Algorithmen zum Ermitteln des Lowest Common Ancestor (LCA)

Inhalt

Problembeschreibung

Gegeben sei ein Baum $T = (V, E)$ mit einem Wurzelknoten r, n Knoten ($|V| = n$) und der Höhe h. Der *Lowest Common Ancestor (LCA)* zweier Knoten u und v ($u, v \in V$) ist derjenige Knoten $z \in V$, der ein Elternknoten von sowohl u als auch v ist und am weitesten von der Wurzel r entfernt liegt. (Dabei kann z auch ein unechter (uneigentlicher) Elternknoten sein, also $z = u$ und/oder $z = v$).

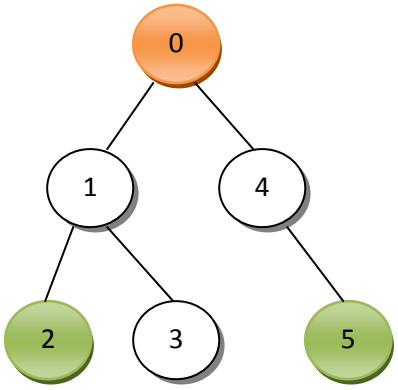

Abbildung 1 Beispiel LCA

In der Grafik von ***Abbildung 1*** Beispiel LCA wurde der $LCA(2,5) = 0$ ermittelt. Weitere Beispiele wären (für denselben Baum) $LCA(2,3) = 1$, $LCA(1,3) = 1$ usw.

Anwendungsgebiete

Das LCA-Problem kann verwendet werden, um Berechnungen auf Speziesstammbäumen und Gen-Bäumen (Bio-Informatik) anzustellen.

Allgemeiner betrachtet, lässt sich das LCA-Problem auch anwenden, wenn man die Distanz zweier Knoten u und v in einem Baum ermitteln möchte. Hier kann man dann die Distanz vom $LCA(u,v)$ zur Wurzel r ignorieren.

Historisches

(Auszug)

- 1973: Das LCA-Problem wurde 1973 von Alfred Aho, John Hopcroft und Jeffrey Ullman definiert
- 1984: Dov Harel und Robert Tarjan entwickelten 1984 die erste effiziente Datenstruktur zur Lösung des LCA-Problems. Dabei wird der Eingabebaum in $O(n)$ vorverarbeitet, so dass die Abfragen in konstanter Zeit, $(O(1)$, beantwortet werden können. Allerdings gilt die Datenstruktur all sehr komplex und schwierig zu implementieren.
- Tarjan fand später einen simpleren, allerdings auch weniger effizienten Algorithmus, der auf der Union-Find-Struktur basiert und den LCA aus einer vorher berechneten Menge von Knotenpaaren ermittelt („Tarjan's Offline Least Common Ancestor Algorithm" (TOLCA))
- 1983: Hal Gabow und Tarjan verfeinerten 1983 TOLCA, so dass der LCA in linearer Zeit berechnet werden kann.
- 1988: Baruch Schieber und Uzi Vishkin vereinfachten 1988 die Datenstruktur von Harel und Tarjan (1984), so dass diese implementierbar wurde und dennoch einen Vorverarbeitungsaufwand von $O(n)$ Zeit und einen Abfrageaufwand von $O(1)$ aufweist.
- 1993: Omer Berkman und Uzi Vishkin entdeckten 1993 einen neuen Weg, das LCA-Problem zu lösen. Sie setzten dabei und die Reduktion auf das Range-Minimum-Query-Problem (RMQ), die basierend auf einer Euler-Tour auf dem Eingabebaum vollzogen werden kann. Das RMQ-Problem wird dann aus einer Kombination aus a) der Erstellung großer zweierpotenzen-langer Intervalle und b) Lookup-Tables für Abfragen auf kleinen Intervallen gelöst. Der Zeitaufwand beträgt auch hier lineare Vorverarbeitungszeit $O(n)$ und konstante Abfragezeit $O(1)$.
- 2000: Michael Bender und Martin Farach-Colton vereinfachten 2000 den Lösungsansatz von Berkman und Vishkin.

