \langle \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$} \rangle$

Beachten Sie, dass das letzte Tripel von R^1 mit weiteren Abschlusszeichen \$ ergänzt wurde.

d) Die sortierten Tripel von R^{12} ohne Duplikate ergeben folgende Ränge:

Index i	4	7	1	0	2	3, 5	6	8
$R^{12}[i]$	\$\$\$	abe	aka	ber	dab	era	kad	ra\$
Rang	0	1	2	3	4	5	6	7

e) Die Ränge definieren eine eindeutige Bezeichnung der Tripel von R^{12}. Damit kann R^{12} dargestellt werden als $s^{12} = \langle 3, 2, 4, 5, 0, 5, 6, 1, 7 \rangle$.

Die Sequenz s^{12} enthält das selbe Zeichen (5) mehrfach. Ansonsten wäre über die Ränge bereits eine vollständige Sortierung von s^{12} –und damit auch von C^{12}– bestimmt. Um diese Sortierung zu erhalten, bestimmt man rekursiv mit dem DC3-Algorithmus das Suffixarray für s^{12}.

f) Das Suffixarray für R^{12} ist –nach Konstruktion– gleich dem Suffixarray für s^{12}. Es ergibt sich zu $\mathrm{SA}^{12} = \langle 9, 4, 7, 1, 0, 2, 3, 5, 6, 8 \rangle$.

Index i	$\mathrm{SA}^{12}[i]$	$s^{12}_{\mathrm{SA}^{12}[i]}$	$R^{12}_{\mathrm{SA}^{12}[i]}$
0	9	$\langle . \rangle$	$\langle\rangle$
1	4	$\langle 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
2	7	$\langle 1, 7, . \rangle$	$\langle \texttt{abe}, \texttt{ra\$}, \rangle$
3	1	$\langle 2, 4, 5, 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{aka}, \texttt{dab}, \texttt{era}, \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
4	0	$\langle 3, 2, 4, 5, 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{ber}, \texttt{aka}, \texttt{dab}, \texttt{era}, \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
5	2	$\langle 4, 5, 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{dab}, \texttt{era}, \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
6	3	$\langle 5, 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{era}, \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
7	5	$\langle 5, 6, 1, 7, . \rangle$	$\langle \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
8	6	$\langle 6, 1, 7, . \rangle$	$\langle \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
9	8	$\langle 7, . \rangle$	$\langle \texttt{ra\$}, \rangle$

Beachten Sie das zusätzliche Abschlusszeichen . für s^{12}. Es ist funktional identisch zum \$ für s. Zur leichteren Unterscheidung wurde aber ein anderes Zeichen gewählt. Das Abschlusszeichen benötigt keine Entsprechung in R^{12}, daher ist hier nur ein leeres Tripel eingetragen. Es wird für die Bestimmung von SA^{12} benötigt, kann aber im weiteren Verlauf ignoriert werden.

Das Suffixarray gibt eine Sortierung der Elemente von R^{12} bzw. s^{12} und damit auch von C^{12} an (für C^{12} ist dies spätestens dann ersichtlich, wenn man in der rechten Spalte alle Tripel nach \$\$\$ entfernt und die restlichen in einer Zeile konkateniert).

g) Aus dem Suffixarray lässt sich folgende Rang-Funktion ableiten:

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$s[i]$	a	b	e	r	a	k	a	d	a	b	e	r	a	\$	\$	...
$rank[i]$	$\perp$	4	7	$\perp$	3	8	$\perp$	5	2	$\perp$	6	9	$\perp$	1	0	...

Beachten Sie, dass $rank[i] = \perp$ anstatt 0 gesetzt wurde f.a. $(i \bmod 3) = 0$. Dies ist zulässig, da diese Einträge von *rank* nie verwendet werden.

h) Es ergeben sich die folgenden Tupel (für $s_i \in C^0$ gilt $(i \bmod 3) = 0$, $i < |s|$):

Index i	0	3	6	9	12
$(s[i], rank[i+1])$	(a, 4)	(r, 3)	(a, 5)	(b, 6)	(a, 1)

Sortiert ergibt sich: $\langle (\texttt{a}, 1), (\texttt{a}, 4), (\texttt{a}, 5), (\texttt{b}, 6), (\texttt{r}, 3) \rangle$. Diese Sortierung entspricht einer Sortierung von C^0. Alle unterschiedlichen Anfangsbuchstaben der Suffixe sind korrekt sortiert. Bei gleichem Anfangsbuchstaben ist über *rank* eine korrekte Sortierung ab dem zweiten Zeichen gegeben.

