1st Symposium on Breakthroughs in Advanced Data Structures

BADS'13, April 11th, 2013, Karlsruhe, Germany

Edited by

Johannes Fischer

Editors

Johannes Fischer
Inst. Theor. Computer Science
KIT, Germany
johannes.fischer@kit.edu

ACM Classification 1998
Data Structures

ISBN 999-9-999999-999-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Center for Informatics gGmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.

Publication date
April, 2013

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.9999.9999.9

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

www.dagstuhl.de/lipics

Dedicated to the students of the class
“Advanced Data Structures” (winter 12/13).

Contents

1st Symposium on Breakthroughs in Advanced Data Structures (BADS'13).
Editor: J. Fischer
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the accepted papers of the 1st Symposium on Breakthroughs in Advanced Data Structures, held on April 11th in Karlsruhe, Germany. Competition was particularly fierce this year; out of 8 submissions we could accept only 8, which corresponds to an acceptance rate of only 100%. I wish to thank the program committee members for their hard work and excellent reviews.

Karlsruhe, April 2013
Johannes Fischer

1st Symposium on Breakthroughs in Advanced Data Structures (BADS'13).
Editor: J. Fischer

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Praktische Rank-/ Select Dictionaries Komprimiert mithilfe von Entropie *

Michael Axtmann[1]

1 **Karlsruher Institut für Technologie (KIT)**
Karlsruhe, Deutschland
michael.axtmann3@student.kit.edu

Abstract

Rank-/ Selekt Dictionaries finden häufig Einsatz in succinct data structures (*kleinen Datenstrukturen*). Bisher wurden diese Dictionaries nur asymptotisch und auf generierten Daten analysiert. Dieses Paper stellt 4 *succinct* Strukturen (**esp**, **recRank**, **vcode** und **sdarray**) vor und analysiert diese experimentell.

1998 ACM Subject Classification Dummy classification

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.9999.9999.9

1 Einführng

Die vorgestellten Datenstrukturen berechnen die folgenden Operationen auf einer geordneten Menge $S \subset 0 \dots n-1$:

- **rank(x, S)**: Anzahl an Elementen kleiner x und
- **select(i, S)**: die Position des i'kleinsten Elements.

Für kleine $|S| = m$ ist die Datenstruktur klein, in realistischen Anwendungen benötigt sie $n \cdot H_0(S)$ ($H_0(S) = \frac{m}{n} \log \frac{n}{m} + \frac{n-m}{n} \log \frac{n}{n-m} \leq 1$ ist die Entropie von S) Platz.

Eine *succinct data strutkture* ist eine Datenstruktur, welche jedes Objekt aus einem Universum der Größe L mittels $(1 + o(1)) \cdot \log L$ Bits [1] kodiert. Im Gegensatz zur normalen Kompression beantworten diese Stukturen die Anfragen sehr schnell. Bisher wurden einige Datenstrukturen vorgestellt, die $nH_0(S) + o(n)$ Bits benötigen [2, 6, 9, 10]. Diese wurden jedoch nur asymptotisch analysiert, auch gibt es davon keine praktische Implementierung. *Gap-basierte Ansätze* [4, 5] wurden zwar auch vorgestellt, beantworten **Rank** und **Select** jedoch nicht in konstanter Zeit.

Tabelle 3 zeigt die *worst case* Laufzeiten der vorgestellten Ansätze. Zu beachten ist, dass die Berechnungen von $O(\frac{\log^4 m}{\log n})$ und $O(\log n)$ in **sarray**, **darray** bzw. **vcode** in den meisten Fällen konstante Zeit benötigen! Insgesamt sind die eigenen Ansätze schneller, als die bisherigen Implementierungen.

2 Allgemeines

Die Arbeit analysiert die Algorithmen im *Word RAM Model*. In diesem Modell ist es möglich, logische und arithmetische Operationen auf zwei $O(\log n)$-bit langen Ganzzahlen in konstanter Zeit auszuführen. Ebenfalls konstante Zeit benötigt zugreifen und speichern von $O(\log n)$

* Zusammenfassung des Papers: Practical Entropy-compressed Rank/ Select Dictionary. [8]

1st Symposium on Breakthroughs in Advanced Data Structures (BADS'13).
Editor: J. Fischer; pp. 1–10

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Ansatz	Größe (Bits)	**rank**	**select**
esp (Sek. 3)	$nH_0(S) + o(n)$	$O(1)$	$O(1)$
recRank (Sek. 4)	$1,44m \log \frac{n}{m} + m + o(n)$	$O(\log \frac{n}{m})$	$O(\log \frac{n}{m})$
vcode (Sek. 5)	$m \log \frac{n}{\log n} + O(m)$	$O(\log n^2)$	$O(\log n)$
sarray (Sek. 6)	$m \log \frac{n}{m} + 1,92m + o(m)$	$O(\log \frac{n}{m}) + O(\frac{\log^4 m}{\log n})$	$O(\frac{\log^4 m}{\log n})$
darray (Sek. 6)	$n + o(n)$	$O(1)$	$O(\frac{\log^4 m}{\log n})$

Table 1 Laufzeiten von **esp**, **recRank**, **vcode** und **sdarray**.

fortlaufenden Bits. Die *verbatim* Repräsentation eines Sets S besteht aus einem Bitvektor $B[0 \dots n-1]$ mit $B[i] := 1 \Leftrightarrow i \in S$ sonst $B[i] := 0$. Diese Repräsentation ist *worst case* optimal, da $\log 2^n$ verschiedene Sets existieren und wir $\log 2^n = n$ Bits benötigen um diese eindeutig zu kodieren. Für $m \ll n$ ist diese Repräsentation jedoch um einiges größer als seine untere Schranke $\lceil \log \binom{n}{m} \rceil$ Bits, welche durch $nH_0(B) \sim m \log \frac{n}{m} + 1,44m$ abgeschätzt wird.

Rank und **Select** Anfragen lassen sich in zwei verschiedene Klassen einteilen. *Dense sets* beantworten Anfragen auf Mengen mit $m \sim \frac{n}{2}$ Elementen. Im Gegensatz dazu beantworten *sparse sets* Anfragen auf Mengen mit $m \ll n$ Elementen. Die folgenden Algorithmen sind hauptsächlich auf *sparse sets* optimiert.

3 Existierende Implementierungen von Rank-/ Select Dictionaries

Der erste Abschnitt beschreibt das *Dictionary* **verbatim**, dessen Repräsentation $n + o(n)$ Bits benötigt. Der zweite Teil beschreibt **ent**, eine Erweiterung von **verbatim**, welche $nH_0(S) + o(n)$ Bits benötigt.

3.1 Verbatim

Um den Bitvektor zu codieren, unterteilt das *Dictionary* den Vektor B nacheinander in zwei Rekursionsebenen. Zu Beginn partitionieren wir B in *große Blöcke* der Größe $l = \log^2 n$. Danach wird jeder *große Block* weiter in *kleine Blöcke* der Größe $s = \log \frac{n}{2}$ aufgeteilt. Die großen und kleinen Blöcke werden nicht explizit gespeichert. Direkt abgespeichert wird ausschließlich die Anzahl der Einsen für jede Position in jedem *kleinen Block* in Relation zum Blockanfang der kleinen Blöcke. Formal: $popcount(i, j)$ ist die Anzahl der Einsen im i'ten *kleinen Block* bis zur Position j, welche relativ zu diesem Block berechnet wird. Diese Anfrage kann in konstanter Zeit mithilfe einer vorberechneten Lookup Tabelle der Größe $O(\sqrt{n} \log^2 n)$ oder mithilfe der *Popcount* Funktion [3] beschrieben werden. Weiter speichern wir die Bereichsgrenzen $(rank(l \cdot i))$ für die *großen Blöcke* im Feld $R_t[0 \dots \frac{n}{l}]$. Dafür benötigen wir $O(\frac{n}{\log^2 n} \cdot \log n)$ Bits. Auch die Bereichsgrenzen für die *kleinen Blöcke* werden in einem Feld $R_s[0 \dots \frac{n}{s}]$ hinterlegt. Jedoch wird dazu der **Rank** bezüglich dem *großen Block*, in dem der *kleine Block* liegt, berechnet. Dieses Feld kann in $O(\frac{n \log \log n}{\log n})$ gespeichert werden. **Rank** berechnet sich nun aus $rank(x, S) = R_t[\lfloor \frac{x}{l} \rfloor] + R_s[\lfloor \frac{x}{s} \rfloor] + popcount(\lfloor \frac{x}{s} \rfloor \cdot s, x \mod s)$. Abbildung 2 zeigt exemplarische den Aufbau einer **verbatim** Datenstruktur.

Select ist mit zusätzlichen $o(n)$ Bits in konstanter Zeit möglich [7]. Mittels binärer Suche und mithilfe der **Rank** Funktion kann **Select** ohne zusätzliche Hilfsstrukturen in $O(\log n)$ Zeit [3] beantwortet werden.

B	10001011								11000100								00010001							
R_t	0								4								7							
R_s	0				1				0				2				0				1			
$popcount(i,j)$	1	1	1	1	1	1	2	3	1	2	2	2	0	1	1	1	0	0	0	1	0	0	0	1

Table 2 Beispiel einer **verbatim** Datenstruktur. Der Bitvektor B wird in drei *große Blöcke* unterteilt. Ein *großer Block* besteht wiederum aus 2 *kleinen Blöcken*. Das letzte Array wird entweder mittels *popcount* oder einer Lookup Tabelle abgespeichert und ausgewertet.

Listing 1 Enumerative Code: Berechnet aus einem gegebenen Vektor B der Länge t mit u Einsen einen eindeutigen Repräsentanten aus dem Wertebereich $[0, \binom{t}{u} - 1]$

```
int enumCode(bool* B, int u, int t) {
  int x = 0;
  for (int i = 0; i < t; i++) {
    if (B[i] == 1) {
      x += choose (t - i - 1, u); // binomial coefficient
      u--
    }
  }
  return x;
```

3.2 Ent

Die Datenstruktur **Ent** ist eine Erweiterung vom **verbatim** Dictionary. Dabei ändert sich die Repräsentation der *kleinen Blöcke*. Diese Blöcke werden mithilfe des *Enumerative Code* codiert. Diese Codierung berechnet aus einem t Bit langem Vektor mit u Einsen eine eindeutige Repräsentation im Wertebereich $[0, \binom{t}{u} - 1]$. Diese Kodierungen haben eine Länge von $\lceil \lg \binom{t}{u} \rceil$ Bits. Der Quellcode 1 beschreibt die Berechnung vom *Enumerative Code*. Es reichen weniger als $nH_0(S)$ Bits [9] aus, um alle *kleinen Blöcke* auf diese Weise zu codieren.

Da die einzelnen Blöcke unterschiedlich lang sind, benötigen wir $\frac{n}{\log n}$ Zeiger der Länge $\log \log n$ um diese zu adressieren. Der Speicherbedarf der Zeiger lässt sich durch $O(\frac{n}{\log n} \cdot \log \log n)$ abschätzen. Wir verwenden zum Codieren und Decodieren eine Lookup Tabelle der Größe $O(\sqrt{n} \log^2 n)$. Insgesamt benötigt das **ent** Dictionary $nH_0(S) + o(n)$ Bits.

4 Estimating Pointer Information

Estimating Pointer Information (**esp**) ist eine Erweiterung des vorgestellten Verfahrens **ent** in Kapitel 3.1. Ziel ist es, auf die explizit gespeicherten Zeiger zu verzichten und diesen erst beim Zugriff direkt zu berechnen. Es ist fraglich, ob dies unbedingt nötig ist, da die $o(n)$ Bits für die Pointer bei praktische Eingaben ähnlich dem Speicherplatz d für die enumerative Codierung ist.

4.1 Idee

Im Folgenden wird nun der grundlegende Ansatz beschrieben. Sei $B[0 \ldots n-1]$ ein Bitvektor mit m Einsen. Weiter sei $L(B)$ der *Enumerative Code* eines Vektors B. Für die Länge der Codierung gilt folgende Abschätzung:

▶ **Theorem 1.** $L(B) \leq nH_0(B) + 1$

▶ **Proof 2.** *$nH_0(B)$ benötigt für jede vorhandene Eins $\lg \frac{n}{m}$ Bits und für jede Null $\lg \frac{n}{n-m}$ Bits. $L(B)$ benötigt hingegen insgesamt nur $L(B) = \lceil \lg \binom{n}{m} \rceil$ Bits. Damit sein das Theorem bewiesen.*

B_i $(i = 1 \dots \lceil \frac{n}{u} \rceil)$ partitioniert B in Blöcke der Größe u. Die Partitionen werden separat codiert. Für die Codierung aller entstandenen Blöcke werden

▶ **Theorem 3.** $\sum_{i=1}^{\lceil \frac{n}{u} \rceil} L(B_i) \leq \sum_{i=1}^{\lceil \frac{n}{u} \rceil} (uH_0(B_i) + 1) \leq nH_0(B) + \frac{n}{u} + 1$

Bits benötigt.

▶ **Proof 4.** *Die erste Abschätzung ergibt sich aus Theorem 1. Die zweite Anschätzung ist korrekt, da $H_0(B)$ eine konkave Funktion ist.*

Die fortlaufenden *enumerativen Codierungen* werden bei **esp** jedoch nicht einfach konsekutiv abgespeichert. Stattdessen schätzen wir eine Position, ab der die vorangegangenen Codierungen auf jeden Fall abgeschlossen sind. Dazu sei $P_t = B[0 \dots t]$ $(t < n)$ der Präfix des Vektors B. Die geschätzte Position berechnet sich aus $L(P_{u \cdot i})$, wobei i für die Partitionsnummer steht. Falls zwischen der geschätzten Position und dem Ende der letzten Codierung eine Lücke entsteht, bleiben diese unbenutzt, so dass wir immer die korrekte Position schätzen. $H_0(P_i)$ kann mithilfe von Rank Informationen berechnet werden. Aus diesem Grund müssen die Zeiger nicht explizit abgespeichert werden. Der Speicherbedarf beläuft sich nach Theorem 3 auf $nH_0(B) + \frac{n}{u} + 1 = nH_0(B) + o(n)$ Bits, da wir im Folgenden $u = \log \frac{n}{2}$ wählen und daraus $\frac{n}{u} + 1 = \frac{n}{\log \frac{n}{2}} + 1 = o(n)$ folgt.

4.2 Details der Implementierung

Das **esp** Dictionary stellt eine Erweiterung von **ent** dar. Im Gegensatz dazu baut sich die Datenstruktur aus drei Stufen auf. Die obere Schicht besteht aus *extrem großen Blöcken* (*SLB*), welche den Bitvektor in $k = \log^3 n$ große Bereiche unterteilen. Diese wiederum teilen sich in einer mittleren Schicht in weitere $l = \log^2 n$ *große Blöcke* (*LB*) auf. Eine letzte Ebene aus *kleinen Blöcken* (*SB*) der Größe $s = \log \frac{n}{2}$ bildet die untere Schicht. Die *kleinen Blöcke* werden mithilfe des *enumerative Codes* repräsentiert in einem Feld abgespeichert. *Zeiger* adressieren die erste Kodierung jedes *SLB's* in diesem Feld. Um innerhalb eines *SLBs* einen *LB*, bzw. innerhalb eines *LBs* einen *SB* zu adressieren, wird jeweils ein *Rank-Dictionary* R_l und R_s abgespeichert. Auf diesen beiden Ebenen wird der in Abschnitt 4.1 beschriebene Mechanismus angewandt. Die Lookup Tabelle R_l speichert an Position x_l den Rang $l_r = R_l[x_l]$ des Teilvektors aus B, welcher am *LB* $x_l = \lfloor \frac{x}{l} \rfloor$ endet und am Anfang des übergeordneten *SLB* beginnt. Analog dazu speichert die Lookup Tabelle R_s an Position x_s den Rang $s_r = R_s[x_s]$ des Teilvektors aus B, welcher am *SB* $x_s = \lfloor \frac{x}{s} \rfloor$ endet und am Anfang des übergeordneten *LB* beginnt.

Um die Position des i-ten *SBs* zu bestimmen, benötigen wir zusätzlich zur Position des relevanten *SLB* auch den relativen Offset des *LBs* und des *SBs*. Diese berechnen sich aus:

$lp = H_0(P^l_{x_l}) = l_r \cdot \lg \frac{l \cdot x_l}{l_r} + (l \cdot x_l - l_r) \cdot \lg \frac{l \cdot x_l}{l \cdot x_l - l_r}$

und

$sp = H_0(P^s_{x_s}) = s_r \cdot \lg \frac{s \cdot x_s}{s_r} + (s \cdot x_s - s_r) \cdot \lg \frac{s \cdot x_s}{s \cdot x_s - s_r}$

Dabei steht $P^l_{x_l}$ für den Vektorbereich aus B, welcher vom *SLB* und dem *LB* eingegrenzt wird, in dem sich der gesuchte *SB* befindet. Analog dazu steht $P^s_{x_s}$ für den Vektorbereich aus B, welcher vom *LB* und dem *SB* eingegrenzt wird, in dem sich der gesuchte *SB* befindet. Die Position des i-ten *SBs* im *enumerative Code* Feld ist $slp + lp + sp$. Die für diese Anfrage

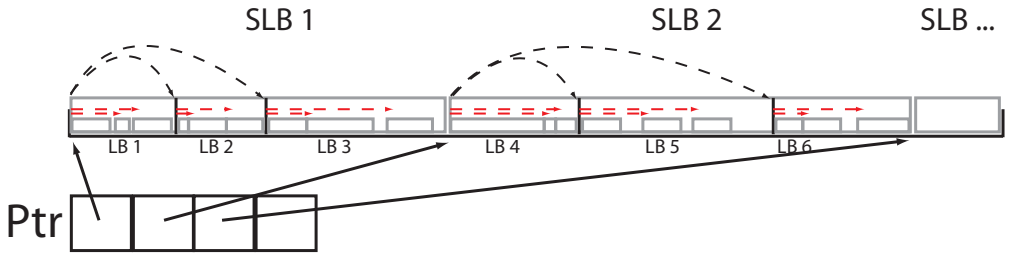


Figure 1 Beispiel einer **exp** Repräsentation. Das Dictionary speichert Zeiger *(Ptr)*, welche auf den Beginn der *SLBs* zeigen. Mithilfe der **Ranks** von *LB* und *SB* lässt sich der Offset der *enumerative Codes* berechnen. Dazu werden die relativen Zeiger (gestrichelte Linien) relativ zum Beginn des übergeordneten Blocks implizit berechnet.

nötigen *Rank Dictionaries* für *LB* und *SB*, sowie die Zeiger für *SLB* werden in insgesamt $o(n)$ Bits gespeichert. Bild 1 zeigt den Zugriff auf die Position eines *kleinen Blocks*.

Um eine **Rank** Anfrage auszuführen, wird der Zeiger auf den Anfang des relevanten *SLBs*, sowie die beiden Offsets vom *SLB* auf den *LB* und vom *LB* auf den *SB* berechnet. Daraus ergibt sich die Position des *enumerative Codes*. Dieser wird wie in Kapitel 3.2 beschrieben in konstanter Zeit mithilfe einer Loopup-Tabelle dekodiert und unter Verwendung der *popcount* Funktion wird, wie in **verbatim**, der richtige Rang berechnet. Die **Select** Anfrage folgt ebenfalls dem in Kapitel 3.2 beschriebenen Vorgehen.

5 RecRank

Diese Datenstruktur basiert ebenfalls auf Rekursion. Ziel ist es, das wahrscheinlich dünn besetzte Eingabefeld mithilfe mehrerer dichten Hilfsfelder selbst auf ein dicht besiedeltes Feld abzubilden.

Der grundlegende Ansatz für einen Komprimierungsschritt wird in [7] Abschnitt 4 *Algorithm 1* beschrieben. Sei dazu $B[0 \dots n-1]$ ein Bitvektor mit m Einen. Wir teilen den Vektor in s Bit lange Blöcke $B_1, \dots, B_{\lceil \frac{n}{s} \rceil}$ auf. Ein *Nullblock* (NB) besteht ausschließlich aus Nullen, ein *Einsblock* (EB) beinhaltet mindestens eine Eins. B wird nicht explizit abgespeichert, sondern in zwei Felder B_c und B_e kleinerer Größe zerlegt. Das kontrahierte Feld B_c bestimmt die Blocktypen in B:

$$B_c[i] = \begin{cases} 1 & \text{if } B_i \text{ ist } EB \\ 0 & \text{if } B_i \text{ ist } NB \end{cases}$$

Das extrahierte Feld B_e speichert fortlaufend ausschließlich die *Einsblöcke*. Mit diesen beiden Vektoren berechnet sich **Rank** auf B wie folgt:

▶ **Theorem 5.** $\textbf{\textit{rank}}(x, B) = \textbf{\textit{rank}}(\textbf{\textit{rank}}(\frac{x}{s}, B_c) \cdot s + (x \mod s) \cdot B_c[\lfloor \frac{x}{s} \rfloor], B_e)$

Dabei berechnet $\textbf{rank}(\frac{x}{s}, B_c)$ die Anzahl der EB bis x und $(x \mod s) \cdot B_c[\lfloor \frac{x}{s} \rfloor]$ die relevanten Bits im letzten Block. Bei geeigneter Wahl der Blockgröße ist B_c ein dicht besetztes Feld. Solange B_e noch nicht dicht ist, wird die Zerlegung auf B_e rekursiv fortgeführt. Somit entstehen in t Aufteilungen die Felder $B_c^1 \dots B_c^t$ und $B_e^1 \dots B_e^t$, wobei B_c^i und B_e^i die kontrahierten und extrahierten Felder im Rekursionsschritt i beschreiben. Ausschließlich das letzte extrahierte Feld wird abgespeichert. Bild 2 veranschaulicht die Datenstruktur.

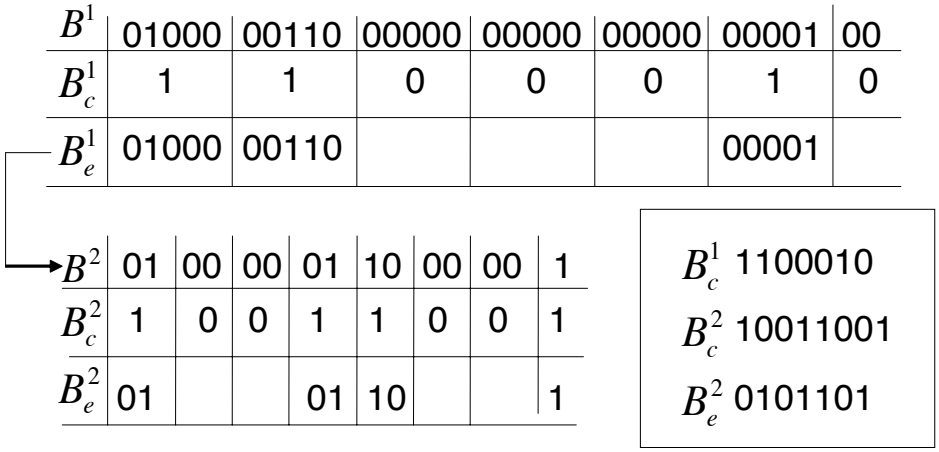


Figure 2 Beispiel einer **recRank** Datenstruktur. Ausgangspunkt ist das Eingabefeld B^1. Dieses Feld wird in die kontrahierten und extrahierten Felder B^1_c und B^1_e zerlegt. Das dünn besetzte extrahierte Feld B^1_e wird in einem zweiten Rekursionsschritt in B^2_c und B^2_e zerlegt. Da das extrahierte Feld B^1_e aus den Feldern B^2_e und B^2_c berechnet werden kann, muss B^1_e nicht in die Datenstruktur (rechter Kasten) aufgenommen werden. *Quelle: Artikel [8]*

.

5.1 Details der Implementierung

Ein dichte besetztes Feld beinhaltet mindestens genauso viele Einsen wie Nullen. Mit geeigneter Blockgröße s wird gewährleistet, dass die Wahrscheinlichkeit einer Eins an einer beliebigen Position im Feld B^i_c gleich $\frac{1}{2}$ ist. Somit ist der neu generierte kontrahierte Vektor immer dicht. Die Rekursion wird abgebrochen, sobald die Wahrscheinlichkeit einer Eins an einer beliebigen Position im extrahierten Vektor B^i_e mindestens $\frac{1}{4}$ beträgt. Sei dazu $P(B) = \frac{m}{n}$ die Wahrscheinlichkeit von Einsen in B. Dann gilt:

▶ **Theorem 6.** *Bei einer Blockgröße von* $s = \frac{1}{-\lg(1-p)}$ *ist die geforderte Dichte gegeben, da* $p(B_c) = \frac{1}{2}$ *gilt.*

Proof. Die Wahrscheinlichkeit, dass in einem Bereich alle Einträge Null sind beträgt $(1-p)^s$. Die Schlussfolgerung $p(B_c) = 0.5$ ist genau dann erfüllt, falls $(1-p)^s = \frac{1}{2}$. Dies ist durch die gegebene Bedingung gewährleistet. ◀

Die Hälfte der Blöcke in B sind *Nullblöcke*, da $P(B_c) = \frac{1}{2}$. Somit halbiert sich die Größe vom extrahierten Feld im Verhältnis zum Ausgangsvektor ($|B_e| = \frac{n}{2}$), die Anzahl der Einsen bleibt jedoch gleich. Somit gilt also $P(B_e) = 2P(B)$, da nur halb so viele Blöcke gespeichert werden. Da sich somit die Wahrscheinlichkeit $p(B^i_e)$ in jeder Rekursion i verdoppelt, reichen insgesamt $t = -\lg p$ Rekursionsschritte aus, um die geforderte Dichte des extrahierten Feldes zu erreichen. Bild 2 zeigt den Aufbau einer **recRank** Datenstruktur nach diesem Vorgehen. Im ersten Schritt beträgt die Blockgröße $5 = \lfloor \frac{1}{-\lg 1-\frac{4}{32}} \rfloor$, da der Vektor B^1 32 Bit lang ist um 4 Einsen enthält. $B^1_e = B^2$ wird im zweiten Schritt in Blöcke der Größe $2 = \lfloor \frac{1}{-\lg 1-\frac{4}{15}} \rfloor$ zerlegt.