Algorithmen zum Ermitteln des Lowest Common Ancestor (LCA)
Ingo Mohr
16. Juli 2009

Erster Lösungsansatz

Wir gehen davon aus, dass

1. weder u noch v die Wurzel r des Baums sind (in dem Fall wären wir sofort fertig mit $LCA(u,v) = r$) ,
2. $u \neq v$ gilt (ansonsten hätten wir sofort das Ergebnis $LCA(u,v) = u = v$) und
3. T kein Pfad ist. In dem Fall wäre der $LCA(u,v)$ der Knoten aus $\{u,v\}$, welcher der Wurzel am nächsten steht.

Für den naiven Ansatz definieren wir uns die Pfade $P_u = [u, p(u), p(p(u)), \dots, r]$ und $P_v = [v, p(v), p(p(v)), \dots, r]$, wobei $p(i)$ der direkte Elternknoten des Knotens i ist.

Für jeden Pfad haben wir die Invarianten, dass

1. das erste Element (an Index 0) im Pfad einer der beiden LCA-Eingabeknoten u bzw. v ist,
2. das letzte Element im Pfad das der Wurzel nächste Element auf dem entsprechenden Pfad ist (bzw. die Wurzel selbst) und
3. das Element an Index i der direkte Elternknoten zu dem an Index $i-1$ ist.

Wir konstruieren nun beide Pfade. Wir erstellen zunächst $P_u = [u]$ und $P_v = [v]$ als 1-elementige Listen, die lediglich den jeweiligen Parameterknoten u bzw. v enthalten.

Von nun an gehen wir Pfad-abwechselnd vor.

Iteration: Wir betrachten den zu konstruierenden Pfad P_x. Sei P[k] das letzte Element (d.h. $|P_x| = k+1$). Wir ermitteln $p(P_x[k])$. Liegt $p(P_x[k])$ nun schon im jeweils anderen Pfad, so ist $LCA(u,v) = p(P_x[k])$ und wir sind fertig, da sich beide Pfade offensichtlich hier treffen. Liegt $p(P_x[k])$ nicht im anderen Pfad, so fügen wir diesen Knoten an P_x an, so dass $P_x[k+1] = p[P_x[k]]$ und $|P_x| = k+2$) gelten. Im Anschluss wechseln wir zum jeweils anderen Pfad, setzen ihn als P_x und beginnen die Iteration von vorn. Kann nur noch ein Pfad weiterkonstruiert werden, so bleiben wir bei diesem und beginnen die Iteration von vorn.

Der Worstcase tritt ein, wenn $LCA(u,v) = r$ gilt und mindestens einer der Knoten u und v ein Blatt in T ist. Das führt zu einer Laufzeit von O(h), wobei h die Höhe von T ist. In Binärbäumen entspricht das einer Laufzeit von $O(log\ n)$. Für allgemeine Bäume geht die Laufzeit gegen $O(n)$.

Weitere Lösungsansätze

Es gibt allerdings auch Lösungswege, in denen sich eine LCA-Abfrage performanter durchführen lässt. Die hier vorgestellten Algorithmen verarbeiten dazu den Eingabebaum T vor.

Notation: Besteht ein Algorithmus aus einer Vorverarbeitungzeit $f(n)$ und einer Abfragezeit $g(n)$, so schreiben wir seine Komplexität als $< f(n), g(n) >$.

Berechnen des LCA auf Basis binärer Pfad-Nummern

Der folgende Ansatz zielt auf eine Aufwandskomplexität von $< O(n), O(1) >$ Zeit.

Obwohl sich dieser Ansatz verallgemeinern lässt, gehen wir für die Beschreibung der Einfachheit halber davon aus, dass T ein Binärbaum mit n Knoten ist.

Der Schlüssel zum Erfolg basiert auf der Idee, dass jeder Pfad in T von der Wurzel r zu einem Knoten v eindeutig binärcodiert werden kann.