Musterlösung:

i) Für das Mischen werden C^0 und C^{12} in sortierter Reihenfolge benötigt. Diese Reihenfolge wurde in den vorherigen Teilaufgaben berechnet. Für den Vergleich beim Mischen wird zusätzlich für jedes Suffix eine bestimmte Tupeldarstellung benötigt.

Für C^0 ergeben sich die sortierte Reihenfolge und die benötigten Tupel wie folgt:

Index i	s_i	$(s[i], rank[i+1])$	$(s[i..i+1], rank[i+2])$
12	a$ $\in C^0$	(a, 1)	(a$, 0)
0	aberakadabera$ $\in C^0$	(a, 4)	(ab, 7)
6	adabera$ $\in C^0$	(a, 5)	(ad, 2)
9	bera$ $\in C^0$	(b, 6)	(be, 9)
3	rakadabera$ $\in C^0$	(r, 3)	(ra, 8)

Für C^{12} sind die sortierte Reihenfolge (z.B. aus $rank$ abzulesen) und die benötigten Tupel:

Index i	s_i	$(s[i], rank[i+1])$	$(s[i..i+1], rank[i+2])$
13	$ $\in C^1$	($, 0)	
8	abera$ $\in C^2$		(ab, 6)
4	akadabera$ $\in C^1$	(a, 8)	
1	berakadabera$ $\in C^1$	(b, 7)	
7	dabera$ $\in C^1$	(d, 2)	
10	era$ $\in C^1$	(e, 9)	
2	erakadabera$ $\in C^2$		(er, 3)
5	kadabera$ $\in C^2$		(ka, 5)
11	ra$ $\in C^2$		(ra, 1)

Zusammengemischt ergibt sich C zu:

Index i	s_i	$(s[i], rank[i+1])$	$(s[i..i+1], rank[i+2])$
13	$ $\in C^1$	($, 0)	
12	a$ $\in C^0$	(a, 1)	(a$, 0)
8	abera$ $\in C^2$		(ab, 6)
0	aberakadabera$ $\in C^0$	(a, 4)	(ab, 7)
6	adabera$ $\in C^0$	(a, 5)	(ad, 2)
4	akadabera$ $\in C^1$	(a, 8)	
9	bera$ $\in C^0$	(b, 6)	(be, 9)
1	berakadabera$ $\in C^1$	(b, 7)	
7	dabera$ $\in C^1$	(d, 2)	
10	era$ $\in C^1$	(e, 9)	
2	erakadabera$ $\in C^2$		(er, 3)
5	kadabera$ $\in C^2$		(ka, 5)
11	ra$ $\in C^2$		(ra, 1)
3	rakadabera$ $\in C^0$	(r, 3)	(ra, 8)

Die Suffixe in C sind offensichtlich lexikographisch sortiert (zweite Spalte). Damit erhält man das Suffixarray für s als SA $= \langle 13, 12, 8, 0, 6, 4, 9, 1, 7, 10, 2, 5, 11, 3 \rangle$.

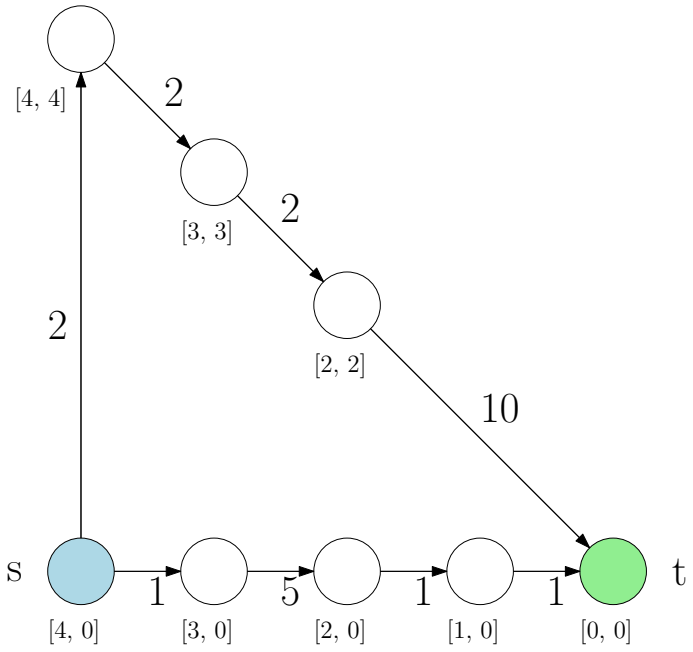
Aufgabe 4 (*Rechnen: A* Suche*)

Gegeben sei der unten abgebildete Graph. An den Kanten sind Kosten für die Nutzung der Verbindung eingetragen und die Knoten tragen Ortskoordinaten.

a) Ergänzen Sie den gegebenen Graphen um Knotenpotentiale für eine A* Suche von s nach t. Verwenden Sie Knoten t als Landmarke und die Manhatten-Distanz ($\hat{=}$ Einsnorm $||\cdot||_1$) als Abschätzung für die Entfernung zum Ziel.
Hinweis: $||\cdot||_1 : ||(x_1, y_1), (x_2, y_2)||_1 = y_2 - y_1 + x_2 - x_1$.

