Fortgeschrittene Datenstrukturen — Vorlesung 8

Schriftführer: Maximilian Schuler

15.12.2011

1 Tree Decomposition: Level Ancestor Queries

1.1 Literaturempfehlungen

- M.A. Bender, M. Farach-Colton: *The Level Ancestor Problem Simplified*, Theor. Comput. Sci. 321(1): 5–12 (2004). `http://cg.scs.carleton.ca/~morin/teaching/5408/refs/bf-c04.pdf`

1.2 Einführung

Zunächst sollte, zugunsten einer übersichtlicheren Notation, der Hyperfloor-Operator eingeführt werden: $\lfloor\lfloor\, x \,\rfloor\rfloor := 2^{\lfloor \lg x \rfloor}$ bezeichne die größte Zweierpotenz nicht größer als x.

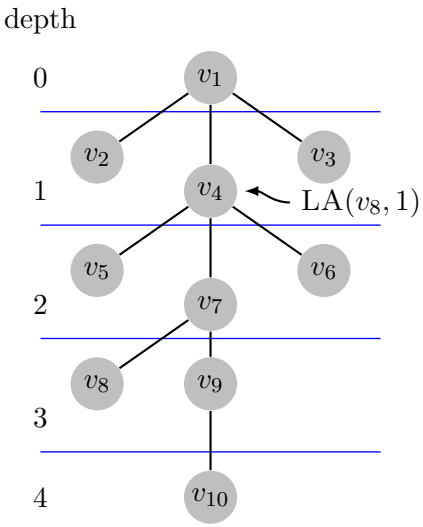
Für das Problem sei ein statischer Baum T mit n Knoten gegeben, der so *vorverarbeitet* werden soll, sodass Anfragen der folgenden Art möglichst effizient beantwortet werden können:

$$\textsc{LevelAncestor}_T(u, d) : \text{return } u\text{'s ancestor at depth } d \leq \text{depth}(u)$$

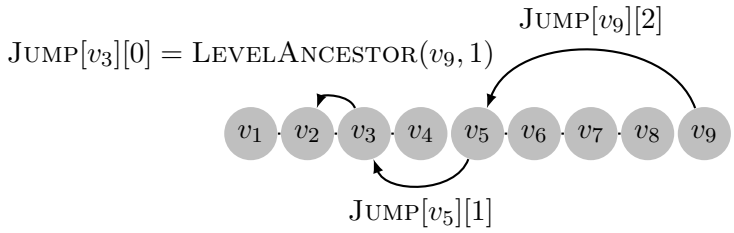
Daraus folgt, dass $\textsc{LevelAncestor}_T(u, 0)$ die Wurzel und $\textsc{LevelAncestor}_T(u, \text{depth}(u))$ u selbst zurückliefert.

Direkt fallen zwei naive Ansätze ein, die in Laufzeit und Speicherverbrauch komplementär sind:

- speichere nur den Baum; suche nach Vorgänger mittels direkter Baumtraversierung
  (Speicherverbrauch: *nicht zutreffend*, Laufzeit: $\mathcal{O}(n)$).
- speichere $\textsc{LevelAncestor}_T(u, d)$ für alle u und alle $0 \leq d < \text{depth}(u)$
  (Speicherverbrauch: $\mathcal{O}(n^2)$, Laufzeit: $\mathcal{O}(1)$).



Nun ist die Herausforderung, die Vorteile beider Ansätze in einem einzigen Vorverarbeitungsverfahren zu vereinen. Es wird sich zeigen, dass sich in der Tat ein Verfahren konstruieren lässt, das sowohl linear im Platzverbrauch ist und mit welchem sich in konstanter Zeit das LevelAncestor-Problem beantworten lässt.