Die erwartete Größe von **recRank** beträgt $1,44m \lg \frac{n}{m} + m$ Bits ([8]). Die letztendliche Größe hängt vom Bitvektor selbst ab. Um **Rank** auf dem Feld B zu berechnen, wird die Vorschrift 5 rekursiv auf B^i_c angewandt und zum Schluss der **Rank** im dichten extrahierten Feld B^t_e berechnet. Jede Berechnung benötigt konstante Zeit auf jeder der $-\lg p = \lg \frac{n}{m}$

Ebenen. Somit beantwortet die Datenstruktur eine Anfrage in $O(\log \frac{n}{m})$ Zeit. Im Paper [8] wird jedoch nicht auf die Zeit in der untersten Rekursionsebene eingegangen. Dies ist nach Meinung des Autors dieses Artikels nur zulässig, falls der Basisfall mit einem Vorgehen in mindestens $O(\log \frac{n}{m})$ berechnet werden kann. Davon ist auszugehen. Um **Select** zu berechnen, greifen wir auf die Methode in den vorigen Kapiteln zurück. Dabei wird **Select** auf jeder Ebene berechnet. Somit ist **Select** auch in $O(\log \frac{n}{m})$ auswertbar.

6 Vertical Code

In der Praxis erweist sich Vertical code (**vcode**) als eine schnelle und platzsparende Methode, **Select**-Anfragen auszuwerten. **Vcode** ist eine opportunistische Datenstruktur, dessen Platzbedarf bei realen Eingaben häufig nahe dessen Entropie liegt, im Worst-Case jedoch keine auf der Entropie von B komprimierte Datenstruktur ist.

Die Datenstruktur ergibt sich aus den Arrays $D, T, V_i[j]$ und S. Das Array D muss nicht gespeichert werden, da es sich implizit aus $V_i[j]$ ergibt. Im Folgenden wird beschrieben, wie die Arrays aufgebaut werden und die **Select**-Anfrage effizient ausgewertet wird.

Für jede Eins $i = 0 \dots m-1$ speichert $D[i] = select(B, i+1) - select(b, i) - 1$ (mit $select(b, 0) := 0$) die Anzahl an Nullen zwischen der i'ten und i+1'ten Eins im Array B[1]. Nun wird das Array D in Blöcke der Größe $p \in O(\log^2 n)$ aufgeteilt und weiter verarbeitet. Dabei speichert $T[k] = \lg\lceil max_{j=0\dots p-1} d[kp+j]\rceil$ für jeden Block k ($k = 0 \dots \lceil \frac{m}{p} \rceil$) die maximale Länge der Binärrepräsentationen der Null-Abstände zweier Einsen im Block k. Die Bitfelder $V_k[l]$ repräsentieren nun die Werte aus D wie folgt: Sei dazu $D[pk+i][l]$ das l'te Bit der Zahl $D[pb+i]$ im Block b an Position i. Jeder Block k verweist zur Kodierung von D auf $l = 0 \dots T[k]-1$ Bitvektoren $V_k[l]$. Jeder dieser Bitvektoren $V_k[l]$ speichert an Position i ($V_k[l][i]$) das Bit $D[pk+i][l]$. Somit speichert $V_k[l][i]$ das l'te Bit des i'ten Eintrags in D bzgl. des Blocks k. Darüber hinaus speichert $S[i] = select(B, ip)$ die Position der Eins, welche Block $i-1$ terminiert.

Wir definieren nun $q := i \mod p$ als die q'te Eins in Block $b := \lfloor \frac{i}{p} \rfloor$, in dem sich die i'te Eins befindet. Die **Select**-Anfrage brechnet sich nun wie folgt:

▶ **Theorem 7.** $select(B, i)$
$= select(B, \lfloor \frac{i}{p} \rfloor) + [(select(B, i) - select(B, \lfloor \frac{i}{p} \rfloor))]$
$= S[b] + \sum_{j=0}^{q} (D[j+pb] + 1)$
$= S[b] + (q + \sum_{j=0}^{q} D[j+pb])$

Die Anzahl an Nullen im letzten Teilblock bis zur Position i ($\sum_{j=0}^{q} D[j+pb]$) müssen in diesem Fall aus den Bitfeldern $V_k[l]$ berechnet werden.

Da $V_b[k][j]$ das k'te Bit von $D[bp+j]$ speichert, ergibt sich $D[bp+j]$ aus $\sum_{l=0}^{t[b]-1} V_b[l][j] \cdot 2^l$. Die gesuchte Anzahl an Nullen berechnet sich nun aus

▶ **Theorem 8.** $\sum_{j=0}^{q} D[pb+j] = \sum_{j=0}^{q} \sum_{l=0}^{t[b]-1} V_b[l][j] \cdot 2^l = \sum_{l=0}^{t[b]-1} \sum_{j=0}^{q} V_b[l][j] \cdot 2^l$.

Code 2 beschreibt die **Select**-Operation. Diese Implementierung addiert die innere Summe von Theorem 8 mittels *popcount* auf. Der Vorteil von **vcode** ist, dass die Kosten von $\sum_{j=0}^{q} D[j+pb]$ in $O(T[b])$ liegen, falls die Blockgröße p ein Vielfaches von acht ist. Diese Abschätzung ist korrekt, da die benötigten Operationen in diesem Fall *byte-aligned* sind.

[1] Gezählt wird ab der ersten Eins. Die 0'te Eins wird am Position -1 implizit angenommen.

Listing 2 Vertical Code: Der Quellcode wertet eine Select-Anfrage aus. Zu Beginn wird der Block-Offset berechnet. Im Anschluss werden die verbleibenden Einsen (q) addiert, danach die verbleibenden Nullen aufsummiert.

```
int select_vc(int i) // Wertet select(i, B) aus.
  int b = i / p; // Blocknummer
  int q = i % p; // Offset im Block b
  for (int j = 0; j < T[b]; j++) // Summiere j'te Bits auf.
    // Addiere Einsen und verschiebe (Potenz).
    x += popcount[V[b][j] & ((1U << q) - 1)] << j;
  return x;
```

0011|0011|1001|0000|1000|0010|1000|11												
i	0	1	2	3	4	5	6	7	8	9	10	
D	2	0	2	0	0	2	4	5	1	3	0	
T	$\lg 2 = 2$				$\lg 5 = 3$				$\lg 3 = 2$			
$V_0[0]$	0	0	0	0	$V_1[0]$	0	0	0	1	$V_2[0]$	1	1	0
$V_0[1]$	1	0	1	1	$V_1[1]$	0	1	0	0	$V_2[1]$	0	1	0
$V_0[2]$					$V_1[2]$	0	0	1	1	$V_2[2]$			

Table 3 Die Tabelle zeigt exemplarisch ein **Vcode**-Dictionary. Die Blockgröße beträgt 4 Elemente. Der Ausgangsvektor besteht aus 32 Bits und wurde zur besseren Lesbarkeit mit Platzhaltern getrennt. Um die Anschaulichkeit zu erhöhen, ist der Vektor D abgebildet. Dieser wird nicht im Dictionary gespeichert, sondern aus den Bitvektoren $V_i[j]$ berechnet.

Vcode benötigt $T[b]$ Verschiebungen und Additionen, welche jeweils in konstanter Zeit ausgeführt werden können. Die Kodierung einer Nullsequenz kann jedoch im schlechtesten Fall $T[b] \in \theta(\log n)$ betragen. Die Ausführungszeit für **Select** beläuft sich daher auf $O(\log n)$. Um **Rank** und **Select**$_0$ zu beantworten, muss auf den m Elementen möglichen Einsen in der Datenstruktur binäre Suche ausgeführt werden. Daraus ergibt sich im *Worst Case* eine Laufzeit von $O(\log n \cdot \log m)$.

Die Größe von S ist $O(\log n \cdot \frac{m}{\log^2 n}) \in o(n)$, da S für jeden der $\frac{m}{\log^2 n}$ Blöcke maximal $\log n$ Bits speichert. Da der Abstand zwischen zwei Einsen $D[i] < n$ ist, beschränkt sich T ebenfalls auf $O(\log n \cdot \frac{m}{\log^2} n) \in o(n)$. V benötigt am meisten Speicher, falls $D[ip] = \frac{n}{\log^2 n} (0 \leq i < \frac{n}{p})$ und die restlichen $D[i]$ Null sind. In diesem Fall ist V genau $m \frac{\lg n}{\lg^2 n}$ Bits groß. Für reale Eingaben ist jedoch eine derartige Verteilung der Nullen zu erwarten (Werte aus D in einem Block haben ähnliche Werte), so dass der Platzbedarf von V nahe $m \lg \frac{n}{m} \sim nH_0(B)$ liegt und die Anfragezeit in $O(1)$.

7 SDarray

Die Implementierung vom **SDarray** wird in [8] beschrieben. Je nach Dichte des Feldes auf dem die Anfragen ausgeführt wird, kommen zwei unterschiedliche Datenstrukturen **sarray** und **darray** zum Einsatz. Diese sich speziell auf *dünn besetzte*, bzw. *dicht besetzte* Felder optimiert.

8 Experimentelle Ergebnisse

Dieses Kapitel versucht ohne Verwendung von eigenen Messwerten die experimentellen Ergebnisse aus [8] zu beschreiben. Da die Algorithmen nicht selbst implementiert wurden und die Grafiken aus dem Grundlagenpapier geschützt sind, wird an dieser Stelle darauf verzichtet.

Für die Experimente wurden die Datenstrukturen **esp** ($k = 2^{12}, l = 2^8, s = 2^5$), **recrank**, **vcode** ($p = 8$), **sarray** ($L = 2^{10} L_2 = 2^{16}$ und $L_3 = 2^5$) und **darray** verglichen. Diese Implementierungen werden zusätzlich mit den Byte-basierenden Ansätzen **Kim** [7], **Kim2** [3] und **navarro** [3] verglichen. Genauere Details der Testumgebung sind im Grundlagenpapier [8] nachzulesen.

Bei den Experimenten wurden jeweils Felder der Größe $10 \cdot 2^{20}$ Bits verwendet.

Zu Beginn wird die Effizienz der Komprimierung analysiert. Dabei stellte dich heraus, dass **esp** besonders nahe am optimalen Platzbedarf von $nH_0(B)$ liegt. Für besonders dichte Felder sind ebenfalls die *Dictionaries* **recrank**, **sarray** und **vcode** nahe am Optimum.

Für besonders dünn besetzte Felder benötigt **sarray** am wenigsten Platz und liegt mit $15,05$ zu $10,13$ bei 1% Einsen und mit $40,59$ zu $28,62$ bei 5% Einsen sehr nahe am optimalen Platzverbrauch von nH_0.

Kim2, **Navarro** und **darray** führen 10^8 **rank**-Operationen auf unterschiedlich langen Bitvektoren in nahezu konstanter Zeit aus. **Vcode** und **esp** beantworten die Anfragen sehr langsam und verlaufen auch extrem nichtlinear. Dieses Verhalten lässt sich bei **vcode** durch die *binäre Suche* erklären. Für kleine $\frac{m}{n}$ ist einzig **recrank** langsamer, da die Anfrage $O(\log \frac{n}{m})$ Zeit benötigt. Bei konstantem n (hier fest vorgegeben), ist die Laufzeit proportional zu $\log m$.

Die Analyse der Laufzeit von 10^8 **select**-Operationen gestaltet sich schwieriger. **sarray** beantwortet die Anfragen trotz leicht steigender Antwortzeit am schnellsten. **Darray** ist die zweit schnellste Datenstruktur. Jedoch treten hier für dünn besetzte Felder die gleichen Probleme wie bei der **rank**-Anfrage auf.

9 Zusammenfassung

Dieses Paper fasst die Erkenntnisse aus dem Grundlagenpaper noch einmal in anderer Form zusammen. Es wurden verschiedene Datenstrukturen vorgestellt, welche sich teilweise im Platzbedarf und in der Anfragezeit unterscheiden. Jedoch konnte gezeigt werden, dass diese für reale Eingaben sehr effizient sind. Im Vergleich zu anderen **rank**-Datenstrukturen hält sich die Komplexität der Implementierung im Rahmen und ist deshalb praktisch anwendbar. Gängige Ansätze der Algorithmentechnik wie Reduktion bei **recrank** und spezifische Datenstrukturen für dünn-/ dicht besiedelte Felder bei **sdarray** haben sich bewährt.

References

1 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

2 Alexander Golynski. Optimal lower bounds for rank and select indexes. *Theor. Comput. Sci.*, 387(3):348–359, 2007.

3 Rodrigo Gonzalez, Szymon Grabowski, Veli Maekinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)(Greece, 2005)*, pages 27–38, 2005.

4 Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In *WEA*, pages 158–169, 2006.

5 Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007.

6 Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554, 1989.

7 Dong Kim, Joong Na, Ji Kim, and Kunsoo Park. Efficient implementation of rank and select functions for succinct representation. *Experimental and Efficient Algorithms*, pages 125–143, 2005.

8 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. *CoRR*, abs/cs/0610001, 2006.

9 Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

10 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *CoRR*, abs/0705.0552, 2007.

Top-K Color Queries for Document Retrieval

Elias Bordolo[1]

1 **Karlsruher Institut für Technologie (KIT)**
Kaiserstraße 12, 76131 Karlsruhe, Deutschland
elias.bordolo@student.kit.edu

Zusammenfassung

In diesem Artikel wird eine effiziente Datenstruktur für das *top-K color problem* beschrieben. Jedem Element eines Arrays A wird eine Farbe c mit Priorität $p(c)$ zugewiesen. Für ein Anfrageintervall $[a, b]$ und einen Wert K sollen K verschiedene Farben aus dem Intervall $A[a, b]$ mit den höchsten Prioritäten aller Farben aus diesem Intervall ausgegeben werden. Die Ausgabe soll nach Priorität sortiert sein. Es wird gezeigt, dass derartige Anfragen in $O(K)$ Zeit mit einer Datenstruktur der Größe $O(N \log \sigma)$ Bits beantwortet werden können. Die Anzahl der Elemente in A wird mit N und die Anzahl der Farben mit σ bezeichnet. Bei diesem Paper handelt es sich um eine Zusammenfassung des Artikels „Top-K Color Queries for Document Retrieval“ von Karpinski und Nekrich [1].

1998 ACM Subject Classification Dummy classification

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.9999.9999.9

1 Einführung

In diesem Artikel wird eine Variante des bekannten *color reporting problem* studiert. Jedem Eintrag eines Arrays A wird darin eine Farbe $c \in C$ mit Priorität $p(c)$ zugewiesen. Für ein Anfrageintervall $Q = [a, b]$ und eine Ganzzahl K soll die Datenstruktur K verschiedene Farben mit den höchsten Prioritäten aller Farben ausgeben, die im Intervall Q vorkommen. Die Untersuchung dieses Problems ist motiviert durch seine Anwendungsmöglichkeiten in den Bereichen Dokument-Retrieval und Suchmaschinen.

Das Problem alle Farben (und damit Dokumente) aus einem Intervall $A[i, j]$ auszugeben, die einen String P enthalten, ist bekannt [3]. Da die Anzahl der Dokumente, die P enthalten, aber sehr groß sein kann, sollen häufig (z.B. in Suchmaschinen) nur die Dokumente mit höchster Relevanz ausgegeben werden. Hon et al. [2] stellen ein Framework vor, mit dem die K relevantesten Dokumente bezüglich eines Anfragestrings P ausgegeben werden können. Ihre Datenstruktur benötigt $O(N \log N)$ Bits Platz und beantwortet Fragen in $O(|P| + K \log K)$ Zeit. Für die Relevanz können verschiedenste Metriken wie z.B. Häufgkeit, minimale Distanz zwischen zwei Vorkommen von P innerhalb eines Dokuments oder der statische PageRank zum Einsatz kommen.

Die in diesem Artikel vorgestellte Datenstruktur benötigt $O(N \log \sigma)$ Bits und kann für eine beliebige Ganzzahl K bis zu K verschiedene Farben mit höchsten Prioritäten (*top-K color query*) aus einem beliebigen Intervall $[a, b]$ in A in optimaler Zeit $O(K)$ ausgeben. Es ist dabei nicht zwingend notwendig, K im Vorfeld zu kennen. Die Farben aus $A[a, b]$ können in absteigender Reihenfolge ihrer Prioritäten ausgegeben werden bis entweder alle Farben ausgegeben wurden, oder die Anfrage vom Anwender beendet wird.

In Abschnitt 2 werden die Probleme *color reporting* und *color counting* definiert. Abschnitt 3 beschreibt eine einfache Datenstruktur, die eine unsortierte Liste der K Farben mit

1st Symposium on Breakthroughs in Advanced Data Structures (BADS'13).
Editor: J. Fischer; pp. 11–19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

höchster Priorität in $O(K + \log^2 N)$ Zeit unter Verwendung von $O(N \log^2 N)$ Bits ausgeben kann. Der Platzbedarf dieser Datenstruktur wird in Abschnitt 4 auf $O(N \log N)$ Bits (linear) verringert, jedoch schlägt sich das in der Anfragezeit nieder. In Abschnitt 5 wird diese auf optimale $O(K)$ gesenkt. Abschnitt 6 verbessert schließlich den Platzbedarf noch weiter von $O(N \log N)$ auf $O(N \log \sigma)$ Bits.

2 Color Reporting und Color Counting

Beim Problem *color reporting* wird jedem Element eines Arrays A eine Farbe c aus der Menge der Farben C zugewiesen. Ist ein Anfrageintervall $Q = A[a, b]$ gegeben, sollen alle paarweise verschiedenen Farben $c_1, \ldots, c_K$ aus Q ausgegeben werden. Dabei muss mindestens ein Element der Farbe c_i für $1 \leq i \leq K$ in Q vorkommen. Im Problem *color counting* soll die Anzahl der paarweise verschiedenen Farben gezählt werden, die in Q existieren. Beide Probleme wurden in [4] ausführlich untersucht. An dieser Stelle werden nur zwei Lemmas zitiert, deren Beweise sind in [4] oder [1] zu finden.

▶ **Lemma 1.** *Im RAM-Modell können* colored range reporting queries *in $O(K)$ Zeit mit einer $O(N \log N)$ Bits großen Datenstruktur beantwortet werden. Im RAM-Modell können* colored range counting queries *von einer $O(N \log N)$ Bits-Datenstruktur in $O(\log N)$ Zeit beantwortet werden.*

▶ **Lemma 2.** *Im externen Speicher-Modell können colored range reporting queries mit $O(\log \log_B N + K/B)$ I/Os mit einer $O(N \log N)$ Bits großen Datenstruktur beantwortet werden. Im externen Speicher-Modell können colored range counting queries von einer $O(N \log N)$ Bits-Datenstruktur mit $O(\log N)$ I/Os beantwortet werden.*

3 Eine Datenstruktur mit $O(N \log^2 N)$ Bits Platzbedarf

In diesem Abschnitt wird folgendes Problem betrachtet: Gebe alle K Farben mit den höchsten Prioritäten im Anfrageintervall $[a, b]$ in zufälliger Reihenfolge aus. Hierfür wird die Menge aller Farben rekursiv entsprechend ihrer Prioritäten aufgeteilt. Die verwendete Datenstruktur ist ein Wavelet-Tree, erweitert um zusätzliche sekundäre Datenstrukturen $COUNT_v$ und REP_v, die das Zählen bzw. Ausgeben der Farben unterstützen. Es folgt die Beschreibung der Datenstruktur.

Jedem Knoten v eines Binärbaums T wird eine Menge von Farben C_v und ein Array A_v zugewiesen. Ist v die Wurzel von T, so gilt $A_v = A$ und $C_v = C$. Ausgehend von v werden A_v und C_v nun rekursiv auf die beiden Kinder w und u von V verteilt (siehe Abbildung 1). Die Menge der Farben C_v wird so auf die Mengen C_w und C_u verteilt, dass beide dieselbe Anzahl von Elementen beinhalten[1]. Weiter haben alle Farben in C_w eine niedrigere Priorität als jede Farbe in C_u. Das Array A_w (A_u) enthält alle Einträge von A_v, die eine Farbe aus C_w (C_u) haben. Zusätzlich wird ein Bitvektor B_v der Länge $N_v = |A_v|$ gespeichert. Das i-te Bit in B_v ist gleich Eins genau dann, wenn die Farbe von $A_v[i]$ in C_u enthalten ist. Jeder dieser Bitvektoren wird um eine rank-Datenstruktur erweitert, sodass die Intervalle $[a_w, b_w]$ und $[a_u, b_u]$ in den Kindknoten von v aus dem Intervall $[a_v, b_v]$ wie folgt in konstanter Zeit berechnet werden können:

[1] Es wird angenommen, dass $\sigma = |C|$ eine Zweierpotenz ist.

	1 2 3 4 5 6 7	
T=	b a n a n a $	$C = \{1, 2, 3, 4\}$
A=	7 6 4 2 1 5 3	$p(1) = 0, p(2) = 1,$
D=	1 1 2 3 4 2 1	$p(3) = 1, p(4) = 3$

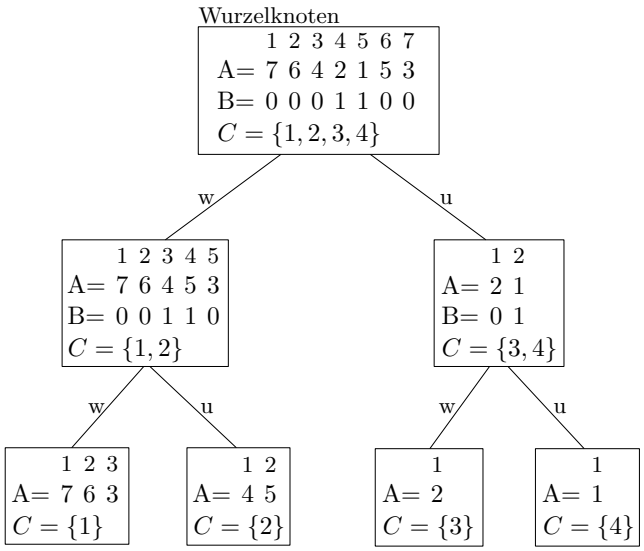


Abbildung 1 Skizze des Wavelet-Trees mit zugrundeliegendem Text T, Suffixarray A, Zuordnung der Farben zu Suffixen D, Menge aller Farben C sowie Prioritäten $p(c)$ der Farben $c \in C$. In jedem Knoten markiert ein Bitvektor B diejenigen Einträge in A, die im rechten Teilbaum dieses Knotens liegen.

$$
\begin{aligned}
a_u &= rank_1(B_v, a_v) \quad \text{erhöhe um 1, wenn } B_v[a_v] = 0 \\
b_u &= rank_1(B_v, b_v) \\
a_w &= rank_0(B_v, a_v) \quad \text{erhöhe um 1, wenn } B_v[a_v] = 1 \\
b_w &= rank_0(B_v, b_v)
\end{aligned}
$$

Der Baum T mit den Bitvektoren B_v ist ein gewöhnlicher Wavelet-Tree, der um die sekundären Datenstrukturen $COUNT_v$ und REP_v erweitert wurde. Seine Höhe ist $\log \sigma \leq \log N$, weshalb jedes Element in $\log \sigma$ sekundären Datenstrukturen gespeichert ist. Jedes Element wird unter Verwendung von $\log N$ Bits in $\log \sigma$ sekundären Datenstrukturen gespeichert, der benötigte Platzverbrauch liegt also in $O(N \log^2 N)$.

Es folgt die Beschreibung des Algorithmus, mit dem die höchstpriorisierten K Farben in einem Intervall $[a, b]$ in unsortierter Reihenfolge ausgegeben werden können. Zu Beginn wird $a_v = a$ und $b_v = b$ gesetzt und bei der Wurzel begonnen.

1. Verwende B_v, um das Intervall $[a_u, b_u]$ im rechten Kind u von v zu finden.
2. Besuche den Knoten u und zähle die Anzahl m_u verschiedener Farben in $A_u[a_u, b_u]$ mit $COUNT_u$.
3. Ist $m_u \geq K$ gebe die höchstpriorisierten K Farben in $A_u[a_u, b_u]$ aus. Das bedeutet: Setze $v = u$ und fahre bei Punkt 1 fort.

4. Ist $m_u < K$ gebe alle Farben in $A_u[a_u, b_u]$ mit REP_u aus.
5. Gebe die höchstpriorisierten $K - m_u$ Farben im linken Kind w von v aus. Verwende hierfür B_v, um das Intervall $[a_w, b_w]$ zu finden, setze $K = K - m_u$ und $v = w$ und fahre bei Punkt 1 fort.