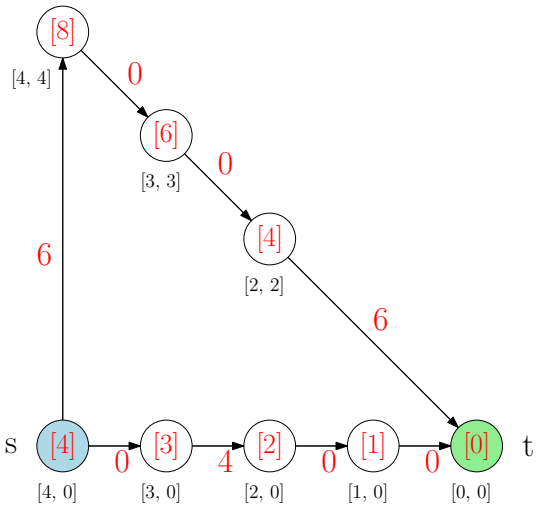
b) Tragen Sie die reduzierten Kantengewichte in den Graphen ein.

c) Wieviele `deleteMin` Operationen führt die A* Suche auf dem Graphen aus? Wieviele eine normale Suche mit Dijkstra's Algorithmus?



Musterlösung:

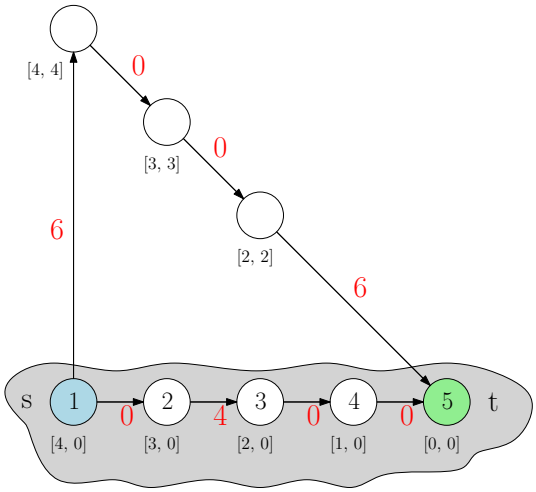
a) Knotenpotentiale pot(·) in Knoten eingetragen; Kantengewichte $c(\cdot)$ durch reduzierte Gewichte $\bar{c}(\cdot) : \bar{c}(u,v) = c(u,v) + \text{pot}(v) - \text{pot}(u)$ ersetzt:



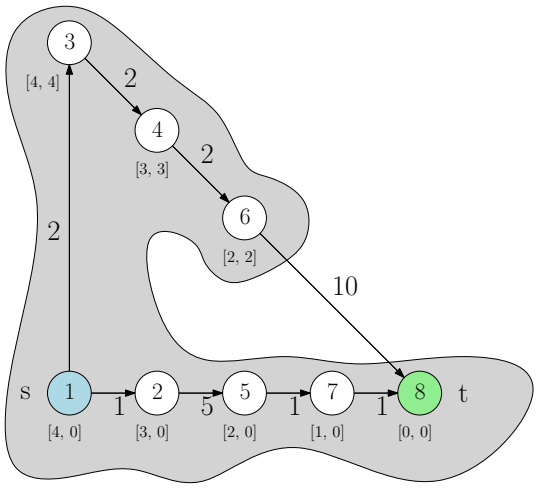
b) Siehe vorherige Teilaufgabe.

c) Die A* Suche benötigt 5 `deleteMin` Operationen, die normale Suche hingegen 8. Die entsprechenden Suchräume sind in den folgenden Abbildungen eingezeichnet. Die Knotennummerierung gibt die Reihenfolge der `deleteMin` Operationen an.

A*:



Dijkstra:



Aufgabe 5 (*Kleinaufgaben: A* Suche*)

a) Sei pot$(\cdot)$ eine gültige Potentialfunktion für die A* Suche nach Knoten t in Graph $G(V,\ E)$. Überprüfen Sie, ob

$$\text{pot}^c = \text{pot} + c,\ c = const.$$

ebenfalls eine gültige Potentialfunktion darstellt.

b) Kann es vorkommen, dass eine A* Suche mehr Knoten absucht als eine Suche mit Dijkstra's Algorithmus für die gleiche Anfrage? Begründen Sie warum nicht oder geben Sie ein Beispiel an.