1.3 $\mathcal{O}(1)$ Level Ancestor mit $\mathcal{O}(n \lg n)$ Platzverbrauch

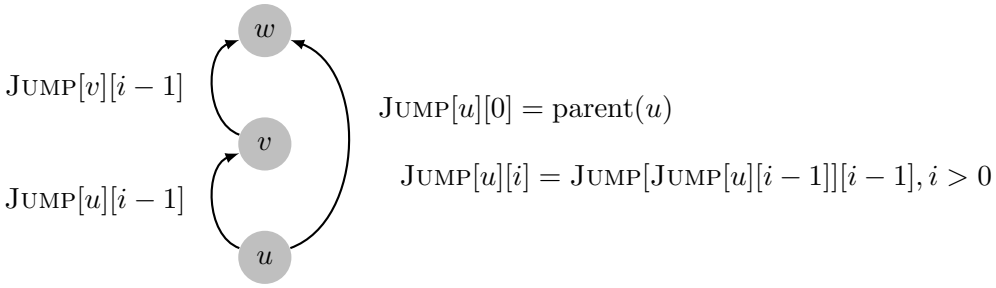
Wir werden uns - wie so häufig - schrittweiße der Lösung nähern, indem wir zunächst zwei Lösungen mit logarithmischer Laufzeit diskutieren und werden daraufhin einen Ansatz vorstellen, um diese beiden Techniken zu verbinden.

1.3.1 Jump-Pointer Algorithmus

Die grundlegende Idee ist es, ähnlich einer Skip Liste, gerade so viele Lösungen (oder Jump-Pointer) zu speichern, um gerade in logarithmisch vielen Schritten am Ziel anzukommen. In jedem Knoten u wird $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ also nur für $d = 1, 2, 4 \ldots, \lfloor\lfloor \operatorname{depth}(u) \rfloor\rfloor$ anstatt für alle Vorgänger gespeichert. Hier bezeichne $\textsc{Jump}[u][i] := \textsc{LevelAncestor}_{\mathrm{T}}(u, \operatorname{depth}(u) - 2^i)$ den i-ten Jump-Pointer des Knotens u.

Die zentrale Beobachtung ist es nun, dass mit jedem *korrekten* Sprung mindestens die halbe Strecke zurückgelegt wird: Sei hierfür $\delta = \operatorname{depth}(u) - d$ die zurückzulegende Distanz, dann überspringt $\textsc{Jump}[u][\lfloor\lfloor \lg \delta \rfloor\rfloor]$ eben $\lfloor\lfloor \delta \rfloor\rfloor \geq \frac{\delta}{2}$ Knoten. Der Algorithmus muss nun lediglich diesen Jump-Pointern folgen bis $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ erreicht ist.

Dies führt zu einer logarithmischen Laufzeit bei einem Platzverbrauch von $\mathcal{O}(n \lg n)$, wobei allerdings jeder Knoten $\leq \lg n$ zusätzlichen Speicher benötigt. Die Vorverarbeitung kann mittels dynamischer Programmierung (Top-Down) in $\mathcal{O}(n \lg n)$ Zeit durchgeführt werden:



1.3.2 Ladder Algorithmus

Für den *Ladder Algorithmus* zerlegen wir zunächst den Baum in lange Pfade, indem wir schrittweiße den längsten Pfad entfernen. Das zerlegt den Baum in Teilbäume $T_1, T_2, \ldots$, die wiederum rekursiv zerlegt werden.

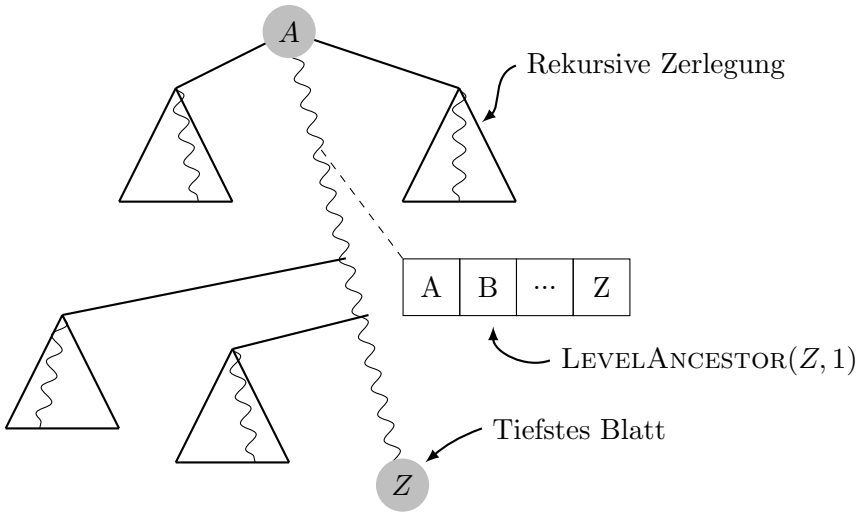


Abbildung 1: Ein längster Wurzel-Blatt Pfad, der den Baum T in vier Teilbäume zerlegt. Die Tabelle zeigt wie eine LevelAncestor-Anfragen entlang des Pfades beantwortet werden kann.