Die Anzahl der besuchten Knoten ist $O(\log N)$, da T höchstens einmal von der Wurzel bis zu einem Blatt traversiert wird. In jedem besuchten Knoten werden höchstens je ein *color counting query* und ein *color reporting query* beantwortet. Nach Lemma 1 kann ein *color counting query* in $O(\log N)$ Zeit beantwortet werden. Das *color reporting query* in einem Knoten v kann in $O(K'_v)$ Zeit beantwortet werden, wobei K'_v der Anzahl der Farben entspricht, die in v ausgegeben werden. Insgesamt können die K höchstpriorisierten Farben in unsortierter Reihenfolge also in $O(\log^2 N + K)$ Zeit ausgegeben werden.

▶ **Lemma 3.** *Es existiert eine Datenstruktur der Größe $O(N \log^2 N)$ Bits, die eine unsortierte Liste der K höchstpriorisierten Farben in einem Anfrageintervall $[a, b]$ in $O(\log^2 N + K)$ Zeit beantworten kann.*

4 Eine Datenstruktur mit $O(N \log N)$ Bits Platzbedarf

In diesem Abschnitt wird die Datenstruktur aus Abschnitt 3 so verändert, dass sie nur noch $O(N \log N)$ Bits Platz benötigt. Für Anfragen wird allerdings $O(N^{1/f} + K)$ Zeit benötigt. Der Ansatz zur Verringerung des Platzbedarfs ist, die Datenstrukturen für das Zählen und Ausgeben von Elementen nur in bestimmten, *wichtigen* Knoten des Baumes zu speichern, sodass jedes Element in T nur in einer konstanten Anzahl von Datenstrukturen gespeichert ist.

Ein Knoten v hat die Tiefe x, wenn er x Vorfahren hat und ist wichtig, wenn er eine Tiefe von $i\lfloor(1/f)\log N\rfloor$ hat für $i = 0, 1, \ldots, f$ und ein konstantes f. Zusätzlich sind alle Blätter wichtige Knoten. Das Array A_v und die Hilfsdatenstrukturen $COUNT_v$ sowie REP_v werden jetzt nur noch in den wichtigen Knoten gespeichert. Für einen wichtigen Knoten v sind $v_1, v_2, \ldots v_t$ die höchsten wichtigen Nachfolger von v, von denen es $t \leq 2^{(1/f)\log N} = N^{1/f}$ viele geben kann. Eine Datenstruktur E_v in jedem wichtigen Knoten erlaubt es, für jedes $i = 1, \ldots, t$ und jedes $j = 1, \ldots, N_v$ die Anzahl Elemente der Farben $c \in C_{v_i}$ an den Positionen $m \leq j$ im Array A_v zu zählen. Hierzu werden für jedes i die Positionen aller Elemente in A_v mit einer Farbe aus C_{v_i} in einer eindimensionalen Range-Counting-Datenstruktur gespeichert.

Jede dieser Datenstrukturen benötigt $O(N_{v_i} \log N_v)$ Bits Platz und beantwortet Anfragen in $O(\log N)$ Zeit. Da $\sum_{i=1}^{t} N_{v_i} = N_v$ ist, benötigt E_v $O(N_v \log N_v)$ Bits Patz. Alle wichtigen Knoten v derselben Tiefe $l = i\lfloor(1/f)\log N\rfloor$ enthalten $O(N)$ Elemente. Daher benötigen die Datenstrukturen REP_v, $COUNT_v$ und E_v $O(N \log N)$ Bits. Da wichtige Knoten nur in konstant vielen Ebenen existieren, liegt der gesamte Platzbedarf in $O(N \log N)$ Bits.

Suchanfragen für ein Intervall $[a, b]$ werden ähnlich wie in Abschnitt 3 beantwortet, jedoch werden nur noch wichtige Knoten besucht. Die Suche beginnt bei der Wurzel, $a_v = a$, $b_v = b$ und $i = t$, wobei t die Anzahl der wichtigen Nachfolger von v ist.

1. Sei $[a_{v_i}, b_{v_i}]$ das Intervall in v_i, das dem Intervall $[a, b]$ in v entspricht. Sind a_v und b_v bekannt, dann können a_{v_i} und b_{v_i} mithilfe von E_v berechnet werden. Anschließend wird die Anzahl der Farben m_{v_i} in $A_{v_i}[a_{v_i}, b_{v_i}]$ und $r_i = \sum_{j=1}^{t} m_{v_i}$ berechnet.
2. Ist $r_i < K$ wird v_i besucht und alle m_{v_i} Farben in $A_{v_i}[a_{v_i}, b_{v_i}]$ ausgegeben. Setze $K = K - r_i$ und fahre mit dem Kind v_{i-1} fort.

3. Ist $r_i \geq K$, setze $K = K - r_{i+1}$,[2] $v = v_i$, $a_v = a_{v_i}$, $b_v = b_{v_i}$ und fahre fort mit Punkt 1.

Die Anzahl besuchter Knoten ist $O(fN^{1/f}) = O(N^{1/f})$. In jedem Knoten wird höchstens ein *color reporting query* und ein *color counting query* beantwortet. In $O(N^{1/f} \log N + K)$ Zeit wird daher eine unsortierte Liste der K höchstpriorisierten Farben ausgegeben. Ist $K < N^{1/f}$ können die Farben in $O(N^{1/f} \log N)$ Zeit nach ihren Prioritäten sortiert werden, für $K \geq N^{1/f}$ wird Radix Sort verwendet. Da für $f' > f$ gilt $O(N^{1/f'} \log N) = O(N^{1/f})$ kann eine Suchanfrage in $O(N^{1/f} + K)$ Zeit beantwortet werden.

▶ **Lemma 4.** *Für ein konstantes f existiert eine Datenstruktur der Größe $O(N \log N)$ Bits, die* top-K color queries *in $O(N^{1/f})$ Zeit beantworten kann.*

5 Eine Datenstruktur mit $O(K)$ Anfragezeit

In diesem Abschnitt wird das Ergebnis von Lemma 4 mit $f = 2$ als Ausgangspunkt verwendet. Obwohl die Ausgabezeit von $O(N^{1/2} + K)$ hoch ist gilt $O(\sqrt{N} + K) = O(K)$, wenn $K = \Omega(\sqrt{N})$. Für ein solches K ist die Datenstruktur aus Lemma 4 optimal. Für $K < \sqrt{N}$ können die Antworten auf bestimmte Suchanfragen explizit mit linearem Platzverbrauch gespeichert werden. Es folgt zunächst eine kurze Beschreibung dieses Ansatzes, anschließend wird die Datenstruktur im Detail dargestellt.

Sei $J = i\lfloor\sqrt{N}\rfloor$ und $L(m, a, b)$ bezeichne die Menge der höchstpriorisierten, in absteigender Reihenfolge sortierten m Farben in $A[a, b]$. Für jedes $i \in J, r = 1, \ldots, \log N$ und für jedes Intervall $[i - 2^r, i]$ und $[i, i + 2^r]$ werden die Listen $L(\sqrt{N}, i - 2^r, i)$ und $L(\sqrt{N}, i, i + 2^r)$ explizit gespeichert. Jedes Intervall $[a, b]$ mit Intervallgrenzen in J kann als Vereinigung der Intervalle $[a, a + 2^x]$ und $[b - 2^x, b]$ geschrieben werden (mit $x = \lfloor\log(b - a)\rfloor$). Die $K \leq \sqrt{N}$ höchstpriorisierten Farben in $A[a, b]$ werden gefunden, indem die ersten K Elemente der Listen $L(\sqrt{N}, a, a + 2^x)$ und $L(\sqrt{N}, b - 2^x, b)$ untersucht werden. Anschließend erfolgt die Ausgabe der K Farben mit den höchsten Prioritäten. Suchanfragen für Intervalle mit Grenzen aus J können somit in $O(K)$ Zeit beantwortet werden.

Um auch Anfragen für beliebige Intervalle beantworten zu können, wird die Datenstuktur aus Lemma 4 für jedes Teilarray $A[i_1, i_2]$ derart gespeichert, dass i_2 in J auf i_1 folgt. Somit enthält jede Datenstruktur $A[i_1, i_2]$ etwa $\sqrt{N}$ Elemente und beantwortet Anfragen in $O(N^{1/4} + K)$ Zeit. Für $K \geq N^{1/4}$ ist dies optimal. Beliebige Anfrageintervalle $[a, b]$ können in drei Teilintervalle $[a, a_1]$, $[a_1, b_1]$ und $[b_1, b]$ zerlegt werden, sodass $a_1 \in J$ und $b_1 \in J$ gilt. Für jedes der Intervalle wird eine sortierte Liste gefunden. Für $[a, a_1]$ und $[b_1, b]$ existiert jeweils eine Liste L und $[a_1, b_1]$ kann wie oben beschrieben in zwei weitere Teilintervalle zerlegt werden. Dann müssen die Listen lediglich durchlaufen und die K Farben mit höchster Priorität ausgegeben werden (4-way merge). Diese Konstruktion wird $O(\log \log N)$ mal durchgeführt. Dadurch ergibt sich für beliebige K die optimale Laufzeit von $O(K)$. Nachfolgend wird die Datenstruktur im Detail beschrieben.

Zunächst folgen einige Definitionen. Sei $\rho(l) = (1/2)^l$ und $\Delta = \log^2 N$. Die Mengen J_l werden definiert als $J_l = \{i\lfloor N^{\rho(l)}\rfloor\Delta \quad | \quad 0 \leq i \leq N^{1-\rho(l)}/\Delta, 1 \leq l \leq h = O(\log \log N)\}$. Für die spätere Platzanalyse der Datenstruktur F wird h so gewählt, dass gilt $N^{\rho(h)} = O(1)$. Für jedes $j \in J_l$ werden die Listen $L(\lceil N^{\rho(l)}\rceil, j - 2^r, j)$ und $L(\lceil N^{\rho(l)}\rceil, j, j + 2^r)$ für $r = 1, 2, \ldots, \log N$ gespeichert.

Für jedes Teilarray $A[j_1 \ldots j_2]$ wird eine Datenstruktur $R(l, j_1, j_2)$ gespeichert, die wie in Abschnitt 4 implementiert wird. Dabei steht j_2 direkt nach j_1 in J_l, $1 \leq l < h$. Für den

[2] $r_{i+1} = 0$ für $i = t$.

Fall $l = h$ wird eine Datenstruktur $R(h, j_1, j_2)$ ebenfalls entsprechend Lemma 4 gespeichert, jedoch wird $f = 1/6$ gesetzt, sodass R *top-K color queries* in $O((j_2 - j_1)^{1/6} + K)$ Zeit beantwortet werden können. Zusätzlich (ebenfalls für $l = h$) wird eine Datenstruktur $F(j_1, j_2)$ gespeichert, die *top-K color queries* in $O(K)$ Zeit unterstützt, wenn gilt $K \leq \log^{1/3} N$. Die Datenstruktur F wird nachfolgend beschrieben. Diese und R arbeiten mit modifizierten Mengen der Farben. Sei $C(j_1, j_2)$ die Menge der Farben, die in $A[j_1, j_2]$ vorkommen. Sie arbeiten auf einer Menge, in der die Farben in $C(j_1, j_2)$ durch den Rang ihrer Priorität ersetzt werden. Auf diese Weise wird sichergestellt, dass alle Farben und Prioritäten in $R(l, j_1, j_2)$ und $F(j_1, j_2)$ im Intervall $[1, j_2 - j_1 + 1]$ liegen.

Die Listen L verwenden $o(N)$ Speicherplatz. Für jedes l gibt es $O(\frac{N}{N^{\rho(l)} \log^2 N})$ Listen und jede Liste benötigt $O(N^{\rho(l)} \log N)$ Bits. Für ein festes l beträgt der Platzbedarf für alle Listen daher $O(N/\log N)$ Bits. Alle Listen R können mit $O(N \log N)$ Bits gespeichert werden: Jede Liste $R(l, j_1, j_2)$ enthält $O(N^{\rho(l)})$ Elemente, deren Farben im Intervall $[1, N^{\rho(l)}]$ liegen. Mit Lemma 4 folgt, dass jede solche Liste R in $O(N^{\rho(l)} \log N^{\rho(l)})$ Bits untergebracht werden kann. Für ein festes l benötigen alle Datenstrukturen $R(l, j_1, j_2)$ $O(N\rho(l) \log N) = \rho(l)O(N \log N) = O(N \log N)$ Bits. Mit der nachfolgenden Beschreibung der Datenstruktur F und deren Größe von ebenfalls $O(N \log N)$ ergibt sich ein Gesamtplatzbedarf von $O(N \log N)$ Bits.

Es folgt nun die Beschreibung von $F(j_1, j_2)$, wobei j_2 in J_h auf j_1 folgt. Da jede Farbe in $[1, j_2 - j_1 + 1] = [1, O(\log^2 N)]$ liegt (vgl. Definition von h und J_h), kann jede Farbe in $O(\log \log N)$ Bits gespeichert werden. Sei $V(j_1, j_2) = v_i$ mit $v_i = j_1 + i\lfloor\sqrt{\log N}\rfloor$ und $v_i \leq j_2$. Für jedes v_i und alle r mit $v_i + 2^r \leq j_2$ wird die Liste $L(\lceil \log_2^{1/3} N \rceil, v_i, v_i + 2^r)$ gespeichert. Für jedes v_i und alle r mit $v_i - 2^r \geq j_1$ wird die Liste $L(\lceil \log_2^{1/3} N \rceil, v_i - 2^r, v_i)$ gespeichert. Da es $O(\log \log N)$ verschiedene Werte von r für jedes v_i gibt, benötigen alle Listen L $o((j_2 - j_1) \log N)$ Bits Platz. Für zwei beliebige aufeinanderfolgende Indizes v_i und v_{i+1} in $V(j_1, j_2)$ werden die Farben aller Elemente in $A[v_i \dots v_{i+1}]$ in einem Maschinenwort $W(v_i, v_{i+1})$ gespeichert. In einer Lookup-Tabelle der Größe $o(N)$ sind alle Wörter W der Datenstruktur abgelegt. *Top-K color queries* auf $A[v_i, v_{i+1}]$ können dann mit bitweisen Operationen in $O(K)$ Zeit beantwortet werden.

Jedes Intervall $[a, b]$ für $j_1 \leq a < b \leq j_2$ kann als Vereinigung der Intervalle $[a, a_f]$, $[a_f, b_e]$ und $[b_e, b]$ dargestellt werden. Dabei gilt $a_f = \lceil (a - j_1)/g \rceil$, $b_e = \lfloor (b - j_1)/g \rfloor$ und $g = \lceil \sqrt{\log N} \rceil$. Für $x = \log(b_e - a_f)$ können die Listen $L(\lceil \log^{1/3} N \rceil, a_f, a_f + 2^x)$ und $L(\lceil \log^{1/3} N \rceil, b_e - 2^x, b_e)$ dazu verwendet werden, die K Farben mit den höchsten Prioritäten in $A[a_f \dots b_e]$ zu finden. In $A[a \dots a_f]$ und $A[b_e \dots b]$ können diese Farben in $O(K)$ Zeit gefunden werden. Abschließend müssen diese drei Listen nur noch gemischt werden, um die Liste der K Farben mit den höchsten Prioritäten in $A[a, b]$ zu erhalten. Daraus ergibt sich folgendes Ergebnis:

▶ **Theorem 5.** *Es existiert eine Datenstruktur der Größe $O(N \log N)$ Bits, die top-K color queries in $O(K)$ Zeit beantworten kann.*

Die beschriebene Datenstruktur kann mit folgendem Algorithmus in $O(N \log N)$ Zeit erstellt werden. Da alle Datenstrukturen R $O(N \log \log N)$ Elemente enthalten, können diese auch in $O(N \log \log N)$ Zeit erstellt werden. Die Datenstrukturen F können in $O(N)$ Zeit erstellt werden.

Die Datenstruktur aus Lemma 3 wird in $O(N \log N)$ Zeit konstruiert. Aus ihr werden alle Listen L erzeugt. Die Anzahl aller Listen L ist $O(N \log N/\Delta)$ und die Anzahl aller Elemente in allen Listen L beträgt $O((N/\Delta) \log N \log \log N)$. Jede Liste $L(N^{\rho(l)}, j_1, j_2)$ kann in $O(\log^2 N + N^{\rho(l)})$ Zeit erzeugt werden. Demnach können alle Listen in $O(N(\log^3 N/\Delta) +$

$(N/\Delta)\log N \log\log N) = O(N(log^3 N/\Delta))$ Zeit generiert werden. Da $\Delta = \log^2 N$ definiert wurde, kann die Datenstruktur aus Theorem 5 in $O(N \log N)$ Zeit konstruiert werden.

Abschließend wird beschrieben, wie die Ausgabe der K höchstpriorisierten Farben im Intervall $[a, b]$ verläuft.

1. Mit der Datenstruktur aus Lemma 1 werden K' Farben aus $[a, b]$ ausgegeben, bis entweder K Farben ausgegeben wurden, oder die Ausgabe endet. Setze $K_q = min(K', K)$.
2. Finde l, sodass gilt $N^{\rho(l)} \geq K_q$. Hierfür werden $h = O(\log\log N)$ Werte in $O(1)$ Zeit durchsucht (unter Verwendung des Ergebnisses aus [5]). Weiter seien $a_1 = \lceil a/\lceil N^{\rho(l)}\rceil\rceil$ und $b_1 = \lceil b/\lceil N^{\rho(l)}\rceil\rceil$. Diese Indizes teilen das Intervall $[a, b]$ in die drei Teilintervalle $[a, a_1]$, $[a_1, b_1]$ und $[b_1, b]$.
3. Bestimme nun die K_q Farben im mittleren Intervall $[a_1, b_1]$ mit den höchsten Prioritäten. Berechne dafür $x = \log(a_1 - b_1)$ und untersuche die Farben in $L(\lceil N^{\rho(l)}\rceil, a_1, a_1 + 2^x)$ und $L(\lceil N^{\rho(l)}\rceil, b_1 - 2^x, b_1)$.
4. In diesem und den nächsten beiden Schritten werden die K_q Farben in den beiden „äußeren“ Intervallen $[a, a_1] := I_1$ und $[b_1, b] := I_2$ mit den höchsten Prioritäten bestimmt. Hierbei werden zwei Fälle unterschieden.
   Ist $l < h$, wird die Datenstruktur $R(l, (a_1 - 1)\lceil N^{\rho(l)}\rceil, a_1\lceil N^{\rho(l)}\rceil)$ verwendet, um K_q Farben in I_1 zu finden. Hierfür wird $O((N^{\rho(l)})^{1/2} + K_q) = O(N^{\rho(l+1)} + K_q)$ Zeit benötigt. Da gilt $K_q > N^{\rho(l+1)}$, werden alle Farben in I_1 in $O(K_q)$ gefunden. Für I_2 funktioniert dieser Vorgang analog.
5. Ist $l = h$ und $K_q \geq \log^{1/3} N$ wird die Datenstruktur $R(h, (a_1 - 1)\lceil N^{\rho(h)}\rceil, a_1\lceil N^{\rho(h)}\rceil)$ verwendet. Da $N^{\rho(h)} = O(\log^2 N)$ ist, benötigt dieser Vorgang nach Lemma 4 $O((\log^2 N)^{1/6} + K_q) = O(K_q)$ Zeit. Für I_2 funktioniert die Suche wieder analog.
6. Ist $l = h$ und $K_q < \log^{1/3} N$ werden die Datenstrukturen $F((a_1 - 1)\lceil N^{\rho(h)}\rceil, a_1\lceil N^{\rho(h)}\rceil)$ für I_1 und $F((b_1)\lceil N^{\rho(h)}\rceil, (b_1 + 1)\lceil N^{\rho(h)}\rceil)$ für I_2 verwendet.
7. Abschließend müssen die K_q Farben mit den höchsten Prioritäten der drei Intervalle in $O(K)$ Zeit gemischt werden, sodass die K_q Farben mit den global höchsten Prioritäten für das Intervall $[a, b]$ vorliegen.

6 Eine Datenstruktur mit $O(N \log \sigma)$ Bits Platzbedarf

Das Ergebnis aus Theorem lässt sich für den Fall $\sigma = o(N)$ noch weiter zu einer Datenstruktur der Größe $O(N \log \sigma)$ Bits verbessern. Diesem Abschnitt liegt die Annahme zugrunde, dass jede Farbe als ganze Zahl zwischen 1 und σ dargestellt wird. Dies kann erreicht werden, indem jede Farbe durch den Rang ihrer Priorität ersetzt wird. Der praktizierte Ansatz teilt die Arrays in Blöcke, sodass die Datenstruktur für jeden Block $O(\log \sigma)$ Bits pro Element benötigt. Für die Suche in mehreren Blöcken wird außerdem die globale Datenstruktur benötigt. Diese enthält $O(N/\log N)$ Elemente und kann deshalb in $O(N)$ Bits gespeichert werden. Nachfolgend werden die Fälle $\sigma^2 \geq \log N$ und $\sigma^2 < \log N$ unterschieden.

$\sigma^2 \geq \log N$: Das Array A wird in die Blöcke $L_i = A[(i-1)\ell + 1 \ldots i\ell]$ aufgeteilt, sodass jedes L_i $\ell = \sigma^3$ Elemente enthält. Für jeden Block wird die Datenstruktur aus Theorem 5 in $O(\ell \log \sigma)$ Bits gespeichert. Alle diese Datenstrukturen belegen dann $O(N \log \sigma)$ Bits. Weiter wird eine Datenstruktur D^T für ein Hilfsarray A^T gespeichert, das für jeden Block σ Einträge enthält. Die Einträge $A^T[(i-1)\sigma + 1 \ldots i\sigma]$ enthalten Informationen über die Farben, die im Block L_i vorkommen. Taucht eine Farbe c in L_i auf, dann hat $A^T[(i-1)\sigma + c]$ die Farbe c. Liegt c nicht in L_i, dann hat $A^T[(i-1)\sigma + c]$ eine Dummy-Farbe c_D, deren Priorität kleiner als die aller anderen Farben ist. Für A^T wird ebenfalls eine Datenstruktur aus Theorem 5 gespeichert.

Gehören a und b zu demselben Block kann eine Anfrage $Q = [a, b]$ mit der Datenstruktur dieses Blockes beantwortet werden. Ist das nicht der Fall, so kann $[a, b]$ in $[a, a']$, $[a' + 1, b']$ und $[b' + 1, b]$ zerteilt werden. Dabei gilt $a' = \lceil a/\ell \rceil \ell$ und $b' = \lfloor b/\ell \rfloor \ell$. Die K Farben mit höchster Priorität in $A[a, a']$ und $A[b' + 1, b]$ werden mit den Datenstrukturen der Blöcke bestimmt, diejenigen Farben in $A[a' + 1, b']$ mit D^T. Die Farbe c liegt in diesem Intervall genau dann wenn sie in $A^T[\lceil a/\ell \rceil \sigma + 1 \dots \lfloor b/\ell \rfloor \sigma]$ enthalten ist. Die Dummy-Farbe c_D muss unter Umständen entfernt werden. Die drei resultierenden Listen sind nach der Priorität der Farben sortiert und müssen nur noch gemischt werden, um die K Farben mit den höchsten Prioritäten in $O(K)$ Zeit zu erhalten.

$\sigma^2 < \log N$**:** Es wird dieselbe Datenstruktur wie für den Fall $\sigma^2 \geq \log N$ verwendet. Der einzige Unterschied besteht darin, dass die Blöcke jetzt aus $\sigma^2 \lfloor N \rfloor$ Elementen bestehen. Jeder Block L_i wird noch einmal in $O(\sigma^2 \lfloor \log \sigma \rfloor)$ Stücke P_j unterteilt. Jedes Stück besteht aus $\lceil \log_\sigma N \rceil$ Elementen. Das Array L_i^T beinhaltet σ Einträge für jedes Stück. Wenn das j-te Stück $L_i[j\lceil \log_\sigma N \rceil + 1 \dots (j+1)\lceil \log_\sigma N \rceil]$ ein Element der Farbe c enthält, dann ist die Farbe von $L_i^T[j\sigma + c]$ c, ansonsten die Dummy-Farbe c_D.

Jedes Stück besteht aus $O(\log_\sigma N)$ Elementen und passt in eine konstante Menge von Wörtern der Größe $\log N$ Bits, weil die Farbe jedes Elementes in $O(\log \sigma)$ Bits gespeichert werden kann. Anfragen nach den Farben mit den höchsten Prioritäten können für jedes Stück mit einer vorberechneten Tabelle T der Größe $o(N)$ beantwortet werden. Diese Tabelle enthält für alle möglichen Folgen seq_l von $s := \lfloor \log_\sigma N/2 \rfloor$ Farben und für jedes $1 \leq x_1 \leq x_2 \leq s$ alle Farben, die zwischen x_1 und x_2 auftauchen. Die Farben sind nach absteigender Priorität sortiert. Ein Stück kann in eine konstante Anzahl von Folgen seq_f von s Farben zerlegt werden. Für jede dieser Sequenzen kann der entsprechende Eintrag in T nachgeschaut werden. Somit können die K Farben mit den höchsten Prioritäten zwischen zwei beliebigen Positionen in seq_f in $O(1)$ Zeit gefunden werden.