Musterlösung:

a) Es ist zu überprüfen, ob

$$c(u,v) + \text{pot}^c(v) - \text{pot}^c(u) \geq 0 \quad (1)$$
$$\text{pot}^c(u) \leq \mu(u,t) \quad (2)$$

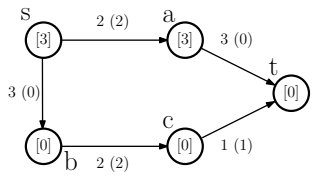
gilt.

Bedingung (1) ist immer erfüllt. Nach Einsetzen ergibt sich $c(u,v) + \text{pot}(v) - \text{pot}(u) \geq 0$. Da nach Voraussetzung pot$(\cdot)$ eine gültige Potentialfunktion ist, ist dies erfüllt.

Bedingung (2) ist hingegen nur erfüllt, wenn $c \leq \mu(u,t) - \text{pot}(u)$ f.a. $u \in V$.

Damit ist pot$^c(\cdot)$ nur für geeignete Wahl von c eine gültige Potentialfunktion.
(Bemerkung: Falls $\text{pot}(t) = 0$ f.a. Potentiale gefordert ist (anstatt nur $\text{pot}(t) \leq 0$), gilt $c = 0$!)

b) Im bidirektionalen Fall kann dies durchaus einfach vorkommen. Im unidirektionalen Fall hängt es von der Reihenfolge der betrachteten Knoten gleicher Distanz ab. Nehmen wir eine FIFO Ordnung der Knoten gleichen Gewichtes an (z.B. in einer Bucket Queue), so ist folgender Graph ein Beispiel. Die Dijkstra Suche scannt die Knoten in der Reihenfolge: s, a, b, t, während A* die Knoten in der Reihenfolge s, b, a, c, t scannt.



Legende: Werte in eckigen Klammern geben Knotenpotentiale an, Werte in runden Klammern reduzierte Kantengewichte.

Aufgabe 6 (*Analyse: Kürzeste Wege (Wiederholung)*)

a) Betrachten Sie eine Suche mit bidirektionalem Dijkstra. Geben Sie eine Familie von Graphen an, bei der ein ausgezeichneter Knoten u eine Anzahl an `decreaseKey` Operationen erfährt, die linear in der Länge des kürzesten Weges für jede mögliche Anfrage ist.

b) Bei manchen Algorithmen kann es sinnvoll sein, unterschiedliche Potentialfunktionen zu kombinieren. Zeigen Sie in diesem Zusammenhang: Sind π_1 und π_2 gültige Potentialfunktionen, so ist auch $\pi = \frac{\pi_1 + \pi_2}{2}$ eine gültige Potentialfunktion.

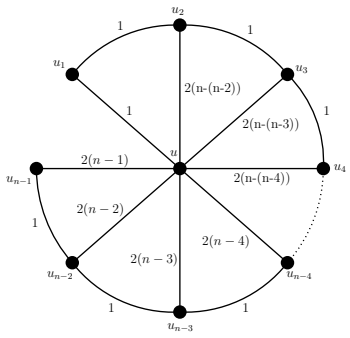
c) Bei der Durchführung von *Dijkstras Algorithmus* können kürzeste Wege gespeichert werden, indem Vorgängerknoten gespeichert werden. Diese werden immer dann geändert, wenn eine bessere vorläufige Distanz gefunden wird. Dieses kann auch auf einem Graphen durchgeführt werden, der negative Kantengewichte enthält. Das Stopkriterium von Dijkstras Algorithmus muss dafür durch ein schwächeres Kriterium ersetzt werden: Es wird gestoppt, sobald keine Verbesserung mehr gefunden werden kann. Zeigen Sie: Wenn auf diese Art ein Kreis entsteht, so ist dieser negativ.

d) (*) Dijkstras Algorithmus ist ein Spezialfall des allgemeinen *Labeling Algorithmus*. Der allgemeine Labeling Algorithmus wählt eine beliebige Kante aus, die das Label des Zielknotens verbessert. Geben Sie einen Graphen sowie eine Reihenfolge der Kanten an, sodass bei positiven Kantengewichten eine exponentielle Zahl von Schritten ausgeführt wird.

Hinweis: Achtung, schwierige Knobelaufgabe!

Musterlösung:

a) Der abgebildete Graph mit n Knoten ist ein mögliches Beispiel.