Für jeden Knoten v in T codieren wir eine Pfad-Nummer der Länge $log\ n$ wie folgt:

- Das i-te Bit (von links) der Pfadnummer von v bezieht sich auf die i-te Kante auf dem Pfad von der Wurzel r nach v.
- Eine 0 für das i-te Bit (von links) zeigt an, dass die i-te Kante auf dem Pfad zum linken Kindsknoten führt. Eine 1 zeigt an, dass die Kante zum rechten Kindsknoten führt.
- Sei k die Anzahl der Kanten im Pfad von r nach v. Wir markieren das (k+1)-te Bit in der Pfadnummer (von links) als „Height Bit“ mit einer 1. Alle restlichen $log\ n - k - 1$ Bits markieren wir mit 0.

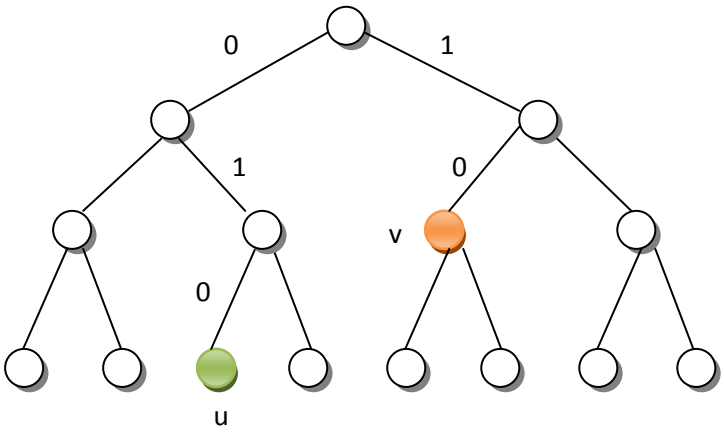

Abbildung 2 Pfadnummern-Kodierung

Abbildung 2 zeigt ein Beispiel für die Pfadnummern-Kodierung. Der Pfad zu Knoten u wäre hierbei 0101 (das Height-Bit ist hervorgehoben). Der Pfad zu Knoten v wäre 1010.

Die Knoten des kompletten Baums lassen sich nun in $O(n)$ mit Pfadnummern versehen, indem wir bei 0001 starten und eine In-Order-Traversierung durchführen (siehe ***Abbildung*** 3).

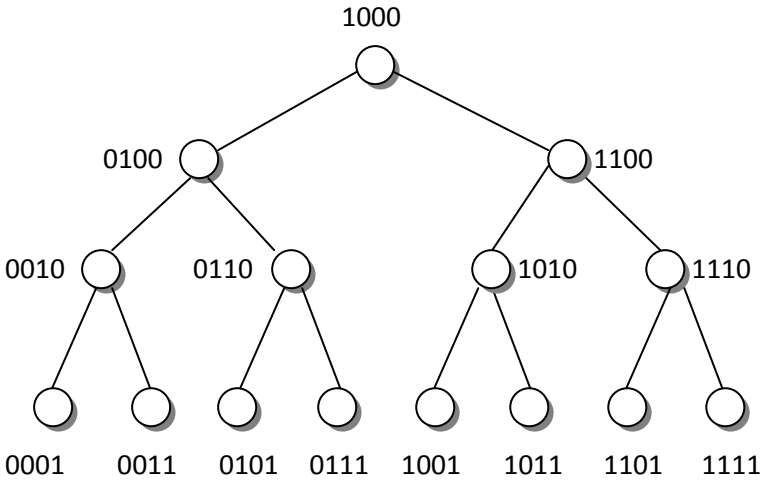

Abbildung 3 Vollständig vorverarbeiteter Baum

Damit ist die Vorverarbeitung des Baums abgeschlossen – in $O(n)$ Zeit.

Nehmen wir nun an, wir haben die Knoten u und v mit den entsprechenden Pfaden $path(u)$ und $path(v)$.

Seien alle Prefix-Bits (in einer Pfadnummer), welche sich auf die Kanten des Pfades von der Wurzel her beziehen, „Path Bits“.