Für den Fall $\sigma^2 \geq \log N$ wurden top-K Queries mit je einer Anfrage an die Blöcke L_i und L_j und einer Anfrage an A^T beantwortet. Analog hierzu wird nun eine Anfrage an einen Block L_i mit je einer Anfrage an die Stücke P_k und P_l und einer Anfrage an L_i^T beantwortet, benötigt also ebenfalls $O(K)$ Zeit. Hieraus ergibt sich folgendes Ergebnis:

▶ **Theorem 6.** *Sei σ die Anzahl der existierenden Farben. Dann existiert eine Datenstruktur der Größe $O(N \log \sigma)$ Bits, die top-K color queries in $O(K)$ Zeit beantworten kann.*

Die Konstruktion einer solchen Datenstruktur liegt in $O(N \log \sigma)$. Dies kann genauso gezeigt werden wie für die Datenstruktur aus Theorem 5.

7 Zusammenfassung

Das Ergebnis aus Theorem 5 kann auf das externe Speicher-Modell übertragen werden. Details hierzu und eine Beschreibung zur Behandlung von Online-Queries ist in den Abschnitten 7 und 8 im originalen Artikel zu finden.

In Abschnitt 6 wurde eine Datenstruktur mit optimaler Laufzeit und optimalem Platzbedarf im Worst-Case beschrieben. Eine komprimierte Datenstruktur, vorgestellt von Hon et al. [2] benötigt weniger Speicherplatz, die Laufzeit für Anfragen liegt aber in $O(\log^{3+\epsilon} N)$ Zeit. Es ist nicht bekannt, ob eine derartige Datenstruktur mit schnellerer Laufzeit existiert.

Literatur

1 M. Karpinski, Y. Nektrich, „Top-K Color Queries for Document Retrieval“, in *SIAM*, 20xx, pp. 401–411.

2 W.-K. Hon, R. Shah, J. S. Vitter, „Space-Efficient Framework for Top-k String Retrieval Problems“, in *Proc. 50th IEEE FOCS 2009*, pp. 713–722.

3 J. Fischer, V. Mäkinen, and N. Välimäki, „Space Efficient String Mining under Frequency Constraints“, in *Proceedings of International Conference on Data Mining*, 2008, pp. 193–202.

4 P. Gupta, R. Janardan, M. H. M. Smid, „Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization“, in *J. Algorithms 19(2)*, 1995, pp. 282–317.

5 M. L. Fredman, D. E. Willard, „Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths“, in *J. Comput. Syst. Sci. 48(3)*, 1994, pp. 533–551.

Labeling Scheme for Small Distances in Trees

Philipp Glaser[1]

1 **Karlsruhe Institut of Technology**
philipp.glaser@usta.de

Abstract

In this paper we present a way to efficiently label a tree so that small distances ($< k$) between two vertices (or if they have a common ancestor that has at most distance k) can be calculated from these labels alone without knowing the tree explicitly. The size of these labels is bounded by $\log n + O(k^2(\log\log n + \log k))$. The Preprocessing can be done in $O(n)$ and the query time is in $O(k^2)$. This data structure was found by Alstrup, Bille and Rauhe [1].

1998 ACM Subject Classification E.1 Graphs and networks

Keywords and phrases labeling, trees, small distances

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Motivation

Due to growing graph sizes a need for a fast way to calculate distances in a tree just knowing the labels of the vertices arises. A labeling scheme the advantage, that we don't need to traverse the graph to find the result we are interested in.

A labeling scheme to calculate any distance between two vertices takes $\Theta(log^2(n))$[1] space. If we just want to know the distance if the vertices are close[2], smaller labels suffices. The labeling scheme proposed only takes $log(n) + O(k^2(log\,log(n) + log(k)))$ bits per label, i.e. for any given k the labels only take $log(n) + O(log\,log(n)))$ space.

2 Preliminaries

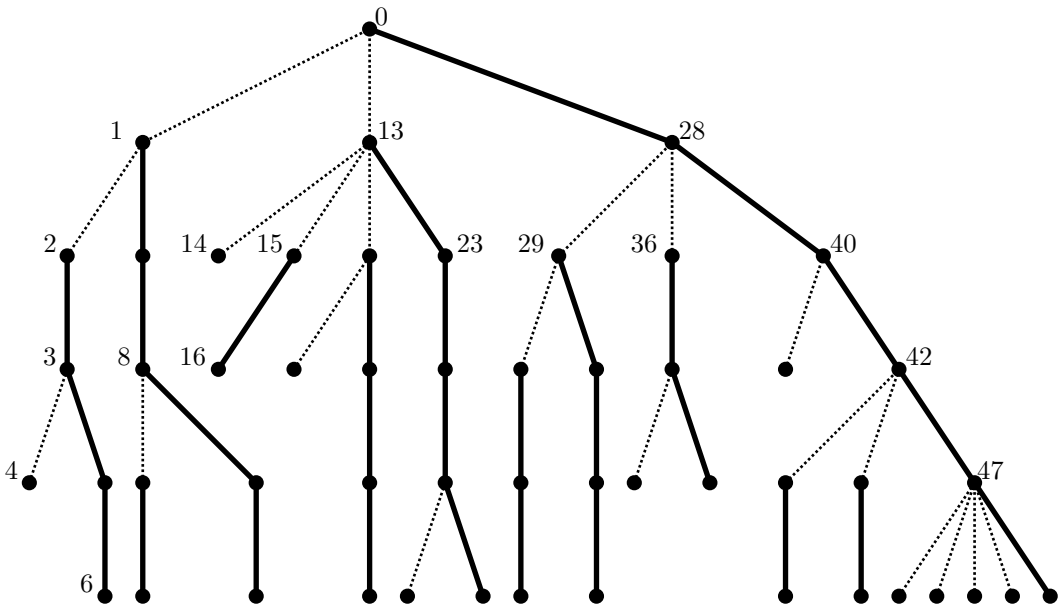
For a graph G we denote the set of vertices by $V(G)$ and the set of edges by $E(G)$. The number of vertices in a Graph is called order of a graph $v(G)$ and the order of the whole graph is denoted n. Let T be a rooted Tree with n vertices and $T(v)$ is the subtree, which is rooted at the vertex v. The number of edges on a path between two vertices $v, w \in T$ is called distance $dist(v, w)$. The distance between a vertex v and the root is called depth $d(v)$. v is called ancestor of w if $w \in T(v)$. It is called proper if $w \neq v$. w is called (proper) descendant of v. A vertex z is called common ancestor of v and w if it is ancestor of v and w. The common ancestor with the largest depth is called nearest common ancestor $nca(v, w)$. The ith level ancestor $A(v, i)$ is the ancestor at depth $d(v) - i$. The vertex $A(v, 1)$ is called parent of v and v is the child of $A(v, 1)$. Vertices with the same parent are called siblings. Vertices without children are called leaves, the other vertices are called internal vertices. Two vertices v, w are k_1, k_2-related if their nca is at distance k_1 from v and at distance k_2 from w.

[1] *log* refers to the binary logarithm in this paper
[2] Their nearest common ancestor is closer than k

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–5

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Figure 1 Example Tree, that is used as reference in the further description



A binary query is a mapping $f : V(G) \times V(G) \to X$ and a labeling scheme defines label assignments (maps vertices to labels) and decoding instructions for such queries using the assigned labels.

3 The Labeling Scheme proposed by Alstrup et. al.

3.1 Calculating distances using heavy path decomposition

Without loss of generality at first we rearrange our tree in a way that for any internal vertex v the children are ordered with the child that is the root of the largest subtree[3] as the rightmost child. We also number all vertices v in our tree by the sequence they are visited in a depth-first search started in the root. We call this the preorder number $pre(v)$. The root vertex has the number 0 and the rightmost leaf has the number $n - 1$.

The child of each vertex with the most descendants (compared to its siblings) is called *heavy* and all other children are called *light*. The root is also light. The edge between a heavy vertex and its parent is called *heavy edge*. All other edges are *light edges*. In figure 1 an example tree is given with heavy edges marked and some preorder numbers. We denote the heavy child of vertex v with $heavy(v)$. The *light subtree* $L(v)$ of an vertex v is defined as the sub tree rooted in v with the subtree rooted in the heavy child ($T(heavy(v))$) removed. If the vertex does not have any children or only one child (which is the heavy child) $L(v)$ contains only v.

The $lightdepth(v)$ of a vertex v is the number of light edges on the path from v to the root and $lightsize(v)$ is the order of $L(v)$. The nearest light ancestor (v itself if it is light) is called $apex(v)$. So the apex is the vertex on the same heavy path as v, that is closest to the root.

[3] The largest subtree is the subtree that contains the most vertices

▶ **Lemma 1.** *For any tree T with n vertices $lightdepth(v) \leq \log n + O(1)$ for any vertex $v \in T$.*

A vertex v is called significant ancestor of w if and only if $w \in L(v)$ and $pre(w) \in [pre(v), pre(v) + lightsize(v) - 1]$ is true for all vertices in $L(v)$. So the next significant ancestor is the vertex in a heavy path one light level above, which is connected to the apex of current vertex. So by removing the light edges the graph can be decomposed into heavy paths. Given the significant ancestors of two vertices and the distances to the apexes of all significant ancestors we can calculate the distances between these two vertices.

In our example graph we can calculate the distance between 23 and 29 by following the significant ancestors up to a common heavy path, which can be recognized by its apex. 23 has the apex 13 (distance 1) and 0 is the second significant ancestor (apex 0 with distance 0). 29 has the apex 29 (distance 0) and the second significant ancestor is 28, which has the apex 0 (distance 1). So the distance can be calculated by the distance of the ancestors on the lowest common heavy path (1, simply the difference of the two corresponding distances to the apex of this path) plus the distances of each vertex to these ancestors (dist(23,0)=1+1 and dist(29,28)=0+1).

3.2 Encoding the required information efficiently

We also define the *significant preorder number* $spre(v)$ as the number greater than or equal the preorder number where the $f(v) = \lfloor log(lightsize(v) \rfloor$ least significant bits are zero. So in our example spre(23) is 23 (lightsize is 1) and spre(29) is 32 (lightsize is 4).

▶ **Lemma 2.** *Regarding this $spre(v)$ the following holds:*[4]

1. $spre(v) \in [pre(v), pre(v) + lightsize(v) - 1]$
2. $v = w$ *iff* $lightdepth(v) = lightdepth(w)$ *and* $spre(v) = spre(w)$
3. $lightdepth(v) = lightdepth(w) \Rightarrow pre(v) < pre(w)$ *iff* $spre(v) < spre(w)$

So we can identify each vertex v by $lightdepth(v)$ and $spre(v)$. On the other hand it is also possible to encode $spre(w)$ of the significant ancestor w of v efficiently using only log log n bits for any light vertex, because in $L(w)$ there can be only two preorder numbers with the $f(w)$ least significant bits set to zero. So if we set the $f(w)$ least significant bits of $pre(v)$ to zero we get either $spre(w)$ or $spre(w) \pm 2^{f(w)}$. We only need $\lceil log \log n \rceil$ bits to represent $f(w)$ since $f(w)$ is bounded by $log\, n$ and two bits to choose one of the three possibilities.

For the heavy paths we also need *diff_heavy* (v, m) as the difference of the $spre(v)$ to the spre of the mth descendant on the heavy path. On the other hand we define *diff_parent* (v, m) as the difference of the $spre(v)$ to the spre of the mth ancestor on the heavy path. Both differences are set to $2n$ if there is no mth vertex on the path in the particular direction.

We also define $index(v)$ as the number of vertices with the same lightdepth and a lower spre.

▶ **Lemma 3.** *For a heavy vertex v and an internal vertex w, w and v are on the same heavy path and w is an ancestor of v if and only if all the following conditions are true:*

1. $spre(w) < spre(v)$,
2. $lightdepth(v) = lightdepth(w)$,
3. $\lfloor log(spre(v) - spre(w)) \rfloor = \lfloor log\ diff\_parent(v, m) \rfloor = \lfloor log\ diff\_heavy(w, m) \rfloor$ *and*
4. $index(v) \bmod m = index(w) \bmod m$.

[4] The proofs are not reproduced in the paper and can be found in Alstrup et. al. [1].

pre(v)	lightdepth(v)	spre(w)	dist(v,w)	apex bit	dist(w,apex(w))	$\lfloor log$ diff_heavy$(w,m)\rfloor$	$\lfloor log$ diff_parent$(w,m)\rfloor$	index(w) mod m
23	1	0/0	0	1	1	3	0	0
						6	1	0
		4/-	2	1	0	6	4	0
						6	5	0
29	1	3/+	0	1	0	6	0	0
						6	1	0
		3/+	1	1	1	4	3	0
						6	3	1
Size								
$log\, n$	$log\, log\, n$	$log\, log\, n$	1	$log\, log\, n$	$log\, log\, n$	$log\, log\, n$	$log\, log\, n$	$log\, k$
$*1$	$*1$	$*k$	$*k$	$*k$	$*k$	$*k^2$	$*k^2$	$*k^2$

Table 1 Values for the vertices 23 and 29 in the example Graph. $Spre(w)$ denotes the significant ancestor w of v. The number i in this column is number of least significant bits that are set to zero and $+/-$ denotes if 2^i is added or subtracted.

Putting it all together we store for each vertex:

1. $pre(v)$, which takes $\lfloor log\, n \rfloor$ bits
2. $lightdepth(v)$, which takes $\lfloor log\, log\, n \rfloor$ bits
3. a table which contains the following for each significant ancestor w with the distance of at most k from v
   a. $spre(w)$, which takes $log\, log(n) + 2$ bits
   b. the distance from v, which takes $log\, log\, n$ bits
   c. an apex bit, which is set if apex(w) is of distance at most k from w
   d. $dist(w, apex(w)$, if the apex bit is set, otherwise undefined
   e. we also store for all $1 < m \leq k$:
      i. $\lfloor log$ diff_heavy$(w,m)\rfloor$, which takes $log\, log\, n$ bits
      ii. $\lfloor log$ diff_parent$(w,m)\rfloor$, which also takes $log\, log\, n$ bits
      iii. index(w) mod m, which takes $log\, k$ bits

Altogether we therefore need $log\, n + O(k^2(log\, log\, n + log\, k))$. The calculation of the labels can be done in $O(k)$ time per vertex after $O(n)$ time preprocessing. Hence the labeling scheme can be computed in $O(nk)$ time.

3.3 Decoding the information to get the distance

Let v' be the significant ancestor of v such that lightdepth(v') = lightdepth($A(v, k_1)$) and if $v' \neq v$ let v'' be the significant ancestor of light depth lightdepth($A(v, k_1)$) + 1. For the vertex w w' and w'' are defined accordingly using k_2 instead of k_1.

▶ **Lemma 4.** *For $k_1, k_2 > 0$ two vertices are (k_1, k_2)-related if exactly one of the following conditions is true:*

1. *$v' = w'$, v'' and w'' are on different heavy paths, $dist(v, v') = k_1$ and $dist(w, w') = k_2$*
2. *v' and w' are on the same heavy path, v' is a proper ancestor of w', $dist(w', v') = k_2 - dist(w, w')$ (i.e. $dist(w, v') = k_2$) and $dist(v, v') = k_1$*
3. *v' and w' are on the same heavy path, w' is a proper ancestor of v', $dist(w', v') = k_1 - dist(v, v')$ (i.e. $dist(v, w') = k_1$) and $dist(w, w') = k_2$*

So we can find out if v and w are (k_1, k_2)-related as follows: If $k_1, k_2 = 0$ the labels are equal, because each vertex has a unique label. If k_1 or k_2 are greater than 0 we will test the above conditions. Let us assume the values of v', w', v'' and w'' are available (closer than k to their respective vertex).

Using Lemma 2 we can test if v' and w' are equal. If they are the distances from v and w are stored in the ancestor table of the vertices. The first three conditions of Lemma 4 (ii) and (iii) can be tested using Lemma 3. So the only thing left to test in Lemma 4 (i) is if v'' and w'' are on different heavy paths. Since the distances $dist(v'', apex(v''))$ and $dist(w'', apex(w''))$ are smaller than k they are stored in the ancestor table (the distance from v'' to v' is the $dist(v'', apex(v'')) + 1$). If they are on the same heavy path their distance is $|dist(v'', apex(v'')) - dist(w'', apex(w''))|$, so we can use Lemma 3 to test if they are on the same path and if they are we already have the correct distance.

In our example we test for a $2, 2$-relationship we can see from the label of vertex 23 that on the same heavy path the vertex has only one left/upper neighbor, so $A(23, 2)$ has to be the second significant ancestor ($23'$), which has the spre 0, which can be calculated by setting the 4 rightmost bits to 0 and subtract 2^4. For the vertex 29 also know from the label that we need the second significant ancestor ($29'$) which has the spre 32, which can be calculated by setting the 3 rightmost bits to 0 and adding 2^3. We now need to find out if they are on the same path and calculating the difference between these two ancestors $|dist(23', apex(23')) - dist(29', apex(29'))| = |0 - 1| = 1$. Using Lemma 3 we know $spre(23') < spre(29')$, lightdepth($23'$)=lightdepth($29'$), $\lfloor log(spre(23') - spre(29')) \rfloor = \lfloor log\,\text{diff\_parent}(29', 1) \rfloor = \lfloor log\,\text{diff\_heavy}(23', 1) \rfloor = 4$ and index($23'$) mod 1 = index($29'$) mod 1= 0 and $23'$ and $29'$ therefore are on the same heavy path.

The query can be done in $O(1)$ time. Repeating this test k^2-times for all possible combinations still takes constant time for a constant k.

4 Conclusion

The presented data structure allows a efficient way to store labels that allows k_1, k_2-relationship queries. If k is fix the labels only have a size of $log\, n + O(log\, log\, n)$. The labels can also be computed rather easy, so they can be used in very large trees.

On the other hand the labels are only small for very small k as one can easily see. For the distance $k = log\, n$ the size is already $log\, n + O(log^2\, n(log\, log\, n + log\, log\, n)) = O(log^2\, n\, log\, log\, n$, which is worse than the space for labels which allow the calculation of all distances ($\Theta(log^2\, n)$). Nevertheless there is still potential for such a data structure, because of huge networks or xml-data trees are becoming more common, where we might be only interested in the local neighborhood of a vertex.

References

1 Stephen Alstrup, Philip Bill and Theis Rauhe. *Labeling Schemes for Small Distances in Trees.* Society for Industrial and Applied Mathematics, SIAM Journal on Discrete Mathematics, Vol. 19, Issue 2, 448-462, 2005

Bloom-Filter: Eine probabilistische Datenstruktur zur effizienten Repräsentation von Mengen*

Markus Jung[1]

1 **Karlsruher Instit für Technologie**
markus.jung@stud.uni-karlsruhe.de

Zusammenfassung

Das Bloom-Filter ist eine Datenstruktur zur Repräsentation von Mengen. Gegenüber anderen Datenstrukturen wie den konzeptuell ähnlichen Hashtabellen oder Bitvektoren hat das Bloom-Filter einen geringeren Speicherbedarf. Die effiziente Nutzung des Speicherplatzes wird durch die probabilistische Arbeitsweise der Datenstruktur möglich. Bloom-Filter arbeiten fehlerbehaftet, Elemente können mit einer gewissen Wahrscheinlichkeit fälschlicherweise der dargestellten Menge zugeordnet werden. Die Fehlerwahrscheinlichkeit kann entsprechend anwendungsspezifischer Kriterien gewählt werden und ist bei gleichbleibender Anzahl Bits je Element unabhängig von der Größe der repräsentierten Menge. Bloom-Filter ermöglichen damit einen flexiblen Kompromiss zwischen Fehlerrate und Speicherbedarf.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, E.2 Data Storage Representations

Keywords and phrases bloom filters, hashing, randomized algorithms, data structures, sets

Digital Object Identifier 10.4230/LIPIcs.9999.9999.9

1 Einleitung

Das bereits 1970 von Burton Bloom vorgestellte Bloom-Filter [1] ist eine Datenstruktur zur Repräsentation von Mengen. Sie unterstützt typische Mengenoperationen: Das Hinzufügen von Elementen, die Prüfung der Zugehörigkeit von Elementen, die Bildung des Schnitts und der Vereinigung. Die Elemente einer Menge werden über Hashwerte identifiziert, wobei keine Behandlung von Hashkollisionen erfolgt. Daraus resultiert einerseits das Risiko von Fehlentscheidungen, andererseits aber auch der geringere Speicherbedarf im Vergleich zu anderen Mengenrepräsentationen wie Hashtabellen oder Hashlisten. Die bei Bloom-Filtern möglichen Fehler sind *false positives*, dabei werden einer Menge Elemente zugeordnet, die dieser nicht angehören.

Der Speicherbedarf eines Bloom-Filters ist während seiner Nutzung konstant. Es handelt sich um eine statische Datenstruktur, deren Größe sich aus der geforderten Fehlerrate und der Elementanzahl der darzustellenden Menge ergibt. Keinen Einfluss auf den Speicherbedarf hat das Universum aus dem die Elemente der Menge stammen.

Bloom-Filter eignen sich vor allem für Anwendungsfälle bei denen die auftretenden Fehler zugunsten eines geringeren Speicherbedarfs toleriert oder kompensiert werden können. Ein typisches Anwendungsbeispiel ist die Reduktion von Kommunikationsvolumen bei verteilten Systemen, etwa im Bereich von Netzwerkprotokollen und -anwendungen wie Datenbanken [3].

* Diese Ausarbeitung ist im wesentlichen eine Zusammenfassung der Publikation von Mitzenmacher und Broder [3] hinsichtlich des „klassischen“ Bloom-Filters.

1st Symposium on Breakthroughs in Advanced Data Structures (BADS'13).
Editor: J. Fischer; pp. 25–33
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Hier können Bloom-Filter zum Abgleich von Datenbeständen eingesetzt werden indem die beteiligten Kommunikationspartner diese Filter anstelle vollständiger Datensätze austauschen. Anhand der ausgetauschten Bloom-Filter ist dann eine Vorauswahl der zu versendenden Daten möglich, für den Anwendungszweck uninteressante Datensätze müssen nicht mehr übertragen werden. Potentiell auftretende *false positives* resultieren in diesem Szenario lediglich in der Übertragung von eigentlich nicht benötigten Daten und bleiben ohne weitere Folgen.

Gliederung Das folgende Kapitel setzt sich mit dem Bloom-Filter in seiner ursprünglichen Version von 1970 auseinander und behandelt sowohl die Funktionsweise als auch die zugrunde liegende Mathematik. Im Anschluss werden zwei erweiterte beziehungsweise spezialisierte Formen des Bloom-Filters vorgestellt, den Abschluss bildet ein kurzes Fazit.

2 Das „klassische“ Bloom-Filter

Konzeptuell ähnelt das Bloom-Filter einer Hashtabelle. Anstelle von Tabelleneinträgen indizieren die Hashwerte aber Bits in einem Bitvektor der Datenstruktur, außerdem werden mehrere Hashfunktionen eingesetzt.

Ein Bitvektor V ist ein Tupel aus m Bits $(b_m, b_{m-1}, \ldots, b_1)$, $|V| := m$. Im weiteren Verlauf werden die folgenden Operationen verwendet:

▶ **Definition 1** (Bitweiser Vergleich). $V_a \stackrel{?}{=} V_b$ und $V_a \neq V_b$

▶ **Definition 2** (Binäres Oder). $V_a \oplus V_b$ (als Aggregation über mehrere Vektoren: $\bigoplus$)

▶ **Definition 3** (Binäres Und). $V_a \odot V_b$ (als Aggregation über mehrere Vektoren: $\bigodot$)

▶ **Definition 4** (Binäre Negation). $\neg V$

Im Folgenden wird der Bitvektor V Synonym zum zugehörigen Bloom-Filter verwendet.

Die Elemente einer Menge S werden durch k voneinander unabhängige Hashfunktionen $h_1, \ldots, h_k$ gleichförmig auf den Wertebereich $\{1, \ldots, m\}$ abgebildet. Die Hashwerte indizieren dabei die in V zu setzenden Bits, jedes Element $x \in S$ wird so auf eine Bitmaske reduziert:

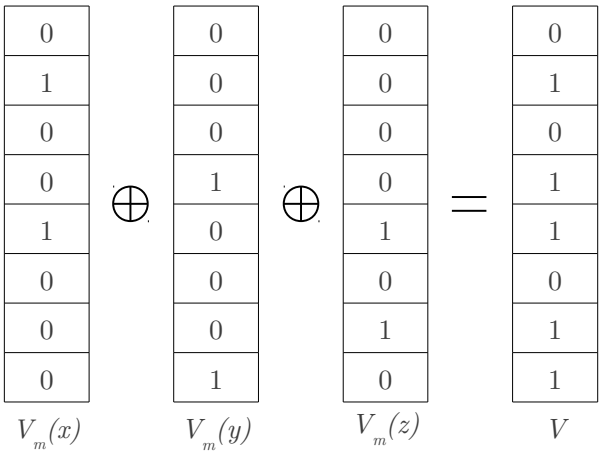
$$V_m(x) := \bigoplus_{i=1}^{k} (b_m = 0, \ldots, b_{h_k(x)} = 1, \ldots b_1 = 0)$$

Der Wert des Bitvektors berechnet sich dann aus dem binären Oder der Bitmasken aller Elemente von S:

$$V = \bigoplus_{x \in S} V_m(x)$$

Damit enthält V die gesetzten Bits aller eingefügten Elemente, für jedes eingefügte Element kann die Zugehörigkeit zu der Menge also zweifelsfrei festgestellt werden, wie im Beispiel in Abbildung 1 zu erkennen ist. Es treten keine *false negatives* auf, $P(x \notin V | x \in S) = 0$.