Dies kann in $\mathcal{O}(n)$ Zeit erreicht werden, indem zunächst für jeden Knoten, ausgehend von den Blättern (Bottom-Up), die größte Knoten-Blatt Entfernung berechnet wird. In einem weiteren Durchlauf, beginnend bei der Wurzel, kann nun für jeden Knoten der Nachfolger als das Kind mit der größten Knoten-Blatt Entfernung gewählt werden.

Entlang eines Pfades ist es *einfach*, den Level Ancestor zu finden: Falls die Knoten des Pfades π der Länge m in einem Array $\textsc{Ladder}_\pi[0, m-1]$ gespeichert sind und $\textsc{Ladder}_\pi[0]$ auf Tiefe h liegt, dann ist $\textsc{LevelAncestor}_\mathrm{T}(u, d) = \textsc{Ladder}_\pi[d-h]$ mit $d \geq h$ und $u \in \pi$. Ansonsten ist $d < h$ und wir können zu dem Elternpfad π' von π springen bis $d \geq h'$

Dies führt zu einer Laufzeit von $\mathcal{O}(\sqrt{n})$ da bis zu $\Theta(\sqrt{n})$ Pfade auf einem Weg von Wurzel zu Blatt liegen können.

Um dies auf $\mathcal{O}(\lg n)$ zu drücken, erweitern wir die Pfade zu Leitern (*Ladders*). Sei $|\pi| = m$, dann speichere in $\textsc{Ladder}_\pi$ nicht nur π, sondern zusätzlich die m direkten Vorgänger von $\pi[0]$.

Der Vorteil dieser Konstruktion ist, dass falls sich $\textsc{LevelAncestor}_\mathrm{T}(u, d)$ nicht auf der Leiter befindet, dann ist $\textsc{Ladder}_\pi[0]$ mindestens *doppelt* so weit vom tiefsten Blatt im Teilbaum des Knotens u entfernt wie der Knoten u selbst.

Weiterhin kann wiederholt auf die nächsthöhere Leiter gewechselt werden, bis $\textsc{LevelAncestor}_\mathrm{T}(u, d)$ auf der Leiter liegt, und da sich mit jedem Wechsel der Leiter die Entfernung zu den Blättern verdoppelt führt dies zu einer logarithmischen Laufzeit.

1.3.3 Beide Techniken verbinden

Konstante Laufzeit kann erreicht werden indem man beide Techniken verbindet. Hierfür springt man zunächst den halben Weg mit $\textsc{Jump}[u][\lfloor \delta \rfloor]$ nach oben. Man erreicht auf diese Weiße einen Knoten v mit $\mathit{height}(v) \geq \lfloor \lfloor \delta \rfloor \rfloor$. Da nun die Entfernung von v nach $\textsc{LevelAncestor}_\mathrm{T}(u, d)$ kleiner als δ ist, ist $\textsc{LevelAncestor}_\mathrm{T}(u, d)$ in der Leiter des Knotens v enthalten und man erreicht es in $\mathcal{O}(1)$.

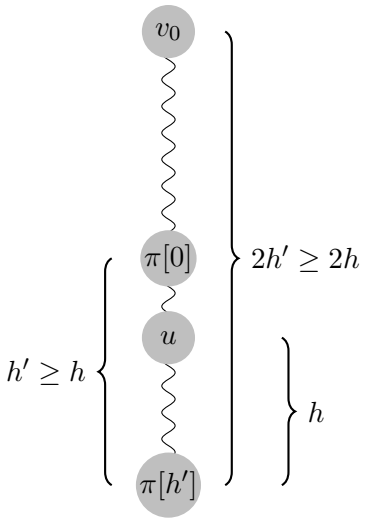


Abbildung 2: Diese Abbildung zeigt schematisch eine Leiter, die aus dem Pfad π entstanden ist. Hier sind h' die Länge des Pfades π, deshalb $2h'$ die Länge der Leiter und h die Entfernung von u zu $\pi[h']$.