Nun berechnen wir zunächst $path(u)$ **XOR** $path(v)$, ermitteln auf dem Ergebnis das erste auf 1 gesetzte Bit (von links) und fahren wie folgt fort:

- Gibt es kein 1-Bit, so muss gelten $path(u) = path(v)$ und also auch $u = v$.
- Wenn sowohl das k-te Bit in $path(u)$ als auch das k-te Bit in $path(v)$ „Path Bits“ sind, stimmen die Pfade für die ersten $k - 1$ Kanten von der Wurzel aus – hin zum $LCA(u, v)$ – überein.
- Wenn nur das k-te Bit von $path(u)$ oder das k-te Bit von $path(u)$ ein „Path Bit“ ist (oder wenn keines der beiden k-ten Bits ein „Path Bit“ ist), so ist einer der beiden Knoten ein Elternknoten des anderen. In dem Fall können wir den LCA leicht über das „Height Bit“ bestimmen.

Wir erhalten den LCA-Pfad, indem wir den $k - 1$ Prefix übernehmen, eine 1 anfügen und die übrigen $log\ n - k$ Stellen mit 0en auffüllen.

Für die Knoten u und v aus ***Abbildung 2*** hätten wir also:

0101 XOR 1010 = 1111 => 1000

Schon das erste Bit (von links) ist mit einer 1 markiert. Folglich teilen die beiden Pfade überhaupt keine Kante, womit $LCA(u, v) = r$ wäre. Und nach dem Erstellen des LCA-Pfads erhalten wir (wie erwartet) mit 1000 den Pfad zur Wurzel r.

Wenn wir voraussetzen, dass sich alle diese Bits-Operationen (im RAM-Modell) in konstanter Zeit durchführen lassen, so liefert uns das eine Abfrage-Laufzeit in $O(1)$.[1]

Folglich haben wir für diesen Ansatz eine Zeitkomplexität von $< O(n), O(1) >$.

Dieser Ansatz lässt sich für beliebige Bäume erweitern, indem das LCA-Problem auf das Restricted Range Minima (RRM) – Problem reduziert und mit diesem gelöst wird.[2]

Reduktion auf Range Minimum Query (RMQ)

Wir lösen nun das LCA-Problem, indem wir es auf das Range Minimum Query Problem (RMQ) reduzieren. Wir legen außerdem fest, dass wir das RMQ-Problem in zwei Problemteile zerlegen: Vorverarbeitung (Preprocessing) und Abfrage (Query).

Problembeschreibung RMQ

Auf einem Array A der Länge n aus Zahlen wird für die Indizes i und j ($0 <= i, j < n$) der $RMQ(i, j)$ gesucht: der Index, der in dem Sub-Array $A[i \ldots j]$ auf den kleinsten Wert verweist.

Indizes:	0	1	2	3	4	5	6	7	8	9	10
Werte:	0	1	2	1	3	1	0	4	5	4	0

Abbildung 4 RMQ Beispiel 1

Abbildung 4 zeigt ein Beispiel-Array der Länge 11, auf welchem die Anfrage $RMQ(3, 7)$ ausgeführt wird. Das Ergebnis ist 6, da es mit 0 auf den kleinsten Wert im Sub-Array A[3…7] verweist.

Reduktion LCA -> RMQ

Lemma 1: Gibt es eine Lösung für das RMQ-Problem in $< f(n), g(n) >$ Zeit, so gibt es eine Lösung für das LCA-Problem in $< f(2n-1) + O(n), g(2n-1) + O(1) >$ Zeit.

Beweis (Reduktion): Sei T der Eingabebaum. Die Reduktion basiert auf folgender Beobachtung.

Beobachtung 1: Der $LCA(u, v)$ ist jener Knoten auf dem Pfad der Depth-First-Search (DSF)-Traversierung von u nach v, der den niedrigsten Level im Baum aufweist (am nächsten zur Wurzel r liegt).

[1] Für das Erfassen des i-ten Bits von links in $O(1)$ ließe sich eine konstante Zahl von Lookup-Tabellen der Größe $O(n)$ nutzen.
[2] Das Restricted Range Minima Problem sucht die kleinste Zahl in einem gegebenen Interval einer fest vorgegebenen List e von Zahlen, wobei die Differenz zwischen zwei aufeinanderfolgenden Zahlen exakt 1 beträgt.