Die Größe des Bitvektors wird einmalig im Zuge der Initialisierung der Datenstruktur gewählt ist danach unveränderlich, gleiches gilt für die Hashfunktionen des Bloom-Filters.



e	$h_1(e)$	$h_2(e)$
x	2	5
y	8	4
z	5	7

Abbildung 1 Konstruktion eines Bloom-Filters

2.1 Funktionalität

Das Bloom-Filter wird als leere Menge initialisiert, also mit $V = (b_m = 0, \ldots, b_1 = 0)$. Danach können Elemente der Menge hinzugefügt werden, in dem alle durch die Hashwerte indizierten Bits im Bitvektor gesetzt werden,

$$V = V \cup x \quad \Leftrightarrow \quad V = V \oplus V_m(x) \; .$$

Abbildung 1 zeigt die Konstruktion eines Bloom-Filters aus drei Elementen, wobei zwei Elemente einen gemeinsamen Hashwert aufweisen ($h_2(x) = h_1(y)$). Beim Versuch eines dieser Elemente aus dem Bloom-Filter zu entfernen wäre zwangsläufig auch das andere Element betroffen. Aus diesem Grund ist es nicht möglich, ein einmal eingefügtes Element aus dem Bloom-Filter wieder zu entfernen. Das *Counting Bloom Filter* (Kapitel 3.1) versucht diese Einschränkung zumindest teilweise zu beheben.

Aus dem gleichen Grund kann auch die Komplementärmenge für $S \subseteq U$ nicht ermittelt werden. Diese Operation entspräche dem Entfernen aller Elemente in S aus einem Bloom-Filter der Universumsmenge U:

$$\neg V = \neg \bigoplus_{x \in S} V_m(x) = (b_m = 1, \ldots, b_1 = 1) \odot \bigodot_{x \in S} \neg V_m(x)$$

Die Zugehörigkeit zur Menge kann durch ein Bloom-Filter nicht eindeutig bestimmt werden. Ein Element w mit $h_1(w) = 7$ und $h_2(w) = 2$ würde vom Bloom-Filter in Abbildung 1 als zugehörig akzeptiert werden, obwohl es nie in diesen eingefügt wurde. Um das Risiko eines solchen *false positive* zu reduzieren, verwendet das Bloom-Filter mehrere Hashfunktionen. Würde, wie bei einfachen Hashtabellen, nur eine Hashfunktion verwendet, hätte jede Hashkollision eine Fehlzuordnung als Folge. Ein Element y ist aber sicher nicht Teil der Menge, wenn nicht jedes der von ihm indizierten Bits im Bitvektor gesetzt ist. Es gilt:

$$y \notin V \quad \Leftrightarrow \quad V \odot V_m(y) \neq V_m(y)$$

Den Schnitt von Mengen können Bloom-Filter approximieren, die Vereinigung von Mengen kann mit der Datenstruktur exakt ermittelt werden. Die binären Mengenoperationen erfordern dabei die Verwendung identischer Hashfunktionen, gleiches gilt für die Größe der verwendeten Bitvektoren.

Die Schnittmenge zweier Bloom-Filter kann durch das binäre Und der beiden Bitvektoren angenähert werden. Die Schnittmenge schließt sicher alle Elemente aus die nur von einem der beiden Bloom-Filter als zugehörig identifiziert werden, es können aber weitere Bits gesetzt bleiben die zu keinem der enthaltenen Elemente zuzuordnen sind.

Die Vereinigung kann durch ein binäres Oder exakt berechnet werden, das resultierende Bloom-Filter verhält sich dann so, als ob alle Elemente einzeln zu der Menge hinzugefügt worden wären:

$$V_1 \oplus V_2 = \bigoplus_{x \in S_1} V_m(x) \oplus \bigoplus_{x \in S_2} V_m(x) = \bigoplus_{x \in S_1 \cup S_2} V_m(x)$$

Jede der in diesem Abschnitt erläuterten Operationen weist eine konstante Laufzeit auf, ein Bloom-Filter für eine Menge S mit $n := |S|$ kann in $\mathcal{O}(n)$ aufgebaut werden. Zeitaufwändige Berechnungen wie die Auswertung der Hashfunktionen können parallelisiert werden, gleiches gilt für das Einfügen größerer Mengen.

2.2 Mathematische Analyse

Wie im vorherigen Kapitel erläutert wurde, können bei Bloom-Filtern nur *false positives* auftreten, $P(x \in V | x \notin S) > 0$. Die Wahrscheinlichkeit einer solchen Fehlentscheidung hängt von der Anzahl der ungesetzten Bits im Bloom-Filter ab. Ein Bit eines Bloom-Filters der Größe m wird durch eine gleichverteilte Hashfunktion mit der Wahrscheinlichkeit

$$1 - \frac{1}{m}$$

nicht gesetzt. Werden $n := |S|$ Elemente unter Verwendung von k Hashfunktionen in das Bloom-Filter eingefügt, beträgt die Wahrscheinlichkeit, dass ein bestimmtes Bit nicht gesetzt ist,

$$p' := \left(1 - \frac{1}{m}\right)^{kn}$$

wenn die Hashfunktionen, wie eingangs gefordert, voneinander unabhängig und gleichverteilt sind. Die Größe des Universums aus dem die Elemente von S stammen, hat auf diese Wahrscheinlichkeit keinen Einfluss. Basierend auf einer Definition der Exponentialfunktion

$$\mathrm{e}^x = \lim_{n \to \infty} \left(1 + x \cdot \frac{1}{i}\right)^i$$

folgt nach [7] mit

$$p' = \left(1 - \frac{1}{m}\right)^{kn} = \left(1 - \frac{kn}{m} \cdot \frac{1}{kn}\right)^{kn} = \left(1 - \frac{kn}{m} \cdot \frac{1}{i}\right)^i$$

eine einfachere Formel zur Approximation von p' für große Mengen S:

$$p := \mathrm{e}^{-kn/m}$$

Die Wahrscheinlichkeit eines Bits, nach dem Einfügen von S gesetzt zu sein, ist also

$$(1 - p') \approx (1 - p) \; .$$

Es wäre also zu erwarten, dass das Risiko eines *false positives* aus der Wahrscheinlichkeit für k gesetzte Bits ergibt:

$$f' := (1 - p')^k \text{ und } f := (1 - p)^k$$

Tatsächlich handelt es sich dabei um eine für große Bloom-Filter akzeptable Näherung, nach Bose et al. [2] unterschätzt diese weit verbreitete „klassische“ Formel die Fehlerwahrscheinlichkeit aber für kleine Bloom-Filter. Auf diesen Ergebnissen aufbauend, führte die Analyse von Christensen et al. [4] zu folgender Gleichung:

$$P(x \in V | x \notin S) = f_{neu} := \frac{m!}{m^{k(n+1)}} \sum_{i=1}^{m} \sum_{j=i}^{i} (-1)^{i-j} \frac{j^{kn} i^k}{(m-i)!j!(i-j!)}$$

Die Fehlerwahrscheinlichkeit eines Bloom-Filters kann durch Anpassung der Bitvektorgröße und die gewählte Anzahl der Hashfunktionen optimiert werden. Für große Bloom-Filter ist eine Minimierung der Fehlerrate auf der Basis von f möglich, unter Vorgabe des Verhältnisses m/n (Bits je Element). Im Folgenden wird die diesbezüglich optimale Anzahl an Hashfunktionen bestimmt. Zur Vereinfachung der Herleitung wird anstelle von f der natürliche Logarithmus von f genutzt, der die gleichen Extremstellen wie f aufweist.

$$g := \ln(f) = k \ln(1 - \mathrm{e}^{-kn/m})$$

$$\begin{aligned}
\frac{\delta g}{\delta k} &= \ln(1 - \mathrm{e}^{-kn/m}) + \frac{kn}{m} \frac{\mathrm{e}^{-kn/m}}{1 - \mathrm{e}^{-kn/m}} \stackrel{!}{=} 0 \\
\Leftrightarrow \quad & \ln(1-p) + \frac{kn}{m} \frac{p}{1-p} = 0 \\
\Leftrightarrow \quad & \ln(1-p) = -\frac{kn}{m} \frac{p}{1-p}
\end{aligned}$$

Mit

$$p = \mathrm{e}^{-kn/m} \Leftrightarrow \ln(p) = -\frac{kn}{m}$$

folgt

$$\begin{aligned}
& \ln(1-p) = \ln(p) \frac{p}{1-p} \\
\Leftrightarrow \quad & \ln(1-p)(1-p) = \ln(p)p \\
\Leftrightarrow \quad & \mathrm{e}^{\ln(1-p)(1-p)} = \mathrm{e}^{\ln(p)p} \\
\Leftrightarrow \quad & (1-p)^{(1-p)} = p^p
\end{aligned}$$

Durch Koeffizientenvergleich folgt $p \stackrel{!}{=} \frac{1}{2}$, es handelt sich hierbei um ein globales Minimum. Mit diesem Wert für p folgt für die optimale Anzahl Hashfunktionen:

$$p = \mathrm{e}^{-kn/m} \Leftrightarrow k_{opt} = \frac{m}{n} \ln(2)$$

Und es ergibt sich eine Fehlerwahrscheinlichkeit von:

$$f_{opt} = \left(\frac{1}{2}\right)^{k_{opt}} = \left(\frac{1}{2^{\ln(2)}}\right)^{m/n} \approx 0,6185^{m/n}$$

Die Fehlerwahrscheinlichkeit eines Bloom-Filters hängt nur von der Anzahl Bits je Element ab und ist hinsichtlich der Größe der dargestellten Menge invariant. Der insgesamt erforderliche Speicherplatz ist $\Theta(n)$. Auch die Anzahl der Hashfunktionen ist ausschließlich von der geforderten Fehlerwahrscheinlichkeit abhängig. Ihre Zahl, und damit auch der Berechnungsaufwand, steigt wenn eine geringere Fehlerrate gewünscht ist.

2.3 Vergleich mit Hashlisten

Hashlisten sind bezüglich ihrer Eigenschaften den Bloom-Filtern nicht unähnlich. Die Hashliste speichert je Element einen Hashwert von $\Theta(\log n)$ Bit in einer sortierten Liste. Ein Element wird der durch diese Datenstruktur dargestellten Menge zugeordnet, wenn die Liste den Hashwert des Elements beinhaltet. Es besteht das Risiko einer Hashkollision und damit eines *false positives*. Mit $2\log_2 n$ Bits je Element liegt die Kollisionswahrscheinlichkeit zweier Hashwerte bei

$$\frac{1}{2^{2\log_2 n}} = \frac{1}{n^2} ,$$

die Fehlerwahrscheinlichkeit ergibt sich damit durch die Größe der Hashliste zu

$$n \cdot \frac{1}{n^2} = \frac{1}{n} .$$

Für große Mengen geht die Fehlerrate einer Hashliste gegen Null, bei Bloom-Filtern bleibt sie, bei gleichbleibender Anzahl Bits je Element, konstant. Für praktische Anwendungen ist das Bloom-Filter damit oft schon ausreichend und im Hinblick auf den Speicherbedarf der Hashliste überlegen.

2.4 Eine untere Schranke

Sei U das Universum der Größe $u := |U|$ und $S \subseteq U$ eine Teilmenge der Größe $n := |S|$. Um die $\binom{u}{n}$ möglichen Mengen fehlerfrei zu unterscheiden, ist dann ein Bitvektor V mit $m := |V|$ Bit erforderlich, mit $2^m \geq \binom{u}{n}$.

Mit einer *false-positive*-Rate von ε ordnet V zusätzliche $\varepsilon(u-n)$ Elemente der Menge S zu. Durch diese Unschärfe repräsentiert ein Bitvektor nicht mehr nur eine einzige Teilmenge von U, die Anzahl der dargestellten Mengen beträgt

$$\binom{n+\varepsilon(u-n)}{n} .$$

Folglich reduziert sich die Größe des zur Unterscheidung dieser Teilmengen erforderlichen Bitvektors V:

$$\begin{aligned}
& 2^m \binom{n+\varepsilon(u-n)}{n} \geq \binom{u}{n} \\
\Leftrightarrow \quad & 2^m \geq \frac{\binom{u}{n}}{\binom{n+\varepsilon(u-n)}{n}} \\
\Leftrightarrow \quad & m \geq \log_2 \frac{\binom{u}{n}}{\binom{n+\varepsilon(u-n)}{n}}
\end{aligned}$$

Ist die Menge S gegenüber dem Universum klein, gilt mit $n \ll \varepsilon u$ näherungsweise

$$\begin{aligned}
& m \geq \log_2 \frac{\binom{u}{n}}{\binom{n+\varepsilon(u-n)}{n}} \approx \log_2 \frac{\binom{u}{n}}{\binom{\varepsilon u}{n}} \\
\Rightarrow \quad & m \geq \log_2 \frac{\binom{u}{n}}{\binom{\varepsilon u}{n}} = \log_2 \frac{\frac{u!}{n!(u-n)!}}{\frac{(\varepsilon u)!}{n!(\varepsilon u-n)!}} = \log_2 \frac{n!u!(\varepsilon u-n)!}{n!(u-n)!(\varepsilon u)!} \\
\Leftrightarrow \quad & m \geq \log_2 \frac{u(u-1)\cdots(u-n+1)}{\varepsilon u(\varepsilon u-1)\cdots(\varepsilon u-n+1)} \\
\Leftrightarrow \quad & m \geq \log_2 \frac{u(u-1)\cdots(u-n+1)}{\varepsilon^{u-n+1}(u(u-\frac{1}{\varepsilon})\cdots(u-\frac{n}{\varepsilon}+\frac{1}{\varepsilon}))}
\end{aligned}$$

Mit $n \ll \varepsilon u$ und damit $n \ll u$ folgt

$$\Leftrightarrow \qquad m \geq \log_2 \varepsilon^{-n} = n \log_2(1/\varepsilon)$$

Damit ein Bloom-Filter eine Fehlerrate von mindestens ε zu erreicht, muss gelten:

$$\begin{aligned} & \varepsilon \geq f_{opt} = \left(\frac{1}{2^{\ln(2)}}\right)^{m/n} \\ \Leftrightarrow \quad & \log_2 \varepsilon \geq -\frac{m}{n} \ln 2 \\ \Leftrightarrow \quad & m \geq -n \log_2 \varepsilon \, \frac{1}{\ln 2} \\ \Leftrightarrow \quad & m \geq n \log_2(1/\varepsilon) \log_2 \mathrm{e} \end{aligned}$$

Der Speicherbedarf eines Bloom-Filters liegt bei gleicher Fehlerrate also um mindestens Faktor $\log_2 e \approx 1,44$ über dem der angegebenen unteren Schranke.

3 Erweiterte Formen des Bloom-Filters

3.1 Counting Bloom Filter

Wie bereits in Kapitel 2.1 erläutert wurde, kann jedes Bit in einem Bloom-Filter durch mehr als nur ein Element gesetzt werden. Da bereits ein einziges nicht gesetztes Bit zum Ausschluss aus der Menge führt, ist es nicht ohne weiteres möglich, ein einmal eingefügtes Element aus dem Filter zu entfernen, ohne dabei ungewollt weitere Elemente zu beeinflussen. *Counting Bloom Filter* lösen dieses Problem, indem sie den Bitvektor durch einen Vektor C aus (natürlichen) Zahlen ersetzen. Dieser Vektor wird wie beim „klassischen“ Bloom-Filter über die Hashwerte indiziert, beim Einfügen eines Elements wird der entsprechende Wert inkrementiert (Abbildung 2a) und beim Entfernen dekrementiert (Abbildung (2b). Die Einträge im Vektor zählen so die Anzahl der Abbildungen auf den jeweiligen Hashwert.

Für die Fehlerwahrscheinlichkeit ist der Wertebereich der Zahlen von entscheidender Bedeutung. Ein zu kleiner Zählbereich führt schnell zu einem Überlauf und damit in Folge zu *false negatives*. Nach der Analyse von Fan et al. [5] ist die Wahrscheinlichkeit eines solchen Überlaufs bereits bei vier Bit je Zahl verschwindend gering.

Sei C ein *Counting Bloom Filter*, $C[i]$ der Eintrag für den Hashwert i und j die Anzahl der Zählschritte bis zum Überlauf. Dann beträgt die Fehlerrate mit $k \leq k_{opt}$

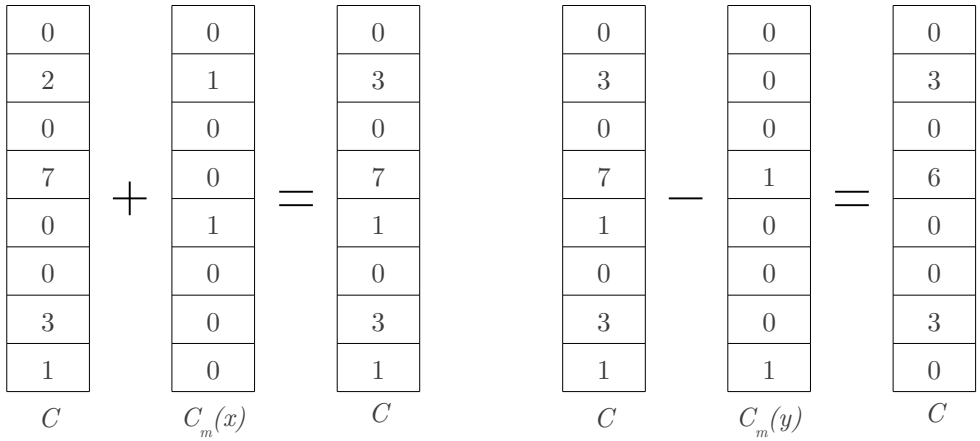
$$\mathbb{P}(\max_i C[i] \geq j) \leq m \left(\frac{\mathrm{e} \ln 2}{j}\right)$$

und bei Zählern mit vier Bit ($j = 16$) ergibt sich ein Wert von

$$\mathbb{P}(\max_i C[i] \geq 16) \leq 1,37m \cdot 10^{-15} \, .$$

3.2 Compressed Bloom Filter

Werden die Parameter eines Bloom-Filters optimal gewählt, so ist die Wahrscheinlichkeit für das Auftreten eines gesetzten oder nicht gesetzten Bits gleich, ($p = \frac{1}{2}$). Der resultierende Bitvektor wäre durch einen Kompressionsalgorithmus nicht weiter zu verkleinern. Für ein *Compressed Bloom Filter* werden die Parameter stattdessen im Hinblick auf die Größe des Filters nach Anwendung eines Kompressionsalgorithmus optimiert, was letztendlich eine



(a) $C \cup x = C$ **(b)** $C \setminus y = C$

Abbildung 2 Verhalten des *Counting Bloom Filters* beim Einfügen und Entfernen von Elementen

Bits je Element	m/n	16	28	48
Bits je Element nach Kompression	z/n	16	15,846	15,829
Hashfunktionen	k	11	4	3
Fehlerrate	f	0,000459	0,000314	0,000222

Tabelle 1 Fehlerrate eines *Compressed Bloom Filters* (Beispiel aus [3])

Reduktion des Anteils der gesetzten Bits im Bloom-Filter bedeutet. Dazu wird die Anzahl der Bits je Element deutlich erhöht und eine suboptimal geringe Anzahl an Hashfunktionen gewählt. Die Fehlerrate kann dabei je nach gewählter Konfiguration konstant gehalten werden, oder deutlich geringer ausfallen.

Nach Mitzenmacher [6] kann mit einem *Compressed Bloom Filter*, etwa bei Netzwerkanwendungen, das erforderliche Übertragungsvolumen z reduziert werden, bei gleichbleibender Fehlerrate. Alternativ erlaubt dieser Ansatz auch eine weitere Reduktion der Fehlerrate. Tabelle 1 zeigt, wie die Fehlerrate bei entsprechend gewählten Parametern gegenüber dem *klassischen* Bloom-Filter zurück geht, während die Größe des *Compressed Bloom Filters* annähernd konstant bleibt.

4 Fazit

Das Bloom-Filter ist eine probabilistische Datenstruktur zur Repräsentation von Mengen. Anders als bei den konzeptuell ähnlichen Hashtabellen können bei Bloom-Filtern Elemente fälschlicherweise der Menge zugeordnet werden (*false positive*), der umgekehrte Fall ist dagegen ausgeschlossen. Die Fehlerwahrscheinlichkeit eines Bloom-Filters ist unabhängig von der Größe der dargestellten Menge, solange das Verhältnis zwischen Bitvektorgröße und Mengengröße konstant bleibt. Alle elementaren Operationen können dabei in $\mathcal{O}(1)$ ausgeführt werden und sind leicht zu parallelisieren.

Durch Bloom-Filter dargestellte Mengen sind, wenn auch eingeschränkt, veränderbar. Elemente können beliebig hinzugefügt werden, dass gilt auch für ganze Mengen (Vereinigung). Der Schnitt mit einer anderen Menge wird durch Bloom-Filter approximiert, *Counting Bloom Filter* ermöglichen zudem das Entfernen von Elementen.

Geeignet sind Bloom-Filter vor allem für Anwendungen bei denen die Ressourcen Speicherplatz, Übertragungsbandbreite oder -volumen stark beschränkt sind und die auftretende *false positives* tolerieren oder entsprechend behandeln können. Die einfache Struktur des Bloom-Filters ermöglicht vielfältige Variationen und Spezialisierungen wie etwa das *Compressed Bloom Filter*, die auf spezifische Bedürfnisse sehr stark zugeschnitten sein können.

Danksagungen Ich möchte mich an dieser Stelle für die hilfreichen Vorschläge und kritischen Anmerkungen bedanken, die im Rahmen des Review-Prozesses geäußert wurden.

Literatur

1 Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

2 Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. On the false-positive rate of bloom filters. *Information Processing Letters*, 108(4):210–213, October 2008.

3 Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

4 Ken Christensen, Allen Roginsky, and Miguel Jimeno. A new analysis of the false positive rate of a bloom filter. *Information Processing Letters*, 110(21):944–949, October 2010.

5 Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

6 Michael Mitzenmacher. Compressed bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pages 144–150, New York, NY, USA, 2001. ACM.

7 Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash- and space-efficient bloom filters. In *Proceedings of the 6th international conference on Experimental algorithms*, WEA'07, pages 108–121, Berlin, Heidelberg, 2007. Springer-Verlag.

Two Simplified Algorithms for Maintaining Order in a List*

Christian Käser[1]

1 **Karlsruhe Institute of Technology, KIT**
Kaiserstraße 12, 76131 Karlsruhe, Germany
christian.kaeser@student.kit.edu

Abstract

The *Order-Maintenance Problem* is about maintaining order in a data structure while handling inserts, deletions and precedence queries. There have been known optimal solutions such as those by Dietz and Sleator [3] for over 20 years, but they are rather complicated and hard to prove. In contrast, the algorithms by Bender et al.[2] that will be presented in this work aim to be much simpler to explain while still maintaining the same performance.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Order-Maintenance, File-Maintenance, Online List Labeling

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Data structures that are suitable to handle the *Order-Maintenance Problem* are called *order data structures* and must provide at least the following operations while always maintaining a total order:

- $Insert(X, Y)$: insert Y directly after X.
- $Delete(X)$: remove X from the data structure.
- $Order(X, Y)$: determine if X is positioned before or after Y in the data structure.

1.1 Related Work

Attempts at creating an efficient order data structures have been around at least since a publication by Dietz [4] in 1982. He proposed a data structure based on an indexed 2-3 tree that supports insertion in amortized $O(log(n))$ and queries in $O(1)$ time. Tsakalidis [7] those bounds by using BB[α]-trees so that not only insertions but also deletions could be done in $O(log * (n))$ amortized time while preserving $O(1)$ worst-case time for order queries.

Further improvements were made by Dietz and Sleator [3] who proposed two new data structures. The first one supports both insertions and deletions in $O(1)$ amortized time and queries in $O(1)$ worst-case time. The second one is a rather complicated data structure based on findings by Willard [8, 9] that supports all operations in $O(1)$ worst-case time.

To overcome the complex and sometimes counter-intuitive proof needed to comprehend those data structures by Dietz and Sleator, Bender et al. [2] proposed another two new data structures in 2002 that were much simpler to proof and explain and will be the main topic of this paper. One held the same $O(1)$ amortized time for updates and $O(1)$ worst case time for

* This work is mainly based on a paper of the same name by Bender et al.[2]

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–7

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

queries as the first one by Dietz and Sleator. They claimed that they could further improve it to support all operations in $O(1)$ worst-case time but didn't go into detail about how this could be achieved. The other one is a *File Maintenance* solution that matches Willard's $O(log^2(n))$ worst-case upper bound but only takes only five instead of about fifty pages to describe.

The *Order-Maintenance Problem* is closely related to the more specialised *Online List Labeling*, also called *File Maintenance* where a dynamic set of n elements is mapped to corresponding integers called tags in the range from 1 to u. Therefore most of the more recent *Order Maintenance* solutions originate from *Online List Labeling* algorithms.