1.4 $\mathcal{O}(1)$ Level Ancestor mit $\mathcal{O}(n)$ Platzverbrauch

Der aktuelle Platzverbrauch ist in den $\mathcal{O}(n \lg n)$ Jump-Pointern begründet, jedoch müssen diese nicht in *jedem* Knoten gespeichert werden, da $\textsc{LevelAncestor}_T(v, d) = \textsc{LevelAncestor}_T(w, d)$ für jeden Nachfolger w von v. Falls man also eine Menge von Jump Nodes, welche mit Pointern ausgestattet sind, auswählt, kann *jeder* Knoten *oberhalb* eines Jump Nodes diesen benutzen um die Anfrage zu beantworten. Alle anderen (also jene unterhalb der Jump Nodes) werden am Ende des Abschnitts behandelt.

Die Idee Jump Nodes zu bestimmen ist angelehnt an y-fast tries: *Wähle als Jump Nodes* die tiefsten Knoten die mindestens $s := \lg n/4$ Nachfolger besitzen (Was dazu führt, dass jedes Kind eines Jump Nodes weniger als s Knoten in seinem Teilbaum besitzt). Darüber hinaus muss für jeden Knoten überhalb der Jump Nodes ein Zeiger JUMP-DESC$[u]$ auf einen der nachfolgenden Jump Nodes gespeichert werden.

Wir werden im Folgenden den Teil des Baumes T oberhalb der Jump Nodes als *macro tree*, und die Teile unterhalb der Jump Nodes als *micro trees* bezeichnen. Da die Jump Nodes Wurzeln von disjunkten Teilbäumen mindestens der Größe s sind, gibt es höchstens n/s Jump Nodes. Deshalb ist der gesamte Platzverbrauch um alle Jump Pointer zu speichern höchstens

$$\frac{n}{s} \lg \frac{n}{s} \leq \frac{n}{s} \lg n = \mathcal{O}(\frac{n}{\lg n} \lg n) = \mathcal{O}(n)$$

und damit linear.

Die Jump Pointer können effizient in Linearzeit bestimmt werden, indem von dem jeweiligen Jump Node wiederholt den Leitern gefolgt wird. Das heißt, sobald die aktuelle Leiter nicht mehr ausreicht um die gerade zu speichernde Anfrage beantworten zu können, wechselt man auf die nächsthöhere. Da allerdings höchstens logarithmisch viele solcher Leiterwechsel benötigt werden

um pro Knoten die logarithmisch vielen Jump Pointer zu bestimmen, kann jeder Jump Pointer in amortisiert konstanter Zeit bestimmt werden.

Um nun auch noch $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ für alle Micro Trees beantworten zu können, kodiert man alle möglichen Bäume mit $s' < s$ Knoten als ein Bitmuster der Größe $2(s' - 1)$: Schreibe eine '0' falls man in einem DFS Schritt auf dem Baum nach unten geht und eine '1' falls man nach oben geht.

Nun speichert jeder Micro Tree sein Bitmuster (mit hinten aufgefüllten Nullen um eine Länge von $2(s-1)$ zu erreichen), einen Zeiger auf seinen nächsten Vorgänger im Macro Tree und eine Tabelle um DFS Nummern in globale Schlüssel umzurechnen. Zusätzlich wird eine *globale Lookup-Tabelle* benötigt, die für jedes auftretende Bitmuster die Antworten zu allen möglichen LevelAncestor-Anfragen enthält, wobei jede Antwort als DFS-Nummer in diesem spezifischen Baum gegeben ist. Zusammen mit der Transformationstabelle lassen sich nun alle $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ Anfragen für u in einem Micro Tree und $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ im selben Micro Tree beantworten. Falls $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ sich nicht in diesem Micro Tree befindet, wird die Anfrage durch den im Micro Tree gespeicherten Vorgänger im Macro Tree beantwortet.