Folglich reduzieren wir:

1. Wir erstellen ein Elementen-Array $E[1, \dots, 2n-1]$ mit den Labels der Knoten aus T in der Reihenfolge der Euler-Tour auf T. Die Euler-Tour entspricht dabei der DFS-Traversierung auf T, beginnend und endend mit der Wurzel r. Dabei verwenden wir jede Kante genau zweimal (Hin- und Rück-Richtung). So verweist jedes $E[i]$ auf den i-ten Knoten in der Euler-Tour auf T.
2. Sei der Level eines Knotens die Distanz von der Wurzel r. Wir erstellen ein Level-Array $L[1, \dots, 2n-1]$, so dass jedes $L[i]$ auf den Level des Knotens $E[i]$ verweist.
3. Sei die Repräsentation eines Knotens in der Euler-Tour der Index seines ersten Vorkommens in der Euler-Tour: $Repr(i) = argmin_j\{E[j] = i\}$. Wir erstellen das Repräsentations-Array $R[1, \dots, n]$, so dass $R[i]$ der Index des Repräsentanten von i ist.

Jeder dieser Schritte nimmt $O(n)$ Zeit ein, was zusammengenommen ebenfalls eine Laufzeit von $O(n)$ ergibt.

Um nun $LCA_T\ (u, v)$ zu berechnen, verweisen wir auf Folgendes:

1. Die Knoten in der Euler-Tour zwischen den ersten Vorkommen von u und v sind $E[R[u], \dots, R[v]]$.
2. Der level-niedrigste Knoten in dieser Sub-Tour befindet sich auf dem Index $RMQ_L(R[u], R[v])$, da $L[i]$ den Level des Knotens in $E[i]$ enthält und RMQ uns den Index des Knotens mit dem niedrigsten Level liefert.
3. Der Knoten an dieser Position ist $E[RMQ(R[u], R[v])]$ und somit das Ergebnis der $LCA_T(u, v)$ – Anfrage.

Wir vervollständigen die Reduktion durch die Vorverarbeitung des Level-Arrays L. Es hat die Länge $2n-1$, und wir benötigen $O(n)$ Zeit, es zu erstellen. Folglich beläuft sich die komplette Vorverarbeitung auf $f(2n-1\ +\ O(n))$. Um die Abfrage zu berechnen, führen wir eine RMQ-Abfrage auf L (in $O(2n-1)$ – bezogen auf die Länge von L) und drei Array-Referenzierungen (jeweils in $O(1)$) aus. Daraus folgt ein Abfrage-Aufwand von $g(2n-1\ +\ O(1))$.

■

Beispiel:

Wir verwenden den Baum aus ***Abbildung 1*** als Eingabe.

Wir erstellen:

Indizes:	0	1	2	3	4	5	6	7	8	9	10
E:	0	1	2	1	3	1	0	4	5	4	0
L:	0	1	2	1	2	1	0	1	2	1	0
R:	0	1	2	4	7	8					

Abbildung 5 Beispielreduktion

Abbildung 5 zeigt das Ergebnis der Vorverarbeitung für die Eingabe T. Für die Erstellung des Elementen-Arrays E beginnen wir die Euler-Tour bei Knoten 0. Es folgen Knoten 1, 2, 1, 3,… und schließlich enden wir wieder mit der Wurzel $r = 0$. Wie erwartet, beträgt die Länge von E $2n - 1 = 11$ (T besitzt 6 Knoten). Im Level-Array L tragen wir die Level der Knoten (in ihrer Euler-Tour-Reihenfolge) ab. Und das Repräsentanten-Array R erhält die E-Indizes der ersten Vorkommen der Knoten, die mit dem jeweiligen R-Index beschriftet sind: Der Knoten 4 kommt in zuerst an $E[7]$, also ist $R[4] = 7$ usw.

Wir berechnen nun LCA(2,4). Dazu lassen wir uns aus R R[2] = 2 und R[4] = 7 ausgeben. Damit begeben wir uns auf das Array L und führen $RMQ_L(2,7)$ aus. Das Ergebnis ist 6. Damit gehen wir zum Array E, springen nach Index 6 und geben $E[6] = 0$ als Ergebnis von LCA(2,4) aus. Fertig.