2 Preliminaries

A very basic approach when dealing with maintaining an order is the use of a linked list in combination with a tree-based data structure on top of it. One very primitive variation of this that uses an ordinary binary tree is usually even taught in beginners' courses. To get the order of two entries one can simply traverse the tree upwards until a common ancestor is found. Although insertions can be as fast as $O(1)$ the main problem with this method is that its worst-case query time is proportional to the height of the tree. Depending on the amount of rebalancing that is done after insertions the upper bound for queries can therefore vary between $O(log(n))$ and $O(n)$.

Therefore other solutions are needed for the more common case where there are usually far more order queries than insertions. Bender et al. as well as several others use a list-labeling approach where the bits of the elements' tags correspond to root-to-leaf paths in a *virtual binary tree*. With this representation order queries are reduced to a simple comparison of the two tags' numerical values and can therefore be performed in $O(1)$ worst-case time as long as the tags fit in a fixed number of computer words. The drawback is that an insertion can occur between two elements whose tags differ by only 1 which means there is no space inbetween to add another tag. In this case some of the surrounding tags must be reassigned which can be considerably more expensive than rebalancing of a tree with the usual representation would be.

As has been mentioned, these problems are related to weight balancing of trees. Though there are lots of algorithms to efficiently balance trees such as red-black trees, BB[α] trees [5], skip lists [6], and weight-balanced B-trees [1], not all of these can be used for this virtual tree representation. Where in a conventional representation only a fixed number of pointers must be updated to move an entire sub-tree from one node to another or to swap the order of two children of a common parent node, we would need to update the tags of all descendants to achieve the same effect.

3 The Amortized Order-Maintenance Algorithm

The first algorithm described in [2] uses the afore-mentioned method of storing a tree implicitly but for an explanation and analysis it is useful to imagine a real tree structure. In the following paragraphs u refers to the *universe size*, the maximum number of available tags, while n refers to the number of elements currently stored in the data structure. To keep things simple, u should always be a power of two.

3.1 Description

When we insert a new element f into the list between two elements e and g we can choose any tag between those of e and g. A relabeling step must occur whenever the tags of e and g differ by only one, leaving no room for f inbetween. This is where the tree representation becomes handy. Every node in the tree corresponds to a certain (possibly empty) sub-list containing those elements that occupy leaves below this node. In terms of tags, each internal node is an *enclosing tag range* of its descendant leaves' tags.

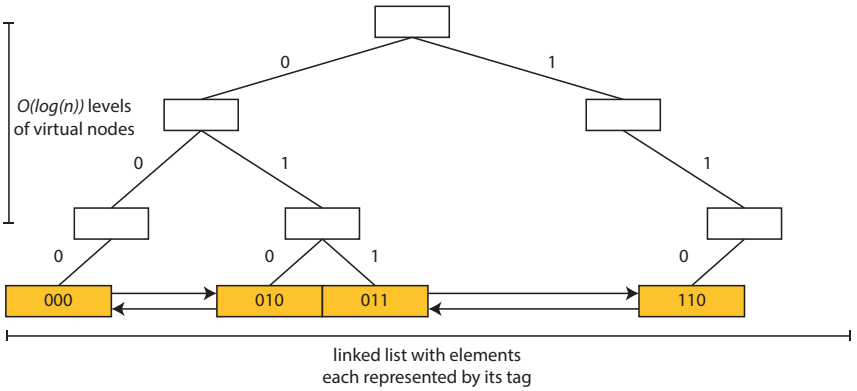


Figure 1 The virtual tree structure utilised in the *Order Maintenance* algorithm

This allows us to regard the *enclosing tag ranges* of e as if we would walk up the actual tree from leaf to root until we find a node that has few enough descendants to make relabeling viable. To define how sparse a subtree should be, we introduce a constant T between 1 and 2 that leads to so-called *overflow thresholds* τ_i for intervals of size 2^i. We define the overflow threshold for a single leaf $\tau_0 := 1$ and all other thresholds as $\tau_i := \tau_{i-1}/T = T^{-i}$. So as soon as we reach a node of level i with a density of at most τ_i (meaning it has no more than $2^i/\tau_i$ descendants) we relabel all of its descendants by evenly spreading out their tags. It is not necessary to further traverse the tree even if there might be more nodes on higher levels that exceed ther thesholds.

Additionally there is a mechanism to keep u in a reasonable range compared to n so that we can always assume that tags never need more than $O(log(n))$ bits. For a given n we choose a value of u that works at least for a number of elements in the range from $n/2$ to $2n$. As soon as n leaves this range, we rebuild the entire data structure with a new value of u. As we need at least $n/2$ deletions or n insertions until the data structure has to be rebuilt and rebuilding leads to no more than $2n$ relabelings, we conclude that rebuilding can be done in constant amortized time. Considering that a real machine has a fixed word size m we can alternatively just set $u := 2^m$ and limit the number of elements the data structure can store.

3.2 Analysis

To to determine the overall amortized cost of insertions first consider the maximum number of relabelings on a single level. To be chosen for relabeling a tag range if size 2^i must have a density of τ_i or less as we only relabel ranges that are not overflowing. This means that a range that gets relabeled contains no more than $2^i\tau_i = (2/T)^i$ tags that must be reassigned leading to a time complexity of $O((2/T)^i)$.

After relabeling, both child sub-ranges have a density of at most τ_i. The current range

will not be relabeled again until at least one of its children reaches a density of $\tau_{i-1} = T\tau_i$ and overflows. Before this happens, at least $(2 - T/2)(2/T)^i$ insertions must be performed. This leads to an amortized insertion cost *per enclosing tag range* of

$$\frac{(2 - T/2)(2/T)^i}{(2/T)^i} = 2 - T/2 = O(1)$$

As each element has $O(log(n))$ such enclosing tag ranges the overall amortized insertion cost is $O(log(n))$. To get this down to $O(1)$ one level of indirection can be added as described in [3].

Deletions are performed by simply deleting an element from the list and rebuilding if needed. As long as the data structure is not rebuilt, no relabeling occurs. Therefore deleting an element needs $O(1)$ amortized time if rebuilding is used or $O(1)$ worst-case time if it is not.

Queries can obviously be done in $O(1)$ worst-case time assuming tags fit in a fixed number of machine words.

3.3 Tradeoffs

The main tradeoff comes with the choice of T as it influences the amortized insertion cost as well as the maximum number of elements the data structure can hold for a given universe size u. A lower value of T leads to less relabeling operations per insertion but also reduces the number of possible elements before the root would overflow. One addition to the algorithm proposed by Bender et al. is the choice of a variable T that is recalculated for every insertion in a way that there is just enough room for the new element. They show that this addition leads to an algorithm that performs well regardless of the number of elements.

Another point to consider is the choice of a new tag when inserting an element. An obvious choice would be the average of both neighbors but as the analysis never relies on any particular strategy for choosing tags it would be just as good to use the tag of e increased by one. Though this might not be intuitive at all, experiments in [2] show that this does only results in an additive performance change instead of the expected constant factor. The reason is that even though we need more relabeling operations, those operations affect less elements each and so the total number of relabeled elements stays roughly the same.

4 File Maintenance

Even though it is much simpler than the algorithm given by Willard [8, 9, 10] the *File Maintenance* algorithm by Bender et al. is still far more complex than the *Order Maintenance* algorithm described in the previous section. As a result, this section will not include all details of how the given performance bounds can be proven and instead only describes how it operates. The reader is encouraged to refer to [2] for details.

4.1 Description

The algorithm aims to store n items in an array of size $O(n)$ instead of a linked list while providing the following operations:

- $Insert(X, Y)$: insert Y directly after a stored item X.
- $Delete(X)$: remove X from the data structure.
- $Scanleft(k, X)$: given a pointer to an alredy stored item X scan the next k items starting with X

- $Scanright(k, X)$: analogous to $Scanleft$

In addition, the $Order(X, Y)$ operation can easily implemented using an index comparison.

Similar to the previous algorithm, we imagine a binary tree of height $O(log(n))$ that partitions the array. One big difference is that a leaf of the tree does not correspond to a single item but to a range of $alog(n)$ items within the array for a suitable value of a. This basically just means that the few lowest level are taken away from the tree and instead handled directly as pieces of continuous space in the array. Those pieces are called *cells*.

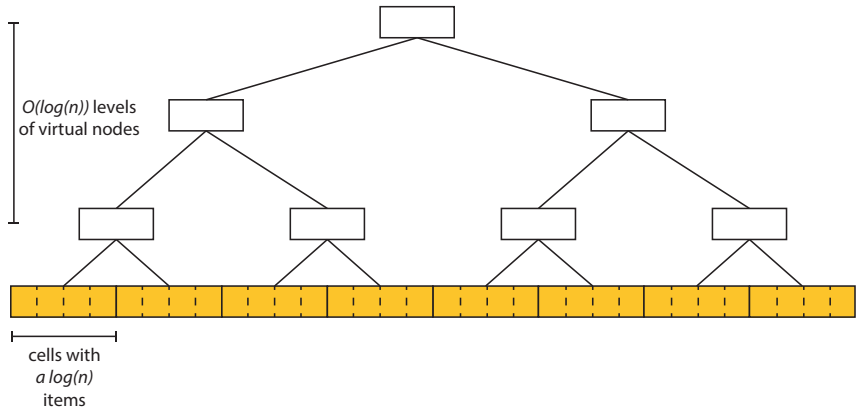


Figure 2 The virtual tree structure utilised in the *File Maintenance* algorithm

As before the levels within the tree are labeled from leaves to root so that the leaves are level 0, the level directly above is level above is level 1 and so on, meaning that a node of level i corresponds to an interval of 2^i cells. This will also be called an i-interval.

Rebalancing is considered after every insertion or deletion. Whenever the difference between the number of items in an i-interval and its sibling is $2 \cdot 2^i$ it is marked for rebalancing. Over the next 2^i insertions and deletions in the respective sub-tree we move 2^i items from the fuller interval to the less full interval. Such an operation is called a $(i + 1)$-interval rebalancing and the piece of work performed during one insertion or deletion is called a *phase*. Note that during every phase more than one individual item can be moved. To keep all intervals of lower levels balanced we make sure the resulting number of elements in each cell changes only by 1.

To maintain the invariant that two sibling i-intervals are imbalanced by at most 2^i items (except temporarily when an ancestor interval is rebalanced), every node must note the imbalance between it's two children. This data is stored in the cell immediately left of the interval's center.

For simplicity this section only describes the process when rebalancing items from left to right as the process works very similarily when applied the other way round.

To rebalance an interval we right-shift items to neighboring cells. Consider $i = 1$ so we have $2^1 = 2$ cells on every side. Initially the left half contains $2^1 = 2$ more items than the right half. We would move 1 item from the first cell to the second, 2 from the second to the third and 1 from the third to the fourth. To avoid overfilling of any cells we don't start from left to right but from right to left or in general from the side that receives items to the side that releases items.

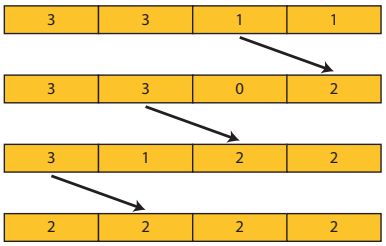


Figure 3 Approximate order in which elements are moved between cells

4.2 Analysis

As stated above, the complete prove is far too long to be included in this paper. Instead only the results will be shown in this section.

It is obvious that order queries can be handled in $O(1)$ worst-case time by simply comparing two element's indices within the array. Scanning operations take $O(k)$ time.

Each $(i+1)$-interval rebalancing touches only $2^(i+1)$ cells of which each can hold up to $a \cdot log(n)$ items. Considering that such a rebalancing is spread out across 2^i insertions or deletions in the corresponding sub-tree that means $2a \cdot log(n)$ items per rebalancing phase. As long as we make sure that every item is only modified a constant number of times this leads us to a time cost of $O(log(n))$ per interval and phase. Because there are $O(log(n))$ levels in the tree, every insertion or deletion can trigger rebalancing phases of up to $O(log(n))$ intervals. This in turn leads to a maximum total of $O(log^2(n))$ items being manipulated per update.

5 Conclusions

Bender et al. managed to create two data structures that match their predecessors' theoretical upper bounds while being more intuitive. At least the $O(1)$ amortized time algorithm is easy enough to explain in a few minutes and can probably be implemented with only a few lines of code.

The *File Maintenance* algorithm is still a little involved as it needs to handle several competing rebalancing operations. In addition a full prove and actual performance measurements have yet to be shown as [2] defers them to "the full paper" which doesn't seem to have ever been published. Therefore it cannot be concluded if this complexity is justified.

References

1 L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, FOCS '96, pages 560–, Washington, DC, USA, 1996. IEEE Computer Society.

2 Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pages 152–164, London, UK, UK, 2002. Springer-Verlag.

3 P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 365–372, New York, NY, USA, 1987. ACM.

4 Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 122–127, New York, NY, USA, 1982. ACM.

5 J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, STOC '72, pages 137–142, New York, NY, USA, 1972. ACM.

6 William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Proceedings of the Workshop on Algorithms and Data Structures*, WADS '89, pages 437–449, London, UK, UK, 1989. Springer-Verlag.

7 Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Inf.*, 21(1):101–112, June 1984.

8 Dan E. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 114–121, New York, NY, USA, 1982. ACM.

9 Dan E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, pages 251–260, New York, NY, USA, 1986. ACM.

10 Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time. *Inf. Comput.*, 97(2):150–204, April 1992.

Splay-Bäume: Theoretische Garantien und praktische Ineffizienz

Dominik Messinger

Karlsruhe Institute of Technology
Kaiserstraße 12, 76131 Karlsruhe, Germany
`dominik.messinger@student.kit.edu`

Zusammenfassung

Splay-Bäume sind selbst-anpassende, dynamische Binäre Suchbäume, in denen zuletzt besuchte Elemente in die Nähe der Wurzel transportiert werden. Durch die zu diesem Zwecke eingesetzte Heuristik kann eine im Worst-Case amortisiert logarithmische Zugriffszeit garantiert werden. Wir geben eine theoretischen Analyse von Splay-Bäumen und präsentieren ausgewählte Splay-Baum-Varianten, die anschließend praktisch evaluiert werden. Im Vergleich mit einfachen Binären Suchbäumen und AVL-Bäumen zeigen wir, dass der Overhead durch Splay-Rotationen auf gleichverteilten oder Zipf-verteilten Eingaben zu Ineffizienz führt. Bei stark wahrscheinlicher direkter Aufeinanderfolge von gleichen Elementen stellen sich Splay-Bäume hingegen als überlegen heraus.

1998 ACM Subject Classification E.1 Data Structures - Trees

Keywords and phrases Data structures – Splay trees – Amortised complexity – Empirical

Digital Object Identifier 10.4230/LIPIcs.9999.9999.9

1 Einleitung

Splay-Bäume, eingeführt von Sleator und Tarjan [9], sind eine Variante von Binären Suchbäumen, dessen Operationen im schlechtesten Fall amortisiert logarithmische Zeit benötigen. Binäre Suchbäume im allgemeinen halten eine Menge von Elementen, deren Schlüssel aus einem Universum mit einer totalen Ordnung stammen. Jeder Knoten eines Binären Suchbaumes hält ein Element und besitzt maximal zwei Kindknoten. Hat ein Knoten ein Element mit Schlüssel i, so sind alle Schlüssel im linken Teilbaum kleiner und alle im rechten Teilbaum größer als i. Um einen Schlüssel im Baum zu finden, wird der Baum von der Wurzel absteigend durchlaufen bis der Schlüssel gefunden wurde. Durch Vergleich des gesuchten Schlüssels mit dem Schlüssel des auf dem Suchpfad aktuellen Knotens ist die Abzweigung in den linken bzw. rechten Teilbaum eindeutig festgelegt. Die Suche benötigt $\Theta(d)$ Zeit, hierbei ist d die Tiefe des Knotens mit dem gesuchten Schlüssel. Ein dynamischer Binärer Suchbaum bietet Operationen zum *Einfügen*, *Löschen* und *Finden* von Schlüsseln und den mit ihnen verbundenen Elementen. Inorder-Traversierung liefert zudem eine sortierte Reihenfolge aller vorhandenen Schlüssel. Durch Augmentierung der Knoten mit zusätzlicher Information wie beispielsweise Teilbaumgrößen bieten Binäre Suchbäume die Möglichkeit für Bereichsabfragen (*range queries*). Da ein Binärer Suchbaum im schlechtesten Fall nur aus einer einzigen, langen Kette von n Knoten bestehen kann, liegt die Worst-Case-Zeit des Einfügens, Löschens und Findens in $\mathcal{O}(n)$. Um diesem zu begegnen, können stattdessen *Balancierte Binäre Suchäume* verwendet werden, die eine Zeit in $\mathcal{O}(\log(n))$ garantieren. Balancierte Bäume halten den Baum in einem ausgeglichenen Zustand, so dass die Höhe stets in $\mathcal{O}(\log(n))$ bleibt. Hierzu sind zusätzliche Informationen zu einem Knoten zu speichern, beispielsweise die Knotenfarbe im Falle von *Rot-Schwarz-Bäumen* [3], oder die Teilbaumgrößen bei *AVL-Bäumen* [1]. *Splay-Bäume* haben keine solchen Zusatzinformationen und sind unbalanciert. Die Worst-Case-Zeit

1st Symposium on Breakthroughs in Advanced Data Structures (BADS'13).
Editor: J. Fischer; pp. 41–52

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

einer einzelnen Operation in einem Splay-Baum liegt wie bei einfachen Binärbäumen in $\mathcal{O}(n)$, amortisiert liegt sie aber in $\mathcal{O}(\log(n))$. Amortisiert bedeutet, dass eine beliebige Folge von m Operationen im schlimmsten Fall $m \cdot \mathcal{O}(\log(n))$ Zeit benötigt.

Der folgende Abschnitt 2 erläutert den Aufbau und das Funktionsprinzip von Splay-Bäumen. Daran angeschlossen betrachten wir in Abschnitt 3 den Beweis für die amortisierte logarithmische Laufzeit. In Abschnitt 4 beschäftigen wir uns mit zwei Splay-Baum-Varianten und evaluieren diese in Abschnitt 5 durch praktische Experimente im Vergleich mit unbalancierten Binärbäumen und AVL-Bäumen.

2 Splay-Bäume

Splay-Bäume sind selbst-anpassende binäre Suchbäume, d.h. sie wenden während jeder auf ihr durchgeführten Operation eine Restrukturierungsregel an [9]. Das unterscheidet sie von balancierten Bäumen, die eine die Gesamtstruktur des Baumes betreffende Invariante zu erhalten suchen. Splay-Bäume verwenden eine restrukturierende Heuristik, das *splaying*. Nach jedem Zugriff auf ein Element durch Einfügen, Finden oder Löschen wird auf dem entsprechenden Knoten (Elternknoten bei Löschen oder Nicht-Finden) die Splay-Operation aufgerufen. Das Einfügen, Finden und Löschen ist zunächst identisch zu den entsprechenden Operationen im einfachen Binären Suchbaum, jedoch veranlasst das Splaying dass der Knoten entlang seines Pfades im Baum nach oben rotiert wird, bis er schließlich die bisherige Wurzel ersetzt. Die intuitive Idee dahinter ist das Ausnutzen des *Lokalitätsprinzips*, das besagt, dass auf kürzlich zugegriffene Elemente in naher Zukunft wahrscheinlich wieder zugegriffen werden wird. Die durchzuführenden Splay-Rotationen hängen von der Position des Knotens zu seinem Eltern- und zu seinem Großelternknoten ab und lassen sich in drei Fälle unterteilen, die im folgenden aufgeführt werden (vgl. [9]). Dabei hat jeder Fall auch eine gespiegelte Regelanwendung, die wir hier der Übersicht halber nicht gesondert aufführen. Wir bezeichnen in den Erklärungen den Knoten, auf dem *splay* aufgerufen wurde, mit x, seinen Elternknoten mit y und seinen Großelternknoten mit z.

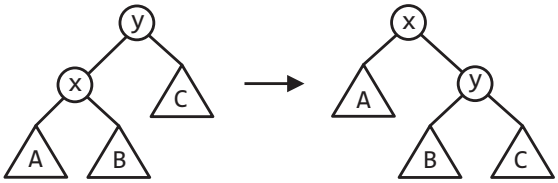


Abbildung 1 Zig case

Zig case (Abb. 1): Wenn der Elternknoten y die Wurzel ist, dann rotiere die Kante zwischen x und y. Dieser Schritt ist der letzte in einer Splay-Operation.

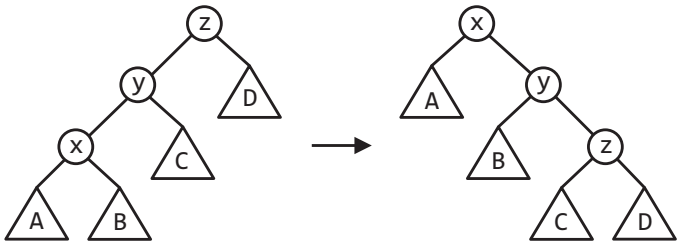


Abbildung 2 Zig-zig case

Zig-zig case (Abb. 2): Ist y nicht die Wurzel und sind x und y beide linke bzw. beide rechte Kinder, so rotiere die Kante zwischen y und z, dann rotiere die Kante zwischen x und y.

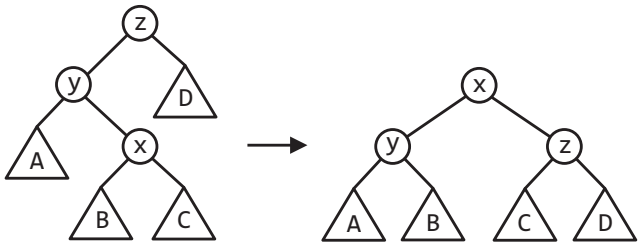


Abbildung 3 Zig-zag case

Zig-zag case (Abb. 3): Ist y nicht die Wurzel und ist x ein linkes und y ein rechtes Kind, oder andersherum, so rotiere die Kante zwischen x und y, dann rotiere die Kante zwischen x und z.

3 Amortisierte Analyse

Es soll gezeigt werden, dass die amortisierte Zeitkomplexität der Splay-Operation in einem Baum mit n Knoten im Worst Case in $\mathcal{O}(\log(n))$ liegt. Der Aufwand für eine Splay-Operation unterscheidet sich von dem Aufwand einer Einfügen-, Löschen- oder Finden-Operation nur um einen konstanten Faktor. Jede der genannten Operationen löst eine Splay-Operation aus, die den bereits hinabgewanderten Suchpfad in halb so vielen Schritten emporsteigt. Rotationen auf diesem Weg benötigen nur konstante Zeit. Eine amortisierte Laufzeitschranke von $\mathcal{O}(\log(n))$ für Splaying gilt dann auch für die genannten Baumoperationen. Der folgende Beweis wurde in leicht abgeänderter Form aus [9] übernommen. Wir verwenden die Potentialmethode [6], um die genannte Schranke zu zeigen. Die aufsummierte amortisierte Zeit für eine Folge von m Operationen ist

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{m} t_i + \Phi_m - \Phi_0 \tag{1}$$

Hierbei sei a_i der amortisierte und t_i der tatsächliche Zeitaufwand der Operation i; Φ_i sei das Potential nach Ausführung der Operation i und Φ_0 sei das initiale Potential. Finden wir eine geeignete Potentialfunktion, die $\Phi_m - \Phi_0 >= 0$ erfüllt, so ist die amortisierte Zeit für eine Sequenz von m Operationen eine obere Schranke für die tatsächliche Zeit.

Bevor wir eine geeignete Potentialfunktion festlegen, definieren wir zunächst die *Größe* $s(x)$

eines Knotens x als die Anzahl der Knoten des Teilbaums mit Wurzel x, x inklusive. Den *Rang* definieren wir als $r(x) = \log(s(x))$. Die Größe eines Knotens ist kleiner oder gleich der Größe seines Elternknotens, analog für den Rang. Wir setzen die Potentialfunktion als die Summe aller Ränge im Baum.

$$\Phi := \sum_{x \in T} r(x) \tag{2}$$

Wir unterteilen eine Splay-Operation in Splay-Schritte, ein Schritt entspricht der Behandlung eines der in Abschnitt 2 skizzierten Fälle. Für die Kostenanalyse messen wir die tatsächliche Zeit als die Anzahl durchzuführender Rotationen.

▶ **Lemma 1** (Access Lemma). *Die amortisierte Zeit für eine Splay-Operation an einem Knoten* x *in einem Baum mit Wurzel* t *ist maximal* $3(r(t) - r(x)) + 1 = \mathcal{O}(\log(s(t)/s(x)))$.

Beweis. Seien s, s', r und r' die Funktionsbezeichnungen für die Größe bzw. den Rang jeweils vor und nach einem Splay-Schritt. Zunächst zeigen wir, dass die Zeit eines Splay-Schrittes im (die Operation abschließenden) *zig-case* maximal $1 + 3(r'(x) - r(x))$ und im *zig-zig* oder *zig-zag case* maximal $3(r'(x) - r(x))$ beträgt. Wir bezeichnen den Elternknoten von x mit y und den Elternknoten von y mit z, sofern vorhanden. Wir unterscheiden 3 Fälle nach der Art der durchzuführenden Rotationen.