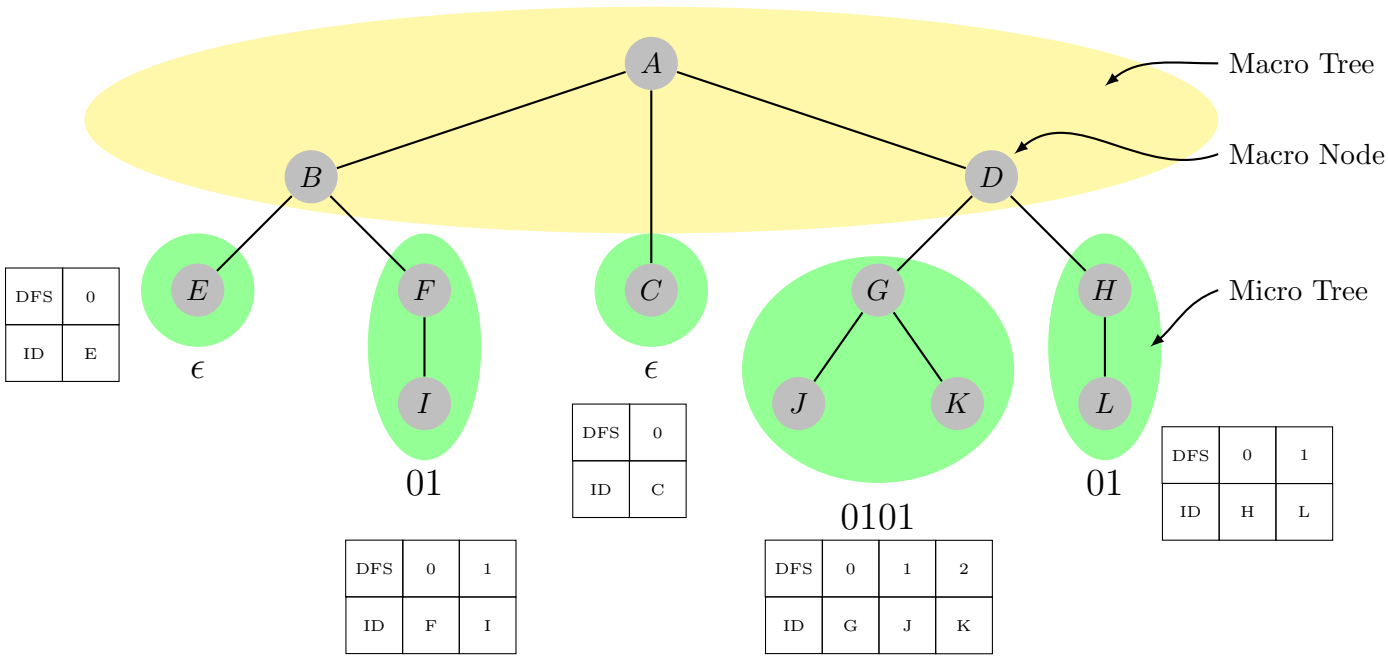
Es ist nun also möglich in konstanter Laufzeit LevelAncestor-Anfragen für alle Knoten vom Typ $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ zu beantworten, wobei sich für die Größe der Tabellen ergibt:

$$\#\text{ Bitmuster der Länge } 2s \times \#\text{ der } u\text{'s} \times \#\text{ der } d\text{'s}$$

$$2^{2s} \times s \times s$$

was gerade $\mathcal{O}(2^{\lg n/2} s^2) = \mathcal{O}(\sqrt{n} \lg^2 n) = o(n)$ ergibt.

$s = 4$



bit pattern	$u =$ / $d =$	0 0 1 2	1 0 1 2	2 0 1 2	
0	000000	0⊥⊥	⊥⊥⊥	⊥⊥⊥	
	000001	? ? ?	? ? ?	? ? ?	keine Micro Trees
	000010	? ? ?	? ? ?	? ? ?	
0	⋮	?	?	?	
1	001100	0⊥⊥	01⊥	012	
	⋮	?	?	?	keine Micro Trees
2	010000	0⊥⊥	01⊥	⊥⊥⊥	
	010001	? ? ?	? ? ?	? ? ?	
	010010	? ? ?	? ? ?	? ? ?	keine Micro Trees
0	010011	? ? ?	? ? ?	? ? ?	
	010100	0⊥⊥	01⊥	02⊥	
1	⋮	?	?	?	
	111111	? ? ?	? ? ?	? ? ?	
0					
1 2					