Naiver Lösungsansatz für RMQ-Abfrage in O(1)

Wenn wir die RMQ-Abfrage für beliebige Paare (i, j) in konstanter Zeit ausführen wollen, ist der einfachste Weg, eine Tabelle zu erstellen, die für jedes (i, j) die Lösung vorberechnet parat hält. Wir betrachten die Länge des L-Arrays mit $|L| = n$. Jede RMQ-Berechung dauert maximal n Schritte. Also erhalten wir eine Vorverarbeitungs-Laufzeit von $O(n^3)$.

Unter Zuhilfenahme von dynamischer Programmierung lässt sich die Vorverarbeitungszeit auf $O(n^2)$ reduzieren.

Da wir die $O(1)$-Abfragezeit aber gerade wegen der zu erwartenden Größe des Baums T haben wollen, bringt uns eine Vorverarbeitungs-Laufzeit in $O(n^2)$ eher wenig.

Für die weiteren Algorithmen beschränken wir unsere Sichtweise auf die Lösung des RMQ-Problems. Dieses soll im Folgenden auf einem Array A als Eingabe gelöst werden.

Schnellerer Algorithmus

Dieser Ansatz basiert, wie der naive Ansatz, auf der Idee der Lookup-Tabelle. Allerdings begrenzen wir hier den Speicher- und Zeitaufwand der Tabelle (und damit der gesamten Vorverarbeitung) auf $O(n\ log\ n)$. Der Abfrage-Aufwand bleibt bei $O(1)$.

Die Idee ist es, alle Abfragen mit der Länge einer Zweierpotenz vorzuberechnen. Das heißt, für jedes i von 1 bis n und jedes j von 1 bis $log\ n$ finden wir das Minimum-Element in dem Block, der bei i beginnt und die Länge 2^j hat. Wir berechnen also $M[i,j] = argmin_{k=i\ldots i+2^j-1}\{A[k]\}$. Demnach hat die Tabelle M eine Größe von $O\ (n\ log\ n)$, und wir benötigen $O(n\ log\ n)$, sie zu füllen - mit dynamischer Programmierung:

- $M[i,j]$ = $M[i,j-1]$, wenn $A[M[i,j-1]] \leq M[i+2^{j-1}-1, j-1]$ und
- $M[i,j]$ = $M[i+2^{j-1}-1, j-1]$ sonst

Für eine allgemeine RMQ-Abfrage selektieren wir zwei sich überlappende Blocks, die den gefragten Intervall komplett ausfüllen. D.h. sei 2^k die Länge des größten Blocks, der in den Intervall von i nach j passt, also $k = \lfloor \log(j-i) \rfloor$. Der $RMQ(i,j)$ kann nun berechnet werden durch das Ermitteln des Minimums der folgenden beiden Blöcke:

- $i \dots i + 2^k$ $(M(i,k))$
- $j - 2^k + 1 \dots j$ $(M(j - 2^k + 1, k))$

Beide Ergebnisse liegen schon vor. Wir bleiben also bei Abfrageaufwand von $O(1)$.

Dies ergibt den *Sparse-Table* (ST) Algorithmus mit einer Komplexität von $< O(n \log n), O(1) >$.

RMQ-Abfrage in <O(n), O(1)> für $^{\pm}RMQ$

Für den hier vorgestellten Algorithmus betrachten wir das Array L (aus der Reduktion des LCA-Problems auf das RMQ-Problem) und stellen fest, dass sich für jeden Index i die Nachbarwerte, also $L[i-1]$ und $L[i+1]$ (so vorhanden), um genau 1 bzw. -1 von $L[i]$ unterscheiden. Wir formulieren diese Eigenschaft als „$^{\pm}RMQ$“. Wir definieren diese Eigenschaft für den folgenden Algorithmus als restriktive Voraussetzung.

Da wir uns ansonsten losgelöst vom LCA-Problem bewegen, definieren wir das Array L für diesen Algorithmus wieder als A.