Fall 1. *zig case*: Es wird genau eine Rotation mit der Wurzel benötigt.
Die amortisierte Zeit dieses Schrittes ist

$$\begin{aligned}
a_{zig} &= 1 + \Phi' - \Phi \\
&= 1 + r'(x) + r'(y) - r(x) - r(y) && \text{nur Ränge von } x \text{ und } y \text{ ändern sich} \\
&\leq 1 + r'(x) - r(x) && r'(y) \leq r(y) \\
&\leq 1 + 3(r'(x) - r(x)) && r'(x) \geq r(x)
\end{aligned}$$

Fall 2. *zig-zig case*: Zwei Rotationen werden benötigt, die amortisierte Zeit des Schrittes ist

$$\begin{aligned}
a_{zig-zig} &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) && \text{nur Ränge von } x, y, z \text{ ändern sich} \\
&\leq 2 + r'(y) + r'(z) - 2r(x) && r'(x) = r(z) \text{ und } r(x) \leq r(y) \\
&\leq 2 + r'(x) + r'(z) - 2r(x) && r'(y) \leq r'(x) \\
&\overset{!}{\leq} 3(r'(x) - r(x))
\end{aligned}$$

Wir formen geschickt um.

$$\begin{aligned}
& 2 + r'(x) + r'(z) - 2r(x) && \leq && 3(r'(x) - r(x)) \\
\Leftrightarrow \quad & 2r'(x) - r'(z) - r(x) && \geq && 2 \\
\Leftrightarrow \quad & -2r'(x) + r'(z) + r(x) && \leq && -2 \\
\Leftrightarrow \quad & \log(\frac{s(z)}{s'(x)}) + \log(\frac{s(x)}{s'(x)}) && \leq && -2
\end{aligned}$$

Für die letzte Ungleichung benutzen wir, dass $\log(p) + \log(q) \leq -2$ für $p, q > 0$ und $p + q \leq 1$ gilt. Dies ist leicht zu zeigen:

$$\begin{aligned}
& \log(p) + \log(q) \\
\leq\ & \log(p) + \log(1 - p) && q \leq 1 - p
\end{aligned}$$

und -2 ist ein globales Maximum, denn

$$\frac{\partial}{\partial p}(\log(p)+\log(1-p)) = \frac{1}{p}+\frac{1}{1-p} \stackrel{!}{=} 0 \quad \Leftrightarrow \quad p=\frac{1}{2} \qquad \textit{Notw.Bed.}$$

$$\frac{\partial^2}{\partial p^2}(\log(p)+\log(1-p)) = -\frac{1}{p^2}-\frac{1}{(1-p)^2} < 0 \qquad \textit{Hinr.Bed.}$$

$$\log(\frac{1}{2})+\log(1-\frac{1}{2}) = -2$$

Durch Betrachtung der *zig-zig*-Rotation ist $s(x)+s'(z) \leq s'(x)$ ersichtlich; um dieses einzusehen, betrachte man die alten und neuen Positionen der umgehängten Teilbäume. Somit gilt nun wie gefordert $s(z)/s'(x)+s(x)/s'(x) \leq 1$.

Fall 3. *zig-zag case*: Zwei Rotationen werden benötigt, die amortisierte Zeit des Schrittes ist

$$\begin{aligned} a_{zig-zag} &= 2+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z) && \text{nur Ränge von } x,y,z \text{ ändern sich} \\ &\leq 2+r'(y)+r'(z)-2r(x) && r'(x)=r(z) \text{ und } r(x)\leq r(y) \end{aligned}$$

Wir behaupten: Der Term $2(r'(x)-r(x))$ ist eine obere Schranke zur letzten Summe. Somit haben wir $-2r'(x)+r'(y)+r'(z) \leq -2$ zu zeigen. Dies lässt sich analog zu *Fall 3* durchführen. Aus der Betrachtung der *zig-zag*-Rotation ist wiederum $s'(y)+s'(z) \leq s'(x)$ ersichtlich.

Die obere Schranke der Summe der amortisierten Zeiten über alle Zwischenschritte einer Splay-Operation ergibt sich durch Auflösen der Teleskopsumme somit als $1+3(r'(x)-r(x))$. Hierbei bezeichne r' den Rang nach der gesamten Splay-Operation und r den Rang davor. Aus $r'(x)=r(t)$ folgt das Lemma. ◀

Mithilfe des Access Lemma lässt sich das Balance Theorem herleiten.

▶ **Theorem 2** (Balance Theorem). *Die tatsächlichen Gesamtkosten für m Splay-Operationen in einem n-Knoten Splay-Baum betragen maximal $\mathcal{O}(m\log(n)+n\log(n))$.*

Beweis. Die amortisierten Kosten einer Splay-Operation sind nach Access Lemma maximal $3(r(t)-r(x))+1 \leq 3(\log(n)-0)+1$. Für die tatsächlichen Gesamtkosten einer Sequenz von m Splay-Operationen addieren wir zu den m-fachen amortisierten Kosten das Anfangspotential und subtrahieren das Endpotential. Das Anfangspotential ist maximal $n\log n$ und das Endpotential ist mindestens 0. Die Gesamtkosten sind dann $\mathcal{O}(m\log(n)+n\log(n))$. ◀

Für eine Sequenz der Länge $m=\Omega(n)$ liegt eine Splay-Operation amortisiert in $\mathcal{O}(\log(n))$. Dies folgt nicht direkt aus dem Access Lemma, da die mit der Potentialfunktion definierten amortisierten Kosten für Löschen negativ sein können und die tatsächlichen Kosten so unterschätzt werden. Da die Kosten für Einfügen, Finden, Löschen sich nur konstant von den Splaying-Kosten unterscheiden, sind Splay-Bäume laut Balance Theorem für ausreichend lange Sequenzen genauso effizient wie balancierte Bäume.

4 Splay-Baum - Varianten

4.1 Top-Down Splay-Baum

Der ursprünglich vorgestellte und in Abschnitt 2 beschriebene Splay-Baum führt die Splay-Operation *Bottom-Up* durch: Zunächst wird der Baum von der Wurzel hinab bis zur (Einfüge-)Position des geforderten Schlüssels durchlaufen. Anschließend werden von dort Splay-Schritte

hinauf bis zur Wurzel durchgeführt. Eine Splay-Baum-Variante, die solch ein zweimaliges Durchlaufen des Pfades zwischen Wurzel und Schlüsselposition vermeidet, stellten Sleator und Tarjan zugleich mit der Einführung von Splay-Bäumen vor [9]. Deren Besonderheit besteht in einer *Top-Down* agierenden Splay-Operation. Hierbei wird entweder der geforderte Knoten zur neuen Wurzel oder, falls dieser nicht im Baum vorhanden ist, der zuletzt besuchte Knoten auf dem Suchpfad.

Beim Top-Down-Splaying wird der Ursprungsbaum während des Durchlaufens des Baumes auf der Suche nach Knoten x in drei Teilbäume unterteilt: x und seine Kinder bilden den ersten Teilbaum. L hält den Teil des Ursprungsbaumes, dessen Knoten kleiner als x sind und sich nicht im in x wurzelnden Teilbaum befinden. Analog hält R den größeren Teil des Ursprungsbaumes. Wie in der Bottom-Up-Vorgehensweise betrachtet der Algorithmus pro Schritt drei Knoten: den aktuell tiefsten Knoten x, seinen Elternknoten y und seinen Großelternknoten z. Abhängig von deren Positionen zueinander werden wie im Bottom-Up-Algorithmus 3 Fälle unterschieden (siehe Abb. 4).

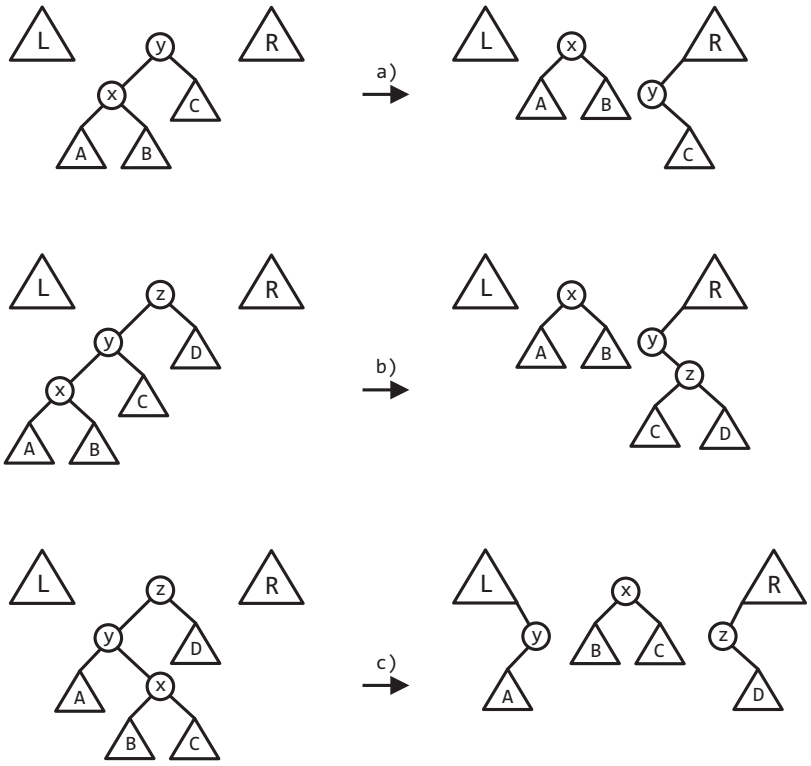


Abbildung 4 Top-Down-Splaying: a) zig-case, b) zig-zig case, c) zig-zag case

zig case: Wenn x der gesuchte Knoten ist, dann hänge den Teil oberhalb von x an R (wenn $x < y$) bzw. L (sonst).

zig-zig case: Wenn x und y beide linke Kinder oder beide rechte Kinder sind, dann rotiere die y-z-Kante und füge den Teil oberhalb von x in L bzw. R ein.

zig-zag case: Wenn x ein linkes Kind und y ein rechtes Kind sind, oder andersherum, dann trenne y und z und füge sie mit ihrem noch verbliebenen (linken oder rechten) Teilbaum in L und R bzw. in R und L ein.

Ist die (Einfüge-)Position des geforderten Schlüssels erreicht, so wird der resultierende Gesamtbaum wie in Abb. 5 gezeigt zusammengesetzt. Dazu wird der linke Teilbaum des zuletzt erreichten x rechts an L angehängt. Genauso wird x' rechter Teilbaum links an R gehängt. Zum Schluss wird L linkes und R rechtes Kind von x, das somit nun die Wurzel des Gesamtbaumes bildet.

Neben der nur einmaligen Pfadverfolgung besitzt der Top-Down-Splay-Baum noch einen weiteren Vorteil gegenüber dem ursprünglichen Bottom-Up-Splay-Baum. Ein Bottom-Up-Splay-Baum muss es ermöglichen, einen bereits besuchten Pfad aufwärts zurückgehen zu können. Techniken hierfür sind Eltern-Zeiger oder das Speichern der besuchten Knoten beim Abstieg; beide Techniken führen zu einem höheren Speicherbedarf im Vergleich zur Top-Down-Variante [9]. Eine andere Möglichkeit ist die Verwendung von zwei Zeigern, die nicht auf den linken und rechten Teilbaum eines Knotens x verweisen, sondern auf den linken Teilbaum und rechten Geschwisterknoten (wenn x linkes Kind) bzw. auf den Elternknoten (wenn x rechtes Kind). Dies führt zwar zum selben Speicherbedarf wie der eines Top-Down-Splay-Baumes oder einfachen Binärbaumes, bringt aber durch die häufigeren Indirektionen Geschwindigkeitseinbußen mit sich [9].

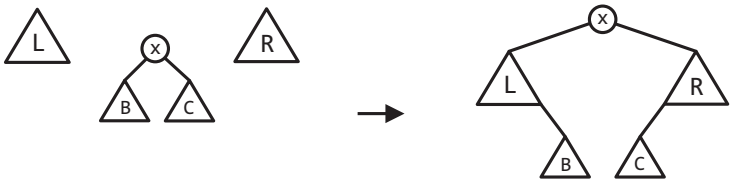


Abbildung 5 Top-Down-Splaying: Abschließendes Zusammensetzen

4.2 Randomisierter Splay-Baum und k-Splaying

Albers und Karpinski [2] stellen eine randomisierte Splay-Baum-Variante vor. Bei einem Zugriff auf einen Knoten wird mit einer festen Wahrscheinlichkeit p eine (Top-Down) Splay-Operation ausgeführt und mit der Wahrscheinlichkeit $1 - p$ bleibt der Baum unverändert. Der zweite Fall entspricht einem einfachen Abstieg im Binären Suchbaum. Empirisch lässt sich für $p = \frac{1}{2}$ eine gegenüber nicht-randomisierten Splay-Bäumen verbesserte Laufzeit zeigen. Die Autoren schlagen vor, deterministisch nach einer festen Anzahl von Operationen Splaying durchzuführen – statt zufällig mit einer Wahrscheinlichkeit p –, um den Overhead der Zufallsgenerierung zu vermeiden [2]. Diese deterministische Variante wird in [7] als *k-Splaying* bezeichnet.

5 Experimente

In der folgenden empirischen Analyse möchten wir die Laufzeiten von Splay-Bäumen in der Praxis untersuchen. Hierzu verwenden wir verschiedene Schlüssel-Zugriffsfolgen, um anwendungsbezogene Hinweise für den Einsatz von Splay-Bäumen geben zu können. Wir vergleichen Splay-Bäume mit einfachen Binären Suchbäumen und AVL-Bäumen.

5.1 Implementierung und Hardware

Alle zu evaluierenden Datenstrukturen sowie das Test-Framework wurden in *C* geschrieben und mit Visual Studio 2010 in der *Release*-Konfiguration auf einem Windows 7 64-bit System

kompiliert. Für die Experimente kam dieselbe Maschine zum Einsatz; sie besitzt eine Intel Core2Duo P8600 CPU à zwei Kerne zu je 2,4 GHz und 3 GB RAM. Die verwendeten Datenstruktur-Implementierungen werden im folgenden beschrieben. Bis auf den Splay-Baum handelt es sich bei den verwendeten Datenstrukturen um Eigenimplementierungen bzw. Modifikationen.

Original Splay-Baum Der Quellcode des Splay-Baumes ist eine Implementierung des Top-Down-Splayings von Daniel Sleator [8].

Splay-Baum Eine Modifikation des *Original Splay-Baum*, die eine bessere Speicherverwaltung umsetzt.

k-Splay-Baum Basiert auf *Splay-Baum* und führt nur jede k-te Operation Splaying durch.

Binärer Suchbaum Ein unbalancierter Binärer Suchbaum.

AVL-Baum Ein balancierter Binärer Suchbaum nach [1], der durch Rotationen eine maximale Höhendifferenz zweier benachbarter Teilbäume von 1 bewahrt.

5.2 Zugriffsfolgen

Um die Laufzeit der Datenstrukturen zu testen, wurden verschiedene Folgen von ganzzahligen Schlüsseln nacheinander in die zu testende Datenstruktur eingefügt. Dabei resultiert ein Aufruf mit einem neuen Schlüssel in einem Einfügen eines neuen Knotens und ein Aufrufen mit einem vorhandenen Schlüssel entspricht einem Abstieg bis zum Knoten mit dem geforderten Schlüssel. Wir beginnen mit einem leeren Baum, der mit der Eingabe wächst.

Gleichverteilt Die Schlüssel werden gleichverteilt aus dem Intervall $[0, 32767]$ gezogen.

Zipf-verteilt Die Schlüssel entsprechen der *Zipf-Verteilung*, d.h. die Wahrscheinlichkeit eines Schlüssels i ist

$$p_i = \frac{1}{i} \cdot \frac{1}{H_i} \tag{3}$$

Dabei ist H_i die i-te harmonische Zahl. Die Zipf-Verteilung beschreibt sowohl die Worthäufigkeiten der englischen Sprache [11] als auch die Häufigkeiten von Webseitenzugriffen [5] und wurde bereits in vorangegangenen Veröffentlichungen zur Analyse von Splay-Bäumen eingesetzt [2][7].

Vorgänger-affin Die Schlüssel werden mit fester Wahrscheinlichkeit entweder gleichverteilt aus dem Intervall $[0, 32767]$ gezogen oder erhalten den Wert des vorangegangenen Schlüssels. Somit gilt für den i-ten Schlüssel:

$$k_i \leftarrow \begin{cases} k_{i-1} & \text{mit Wahrscheinlichkeit } p \\ U_{[0,32767]} & \text{mit Wahrscheinlichkeit } 1-p \end{cases} \tag{4}$$

NIV Die Schlüssel sind die *djb2*-Hashwerte [12] der Wörter der englischen Bibel in der *New International Version* (NIV). Dabei wurde die Reihenfolge beibehalten und Vers-, Kapitelangaben sowie Satzzeichen wurden ausgelassen. Nach dieser Zählung hat die NIV 7.208.209 Wörter, darunter sind 725.909 paarweise verschieden.

5.3 Speicheroptimierung der vorhandenen Implementierung

Die *Original Splay-Baum*-Implementierung alloziert vor dem Einfügen eines Schlüssels Speicherplatz für einen Knoten und gibt diesen wieder frei, falls der Schlüssel bereits im Baum existiert. Dieses führt in der Praxis zu erheblich schlechteren Laufzeiten, als wenn Speicher

auf dem Heap nur bei tatsächlichem Bedarf alloziert wird. So ist die Laufzeit von *Original Splay-Baum* gegenüber dem verbesserten *Splay-Baum* um den Faktor 8 bei 1 Million gleichverteilten Schlüsseln und um den Faktor 14 bei 10 Millionen gleichverteilten Schlüsseln langsamer.

5.4 Vergleich mit unbalanciertem Binärbaum und AVL-Baum

Die im folgenden präsentierten Werte sind jeweils das arithmetische Mittel aus 3 Messungen, die Standardabweichungen folgen in Klammern. Die Schlüsselfolgen und Baumstrukturen passten in den Hauptspeicher der verwendeten Maschine.

#Zugriffe (in Mio.)	Splay-Baum	k-Splaying (k=2)	k-Splaying (k=32)	BST	AVL
1	271 (9)	270 (9)	260 (9)	218 (1)	187 (0)
10	2371 (0)	2319 (18)	2183 (16)	1789 (17)	1472 (9)
20	4686 (18)	4591 (9)	4306 (16)	3505 (18)	2902 (16)
30	7025 (48)	6850 (29)	6422 (50)	5242 (16)	4311 (33)
40	9313 (41)	9142 (54)	8549 (42)	6947 (9)	5752 (39)

Tabelle 1 Laufzeitmessungen (in Millisekunden) für gleichverteilte Zugriffsfolgen

Aus Tabelle 1 und Abb. 6 lässt sich für gleichverteilte Zugriffe unabhängig von der Anzahl eine klare Reihenfolge zwischen den einzelnen Datenstrukturen ableiten. Der Top-Down *Splay-Baum* ist die langsamste Datenstruktur, dicht gefolgt von *k-Splaying (k=2)*, welches unwesentlich schneller ist. Es folgt *K-Splaying (k=32)* und der *Binäre Suchbaum (BST)*. Am effizientesten ist der *AVL-Baum*. Die Splay-Operationen bringen keine Performance-Vorteile und eine Reduktion derselben führt zu besseren Laufzeiten. Die in Relation zu den anderen Zugriffsfolgenlängen größeren Laufzeiten pro Zugriff für eine Folge von 1 Million Zugriffen lassen sich durch Messungenauigkeiten erklären, die bei ca. 60 ms auf die in diesem Fall gemessene Zeit von 180 bis 280 ms beinahe 30% ausmachen können. Die gemessenen Ergebnisse stimmen mit [4] überein, dessen Autoren für gleichverteilte Zugriffe eine schlechtere Performance von Top-Down-Splay-Bäumen gegenüber AVL-Bäumen feststellten. Die von uns gemessenen Werte bestätigen eine Verbesserung durch k-Splaying gegenüber Standard Top-Down-Splay-Bäume wie in [2] beschrieben.

#Zugriffe (in Mio.)	Splay-Baum	k-Splaying (k=2)	k-Splaying (k=32)	BST	AVL
1	66 (8)	64 (3)	56 (12)	46 (15)	55 (8)
10	572 (9)	557 (9)	457 (9)	374 (1)	494 (18)
20	1129 (9)	1092 (0)	893 (22)	738 (9)	962 (24)
30	1643 (18)	1622 (15)	1300 (9)	1092 (42)	1456 (24)
40	2168 (16)	2131 (18)	1716 (0)	1414 (24)	1867(109)

Tabelle 2 Laufzeitmessungen (in Millisekunden) für Zipf-verteilte Zugriffsfolgen

Während bei Zipf-verteilten Zugriffsfolgen die Performance-Reihenfolge zwischen den *Splay-Baum-Varianten* und dem *BST* gleich bleibt, ist der *AVL-Baum* durchgehend langsamer als der *BST* und als *k-Splaying (k=32)* (vgl. Tabelle 2 und Abb. 7). Eine mögliche Erklärung ist die in der Zipf-Verteilung gegenüber den gleichverteilten Zugriffen reduzierte Anzahl paarweise verschiedener Schlüssel, das Verhältnis beträgt ungefähr 2000 zu 32768. Somit sind die Bäume kleiner und Rotationen zur Balanceaufrechterhaltung nicht rentabel. Obwohl Splay-Bäume von dem häufigen Auftreten einiger weniger Schlüssel in der Zipf-Verteilung profitieren könnten, tun sie dies entgegen der Intuition in der Praxis nicht. Zum selben

Ergebnis kommen die Autoren von [7] und [4] in ihren Praxis-Vergleichen mit BSTs bzw. AVL-Bäumen.

Vorgänger-gleichheits-Wahrschein-lichkeit	Splay-Baum	k-Splaying (k=2)	k-Splaying (k=32)	BST	AVL
0,0	9230 (55)	9059 (45)	8502 (31)	6874 (156)	5715 (39)
0,1	8604 (48)	8590 (37)	8195 (59)	6781 (113)	5543 (17)
0,2	7675 (31)	7946 (33)	7738 (16)	6406 (99)	5278 (59)
0,3	6906 (86)	7660(301)	7384 (24)	6328 (125)	5132(107)
0,4	5944 (16)	6630 (41)	6895 (47)	5876 (100)	4739 (36)
0,5	5065 (18)	5736 (18)	6417 (23)	5527 (131)	4555 (68)
0,6	4103 (27)	4779 (18)	5834 (16)	5289 (159)	4212 (27)
0,7	3250 (77)	3827 (24)	5289 (16)	4830 (259)	3998 (26)
0,8	2277 (16)	2761 (16)	4607 (24)	4415 (151)	3681 (16)
0,9	1352 (9)	1617 (9)	3530 (9)	3982 (294)	3343 (24)

Tabelle 3 Laufzeitmessungen (in Millisekunden) für Vorgänger-affin-gleichverteilte 40 Mio. Zugriffe

Die in Abschnitt 5.2 beschriebene *Vorgänger-affine Gleichverteilung* generiert Zugriffsmuster, bei denen Splay-Bäume schneller als BSTs und AVL-Bäume sein können. Ab einer Wahrscheinlichkeit der Gleichheit mit dem vorigen Schlüssel von 60% führen Splay-Rotationen zu den im Vergleich schnellsten Laufzeiten. Rotationsreduktionen durch k-Splaying verschlechtern die Laufzeit (vgl. Tabelle 3, Abb. 8).

	Splay-Baum	k-Splaying (k=2)	k-Splaying (k=32)	BST	AVL
Zeit (ms)	1258 (32)	1258 (3)	1155 (1)	1045 (1)	1082 (8)

Tabelle 4 Laufzeitmessungen (in ms) für 20x konkatenierte NIV-Bibel

Um dem bereits erwähnten Problem eines großen Verhältnisses von Messungenauigkeit zu Laufzeit zu entgehen, wurde für die in Tabelle 4 aufgeführten Messungen die NIV-Bibel 20-mal konkateniert. Wie durch die Messergebnisse bei Zipf-verteilten Zugriffen bereits zu erwarten, sind Splay-Baum-Varianten langsamer als BSTs und AVL-Bäume, dabei sind BSTs wie bei der Zipf-Verteilung die schnellste Datenstruktur. Unsere Messungen auf den real-world-Bibeltexten bestätigen bestehende Ergebnisse von [10], in denen Top-Down-Splay-Bäume auf Textkollektionen schlechtere Laufzeiten als BSTs und Rot-Schwarz-Bäume aufwiesen.