Jeder kürzeste Weg zwischen zwei Knoten geht entlang des äußeren Kreises. Wir bezeichnen den mittleren Knoten mit u und die Knoten auf dem Kreis mit u_1 bis u_{n-1} entsprechend der Abbildung. Betrachten wir den Pfad $u_i, \dots u_j$ mit $i > j$ und kürzester Weglänge $\mu(i,j) = i - j$. O.b.d.A sei $s = u_i$ und $t = u_j$. Die Vorwärtssuche führt zwischen $3(i-j)/4$ und $(i-j)/4$ Schritte entlang des Pfades aus (die Suchrichtungen wechseln sich ab und besuchen ggf. auch Knoten außerhalb des Pfades). Bis auf den ersten Schritt führt jeder dieser Schritte zu einer `decreaseKey` Operation auf Knoten u. Schritte außerhalb des Pfades führen zu keiner `decreaseKey` Operation auf Knoten u.

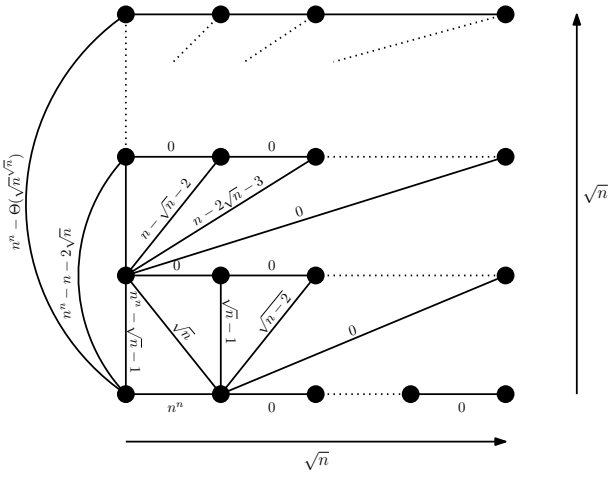
b) Es ist zu zeigen:

- π ist eine untere Schranke der Distanzfunktion: Es gilt: $\pi_1(u) \leq \mu(u,t)$, sowie $\pi_2(u) \leq \mu(u,t)$ für alle Knoten $u \in V$. Daraus folgt: $\pi_1(u) + \pi_2(u) \leq 2\mu(u,t)$ und damit $\pi \leq \mu(u,t)$.
- Die Kantengewichte sind nicht negativ: Für jede Kante $(u,v) \in E$ gilt: $c((u,v)) + \pi_1(v) \geq \pi_1(u)$, sowie $c((u,v)) + \pi_2(v) \geq \pi_2(u)$. Daraus folgt direkt: $2c((u,v)) + \pi_1(v) + \pi_2(v) \geq \pi_1(u) + \pi_2(u)$ und somit $c((u,v)) + \pi(v) \geq \pi(u)$.

Musterlösung:

c) Nehmen wir an, dass es einen solchen Kreis geben würde und dieser wäre nicht negativ. Betrachten wir nun den Moment, in dem der Kreis zum ersten Mal geschlossen wird. Dies geschehe an Knoten u. Der Kreis sei dabei bezeichnet als $k = \{u = n_1, \ldots, n_k = u\}$. Wenn der Kreis ein positives Gewicht hätte, so wäre $d(s,u) + k \geq d(s,u)$ und somit würden wir den Vorgänger von u nicht auf n_{k-1} setzen.

d) Der abgebildete Graph ist ein mögliches Beispiel.



Im allgemeinen Labeling Algorithmus kann die Ausführungsreihenfolge beliebig schlecht gewählt werden. Der angegebene Graph erlaubt es, immer die komplette untere Reihe von Knoten abzulaufen, und dann dem zweiten Knoten dieser Reihe eine um 1 kleinere Distanz zuzuweisen. Dabei kann jede der $\sqrt{n}$ Zeilen dazu genutzt werden, den ersten Knoten der darunter liegenden Reihe $\sqrt{n}$ mal eine kürzere Distanz zuzuweisen. Rekursiv lässt sich somit eine Bearbeitungsreihenfolge festlegen. Wir starten mit der Bearbeitung der untersten Zeile (in Serie). Wenn wir n Zeilen bearbeiten können, so können wir $n+1$ Zeilen bearbeiten, indem wir $\sqrt{n}$ mal den zweiten Knoten der obersten Reihe der n Reihen eine um 1 kürzere Distanz als bisher bekannt zuweisen und danach die Bearbeitung der n Reihen wiederholen. Somit ergibt sich eine Gesamtbearbeitungszeit von $\sqrt{n}^{\sqrt{n}}$ Schritten.