Wir partitionieren nun A in Blöcke der Länge $\frac{\log n}{2}$. Dazu definieren wir ein Array $A'[1, \dots \frac{2n}{\log n}]$, wobei $A'[i]$ das kleinste Element im i-ten Block von A enthält. Außerdem definieren wir ein gleichgroßes Array B, wobei $B[i]$ den Index des kleinsten Elements im i-ten Block von A enthält. Damit können wir uns merken, woher das Minimum in A' kam (denn: RMQ liefert den Index des kleinsten Werts, nicht den Wert selbst).

Der ST-Algorithmus läuft auf Array A' in $< O(n), O(1) >$ Zeit. Wenn wir A' vorbereitet haben, betrachten wir, wie wir RMQ-Abfragen auf A vollziehen können. Die Indizes i und j können sich in demselben Block befinden, also müssen wir jeden Block vorberechnen. Wenn $i < j$ in verschiedenen Blöcken liegen, können wir für $RMQ(i,j)$ wie folgt vorgehen:

Wir berechnen zunächst folgende Werte:

1. Das Minimum von i bis zum Ende seines Blocks
2. Das Minimum aller Blöcke, die sich zwischen dem Block mit i und dem Block mit j befinden
3. Das Minimum des Blocks, der j enthält, von seinem ersten Element bis zu j.

Das Ergebnis der RMQ-Abfrage ist dann das Minimum aus 1., 2. und 3. .

Der ST-Algorithmus liefert uns das Minimum zu 2. In konstanter Zeit.

Für die anderen beiden Minima betrachten wir nun RMQ-Abfragen innerhalb eines Blocks. Hier kommt uns die $^{\pm}RMQ$-Restriktion zu gute. Für alle Blöcke X[1,…,k] und Y[1+c,…,k+c] liefert die RMQ-Berechnung dasselbe Ergebnis. Wir müssen also nicht alle Blöcke berechnen.

Wir können einen Block normalisieren, indem wir seinen initialen Offset von jedem seiner Elemente abziehen.

Lemma 2: Es gibt $\sqrt{n}$ Arten normalisierter Blöcke.

Beweis: Benachbarte Elemente in normalisierten Blöcke unterscheiden sich um +1 bzw. -1. Also können normalisierte Blöcke durch einen ± 1-Vektor der Länge $\frac{1}{2}\log n - 1$ angegeben werden. Es gibt $2^{(\frac{1}{2}\log n)-1} = O(\sqrt{n})$ solcher Vektoren.

■

Wir erstellen $O(\sqrt{n})$ Tabellen, eine für jeden möglichen normalisierten Block. In jede Tabelle tragen wir alle $(\frac{1}{2}\log n)^2 = O(log^2 n)$ Antworten auf alle In-Block – Abfragen ein. Das ergibt einen totalen Vorverarbeitungsaufwand von $O(\sqrt{n}\ log^2 n)$. Zuletzt berechnen wir für jeden Block in A, welche normalisierte Blocktabelle für RMQ-Abfragen benutzt werden soll. Danach benötigen wir für die Abfragen nur noch Lookups.

Alles in allem beläuft sich die Komplexität für die Vorverarbeitung der normalisierten Blocktabellen und A‘ Tabellen auf $O(n)$ und für die Abfrage auf $O(1)$.

Literaturverzeichnis

Bender, M. A., & Farach-Colton, M. (16. Mai 2000). *The LCA Problem Revisited.* Abgerufen am 01. 07 2009 von www.ics.uci.edu/~eppstein/261/BenFar-LCA-00.pdf

Hermelin, D. (kein Datum). *Constant-Time LCA Retrieval.* Abgerufen am 15. 07 2009 von cs.haifa.ac.il/LANDAU/courses/String/String_files/LCA.ppt

Narasimhan, G., & Smid, M. (2007). *Geometric Spanner Networks.* Cambridge.

Wikipedia (LCA). (kein Datum). Abgerufen am 02. 07 2009 von http://en.wikipedia.org/wiki/Lowest_common_ancestor