6 Fazit

Splay-Bäume sind einfach zu implementierende Binäre Suchbäume, die amortisiert logarithmische Zugriffszeiten bieten. Damit sind sie im Worst-Case besser als einfache Binäre Suchbäume und für Zugriffsfolgen im Durchschnitt genauso gut wie balancierte Suchbäume. Dabei sind sie einfacher zu implementieren als AVL-Bäume und benötigen nicht mehr Strukturinformation als einfache Binäre Suchbäume. In diesem Artikel haben wir die Splay-Operation erklärt und die genannte Laufzeitschranke wie in [9] bewiesen. Wir haben zwei Splay-Baum-Varianten erläutert; Top-Down-Splay-Bäume und Randomisierte Splay-Bäume bzw. k-Splaying. In Experimenten konnten wir zeigen, dass Splay-Bäume entgegen ihrer guten theoretischen Schranken in der Praxis schlechtere Laufzeiten als BSTs und AVL-Bäume für solche Zugriffsfolgen aufweisen, die einer Gleichverteilung oder einer Verteilung, die Worthäufigkeiten in Texten entspricht, unterliegen. Ähnliche Ergebnisse lassen sich bereits in verwandten

Veröffentlichungen finden [7][10][4]. Wir konnten aber zeigen, dass Splaying sich lohnt, wenn auf den zur Wurzel beförderten Knoten mit mindestens 60-prozentiger Wahrscheinlichkeit im nächsten Schritt wieder zugegriffen wird. Für eine Anwendung, die dieser Bedingung genügt, sind Splay-Bäume den einfachen BSTs und balancierten Bäumen vorzuziehen.

A Laufzeitdiagramme

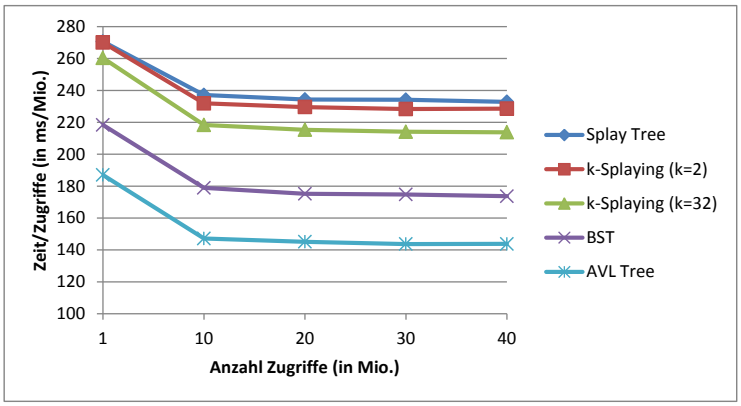


Abbildung 6 Zeit pro Mio. Zugriffe (in ms/Mio.) für gleichverteilte Zugriffsfolgen

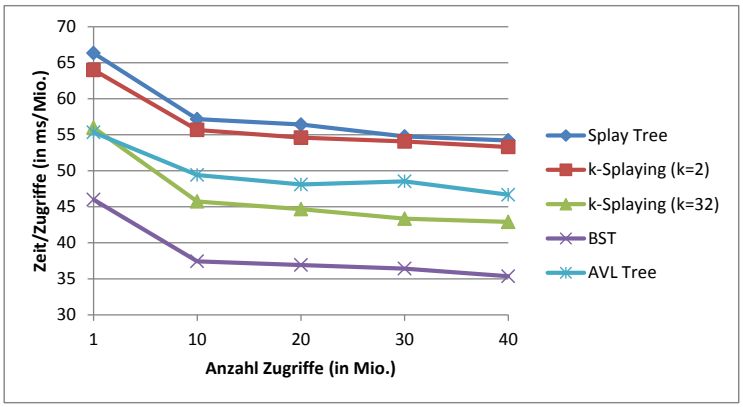


Abbildung 7 Zeit pro Mio. Zugriffe (in ms/Mio.) für Zipf-verteilte Zugriffsfolgen

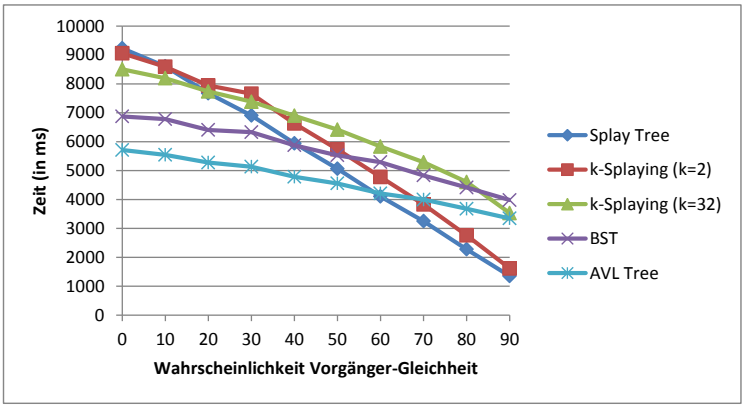


Abbildung 8 Zeit (in ms) für 40 Millionen Vorgänger-affin-gleichverteilte Zugriffe

Literatur

1 G. M. Adelson-Velsky and E.M. Landis. *An algorithm for the organization of information.* Soviet Mathematics Doklady 3 (S.1259-1263), 1962

2 Susanne Albers and Marek Karpinski. *Randomized splay trees: Theoretical and experimental results.* Information Processing Letters Vol. 81 No. 4 (S. 213-221), 2002

3 Rudolf Bayer. *Symmetric binary B-Trees: Data structure and maintenance algorithms.* Acta Informatica Vol. 1 (S. 290-306), 1972

4 Jim Bell and Gopal Gupta. *An evaluation of self-adjusting binary search tree techniques.* Software-Practice and Experience Vol.23, James Cook University, Townsville, Queensland, Australia, 1993

5 L. Breslau, P. Cao, L. Fan, G. Phillips, Scott J. Shenker. *Web caching and Zipf-like distributions: evidence and implications.* Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '99) Vol.1, S. 126-134, 1999

6 Th. H. Cormen, Ch. E. Leiserson, R. Rivest, C. Stein. *Algorithmen - Eine Einführung.* 2. Auflage, Oldenbourg, 2007

7 Eric K. Lee and Charles U. Martel. *When to use splay trees.* University of California at Davis, Davis, CA, USA, 2007

8 Daniel Dominic Sleator. *An implementation of Top-Down splaying.* Verfügbar unter ftp://ftp.cs.cmu.edu/usr/ftp/usr/sleator/splaying/Top-Down-splay.c, Zugriff am 17.02.2013, Carnegie Mellon University, Pittsburgh, PA, USA, 1992

9 Daniel Dominic Sleator and Robert Endre Tarjan. *Self-adjusting binary search trees.* Journal of the ACM Vol.32 (S. 652-686), AT&T Bell Laboratories, Murray Hill, NJ, USA, 1985

10 Hugh E. Williams, Justin Zobel and Steffen Heinz. *Self-adjusting trees in practice for large text collections.* RMIT University, Melbourne, Australia, 2002

11 George K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Cambridge, MA, USA, 1949

12 *djb2 hash function.* Verfügbar unter http://www.cse.yorku.ca/~oz/hash.html, Zugriff am 17.02.2013

Connectivity in Fully Dynamic Graphs*

Marcel Radermacher[1]

1 **Karlsruhe Instutute of Technology**
marcel.radermacher@student.kit.edu

Abstract

In this survey an efficient solution to the connectivity problem in fully dynamic graphs, introduced by [7], is presented. Euler-Tour trees are used to maintain a spanning forest. With a hierarchically decomposed graph and spanning forest it is possible to accomplish amortized $O(\lg^2 n)$ update time and $O(\lg n / \lg \lg n)$ query time.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases fully dynamic graphs, connectivity, Euler-tour trees

Digital Object Identifier 10.4230/LIPIcs.9999.9999.9

1 Introduction

A simple approach to check whether two vertices are connected is to calculate a shortest path between both vertices via a breath first search. This takes $O(n+m)$ time for queries. If many queries are triggered on static graph it is simple to find a faster solution. In an undirected graph the only possibility for two nodes to be connected is to be in the same connected component. To answer a query it is sufficient to calculate all connected components in advance and to manage them with the union-find data structure. One query can be answered within amortized $O(\alpha(n,m))$ time, where $\alpha(n,m)$ is the inverse Ackermann function [5].

We adapt this idea to handle fully dynamic graphs. Instead of the union-find data structure we maintain a spanning forest. Therefore, two vertices are connected if they are contained in the same spanning tree. Inserting a new node means to create a new spanning tree. To insert an edge, we might haves to merge two spanning trees. Deleting an edge is not that simple, if the edge is part of a spanning tree, we have to figure out if there exists a replacement for this edge which keeps the spanning tree connected. Deleting a node is then simple again, delete all incident edges and remove the resulting spanning tree.

We subdivide this problem into two problems. We use *Euler-Tour trees* to manage one spanning tree. With those we are able to answer queries efficiently. Further on, we will see how to maintain the spanning forest so that fully dynamic graphs can be handled in an efficient manner. On this way we can achieve $O(\lg^2 n)$ amortized time for updates and $O(\lg n / \lg \lg n)$ time for queries [7].

1.1 Preliminaries

For an *undirected graph* $G = (V, E)$ with n nodes and m edges a *path* $v \rightsquigarrow w$ is sequence of nodes $\langle v = v_1, v_2, \ldots, v_p = w \rangle$ so that each pair of nodes $\{v_i, v_{i+1}\}$ is an edge in G where $i \in \{1, 2, \cdots, p-1\}$. An edge weight $c : E \to \mathbb{R}$ maps an edge to an arbitrary number. Two nodes $v, w \in V$ are *connected*, if there exists a $v \rightsquigarrow w$ path. A *connected component* in G is a maximum set of nodes C so that every two nodes c_1, c_2 in C are connected.

* Based on Lecture by E. Demaine[3]

1st Symposium on Breakthroughs in Advanced Data Structures (BADS'13).
Editor: J. Fischer; pp. 53–61

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Definition 1.** The *(fully) dynamic connectivity* problem is specified as follows. Maintain an undirected graph $G = (V, E)$ with the following operations.

insert edges or nodes into the graph

delete edges or nodes from the graph

connectivity(v, w) is there a $v \rightsquigarrow w$ path?

We refer to the problem as *fully dynamic* graph problem, if both insert and delete operations are supported. If only one of both operation is implemented, the problem is called *partially dynamic*. Especially, if the insert operation is the only available operation the problem is *incremental*. The problem is called *decremental* if the delete operation is provided only.

For the concept of Euler-Tour trees in section 2.1 it is necessary to split a tree T. The split operation for a node $v \in T$ splits T into two parts T_B and T_A, so that for all elements x in T_B applies $x < v$ and for all elements $y \in T_A$ $y \geq v$ is valid.

A *spanning tree* of a Graph G is a tree which connects all nodes in G. A *spanning forest* is a set of trees with a spanning tree for each connected component. A *minimum spanning tree (forest)* is a spanning tree (forest) that minimizes the sum over all edge weights in this tree with respect to a given edge-weight function.

1.2 Related Work

Connectivity in decremental trees can be solved in amortized constant time, e.g. constant time for update and query [1]. For plane graphs, e.g. graphs with a planar embedding, $O(\lg n)$ time is possible [4]. For general graphs updates can be implemented faster than presented in this paper on account of the query time. With $O(\lg n(\lg \lg n)^3)$ time per update operation the query slows down to $O(\lg n / \lg \lg \lg n)$ time [9]. Demaine et al. [8] proved for general graphs that if the update operation takes $O(x \lg n)$ time for $x > 1$, the query needs at least $\Omega(\lg n / \lg x)$ time. The same result can be shown if both operations are exchanged.

2 Connectivity in Fully Dynamic Graphs

To solve the connectivity problem in fully dynamic graphs we are using the concept of Euler-Tour trees introduced by Henzinger et al. [6] in 1995. Roughly, one Euler-Tour tree is going to maintain one connected component. With a *hierarchically decomposition* of the graph and these Euler-Tour trees we are able to find a replacement edge for a deleted edge, if necessary. Holm et al. [7] are the first to explain this algorithm as solution for the fully dynamic connectivity problem in graphs. To maintain the complete structure of the graph we store an incidence list for each node.

2.1 Euler-Tour Trees

With Euler-Tour trees it is possible to avoid problems which occur with the union-find data structure. If a spanning-tree edge is deleted it is not clear how this does effect the union-find data structure. On the other hand, storing the spanning tree as a tree in the way it was calculated, is not an option either. Recall, we would like to find out if two nodes are in the same spanning tree e.g. both nodes have the same root node. In a naive stored spanning tree this could lead to $O(n)$ worst-case look up time. With *Euler-Tour trees* we can achieve $O(\lg n)$ time for all operations.

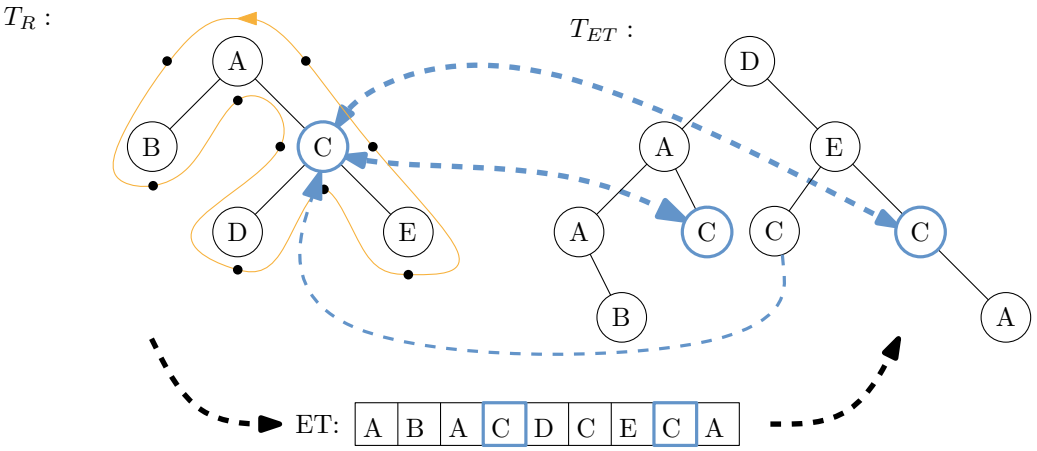


Figure 1 Example for Euler-Tour-Trees.

▶ **Definition 2.** An Euler-Tour (ET) in an (un-)directed graph is a path which contains every edge of the graph exactly once.

In general an ET does not exist in a tree. Therefore, we allow every edge to be traversed two times.

▶ **Definition 3.** An Euler-Tour tree T_{ET} is a balanced binary search tree (BBST) which stores all visits of node of ET of a represented tree T_R. Nodes in T_{ET} are stored in the order in which they occur in ET. Each node v in T_R stores pointers to the nodes in T_{ET} which represent the first an last visit of v in ET. Each node w in T_{ET} stores a pointer to the node in T_R which it represents.

An example is depicted in figure 1. The represented tree T_R is shown on the left side. The orange path visualizes the Euler-Tour of this tree which leads to the node order ET on the bottom. The node visits are then inserted into the BBST T_{ET} in this particular order. For each node in T_R two pointers into T_{ET} are stored. For example, node C holds pointers to the nodes in T_{ET} which represent the first and last visit in ET. Later on the T_{ET} is the result of an arbitrary sequence of *join* and *cut* operations. To get an initial Euler-Tour tree representation of a given Graph G, each node represents its own Euler-Tour tree. Then for each edge the *join* operation is called. On this way, it is not necessary to store ET to build T_{ET}, it is only depicted for clarity. An in-order-traversal of T_{ET} delivers ET. This leads directly to an important property. The root of T_R is always the left most node in T_{ET}.

2.1.1 Findroot

With this property we are able to define our first operation $findroot(v)$ for a node v of T_R. Jump to the first visit v_f of v in T_{ET}. From v_f go up to the root of T_{ET} and then follow the left most path to the node r_f. This node represents the first visit of the root r of T_R. Therefore, jump to r and return this node. The tree T_{ET} is balanced, therefore, this operation takes $O(\lg n)$ time.

2.1.2 Cut

The operation $cut(v)$ cuts the subtree rooted at v of T_R. The subtree of v is a subsequent sequel in ET between the first and last visit of v. With this, we only have to split T_{ET} at

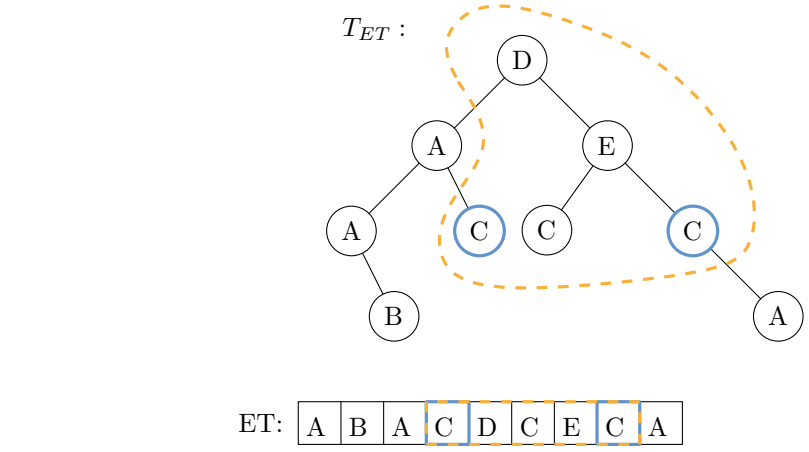


Figure 2 Cut-operation in T_{ET} for the subtree of T_R rooted in C.

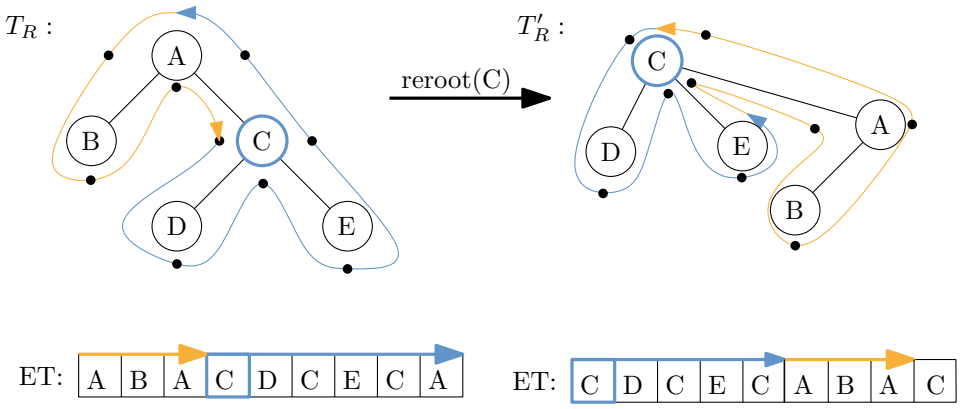


Figure 3 Reroot T_R at the node C

the first and last occurrence of v. This results in three subtrees, one tree T_B for nodes which occur before the first visit of v, one tree T_A for nodes after the last visit and one tree for in between. The concatenation of T_B and T_A results in the new Euler-Tour tree. The parent node w of v is visited one time less then before, therefore one occurrence in T_{ET} has to be deleted. All necessary operations on the BBST can be implemented in $O(\lg n)$ time, this results in $O(\lg n)$ time for the *cut* operation. The relation between this trees is depicted in figure 2.

2.1.3 Join

We need to be able to join two trees, we do so with the $join(v, w)$ operation. This operation adds the tree T_v rooted in v to the children of w. The order of the children of w does not matter, therefore we insert v as the last child of w. All we have to do is to split T_{ET} at the last occurrence of w into two trees T_B and T_A (before and after w). Then concatenate T_B, T_v and T_A in this order, with one extra node for w, as the number of visits of w increases by one. Again, this can be realised in $O(\lg n)$ time in a BBST.

2.1.4 Reroot

Later on we would like to join two not connected nodes v, w, e.g. both nodes are in different spanning trees. Every node in the spanning tree of v might be the root. For the join operation to work, we assumed that v is the root. If v is not the root of its spanning tree we can *reroot* the tree in $O(\lg n)$ time as well. Rerooting in terms of the Euler-Tour means a cyclic shift in a matter that ET starts with first occurrence of v. Figure 3 gives an impression how the reroot operation affects the Euler-Tour. This cyclic shift can be realised with a split operation at the first occurrence of v in T_{ET}. Concatenating the resulting two trees in the opposite order leads almost to the correct representation of the new Euler-Tour. After rerooting the tree v is visited one time more and the old root one time less. Accordingly, one occurrence of v has to be added and one occurrence of the old root has to be deleted.

2.1.5 Subtree Aggregation

A main advantage of Euler-Tour tree is that we can efficiently aggregate over the subtree of a node v. With the BBST this operation can be done by a range query between the first and last occurrence of v in $O(\lg n)$ time.

2.1.6 Faster Queries on Account of Updates

We can even speed up our connectivity query on account of the update time. If we use B-trees [2] with a $\Theta(\lg n)$ branching factor instead of binary search trees. The height of a B-tree is in $O(\log_d n)$ where d is the branching factor. This means, we can realise $findroot$ in $O(\lg n / \lg \lg n)$ time. With B-trees Euler-Tour tree updates can be implemented in $O(d \cdot h)$ time where h is the height of the tree. Therefore, this operations need $O(\lg^2 n / \lg \lg n)$ time, now.

2.2 Fully Dynamic Graphs

Holm et al. [7] introduced the following method to maintain a spanning forest to solve the connectivity problem in fully dynamic graphs. With the operations described in 2.1, we are able to join two trees and to separate a subtree from a large tree. We can use this to handle connectivity in fully dynamic graphs. The main idea is to maintain a Euler-Tour tree for each connected component in the spanning forest. Then, the connectivity query is simple, two nodes are connected if they both have the same root. On an insertion of an edge (v, w), the spanning forest changes only if v and w are not connected. In this case, we have to join both spanning trees. The deletion of an edge (v, w) is more complicated. We have to consider two cases. In the first case, the edge is not part of any spanning tree. Then we can simply delete this edge and no spanning tree is affected. If the edge is part of a spanning tree, we have figure out if v and w are still connected. Therefore, we have to find an edge which joins the subtrees T_v and T_w after extracting them from its mutual spanning tree. Of course a complete minimum spanning tree algorithm is to expensive to find this substitute edge. To solve this problem, we hierarchically decompose the graph and the spanning forest into $\lg n$ level. E.g. we associate an edge with a level. The level of an edge starts with the value $\lg n$ and decreases monotone until it reaches zero. The Graph $G_i = (V, \{e \in E \mid e.level \leq i\})$ is the subgraph of G with all edges of level less or equal to i. For each level maintain a spanning forest F_i of the graph G_i using Euler-Tour trees. We would like to achieve that an edge can only pushed down $\lg n$ times before it has to be deleted. The basic idea behind the decomposition is that a push down operation in a sense pays for a search operation, so

Listing 1 inset($e = (v, w)$)

```
add e incidence list of v and w
e.level = lg n
if ( v,w are not connected in F_{lg n}) {
    join Euler-Tour trees of v,w in F_{lg n}
}
```

that overall an edge can not produces costs more then $O(\lg^2 n)$ time. To achieve this, all operations have to respect the following two invariants.

Invariant 1 every connected component of G_i has at most 2^i vertices.
Invariant 2 $F_0 \subseteq F_1 \subseteq \cdots \subseteq F_{\lg n}$

From the second invariant, we can conclude that F_i equals to the intersection of $F_{\lg n}$ and G_i. We can also conclude that $F_{\lg n}$ is a minimum spanning forest, e.g. each spanning tree is minimal w.r.t. to the level. E.g. we define the weight of the edge equal to the level of the edge. For completeness, algorithm 1 shows the insert operation on this data structure. It is obvious that $F_{\lg n}$ represents the spanning forest of G and can be used to answer connectivity queries.

2.2.1 Delete

The delete operation on an edge $e = (v, w)$ works as follows. Delete e from the incidence list of v and w. If e is not element of $F_{\lg n}$ it is not part of any spanning forest F_i due to invariant two and nothing further has to be done. If it is element of the top spanning forest, we delete e from all spanning forests it is contained in, e.g. $F_{e.level}, \ldots, F_{\lg n}$. At this point, we have to find a replacement edge for e, if there is any. If there exists a replacement edge, it has to be above the level of e. If there would be a replacement edge in a level below, this would corrupt invariant two and the MST property of F_i. Therefore, we only have to search for a replacement edge in the levels $e.level$ to $\lg n$. We are looking for the smallest level in which such an edge exists. Let T_v and T_w denote the trees of v, w in F_i in some level i. Assume that $|T_v| \leq |T_w|$. From invariant one we can conclude that $|T_v| + |T_w| \leq 2^i$ and therefore $|T_v| \leq 2^{i-1}$. As we already know, no replacement edge exists for any edge in T_v. Therefore, we could afford to push all of them one level down without corrupting invariant one.

The search operation for a replacement edge works then as follows. Look at all edges $e = (x, y)$ at level i where x is a element of T_v. If y is part of T_w, we have found a replacement edge and can insert it into the Euler-Tour tree F_i. In this case we are done. If y is element of T_v, we push this edge one level down. The complete delete operation is depicted in Algorithm 2.

Figure 4 depicts a small example of the delete operation. The delete operation on the blue edge $e = (B, D)$ is called. Therefore, the edge is deleted from the incidence list of both nodes in level three and two. The spanning tree on level two is cut into two trees T_v and T_w where $v = B$ and $w = D$ (in this case the order is arbitrary). There are only two incident edges for node B in level two. Let say, the edge (B, E) is looked at first. This edge does not join T_v and T_w and is therefore pushed down to level one. The next edge actually joins both trees and completes this delete operation.

There are some augmentations on the Euler-Tour tree necessary. First of all, all nodes store subtree sizes. A second augmentation efficiently finds nodes with incident level-i edges. The tree T_v can be traversed in-order and a node can be skipped, if there are no nodes in

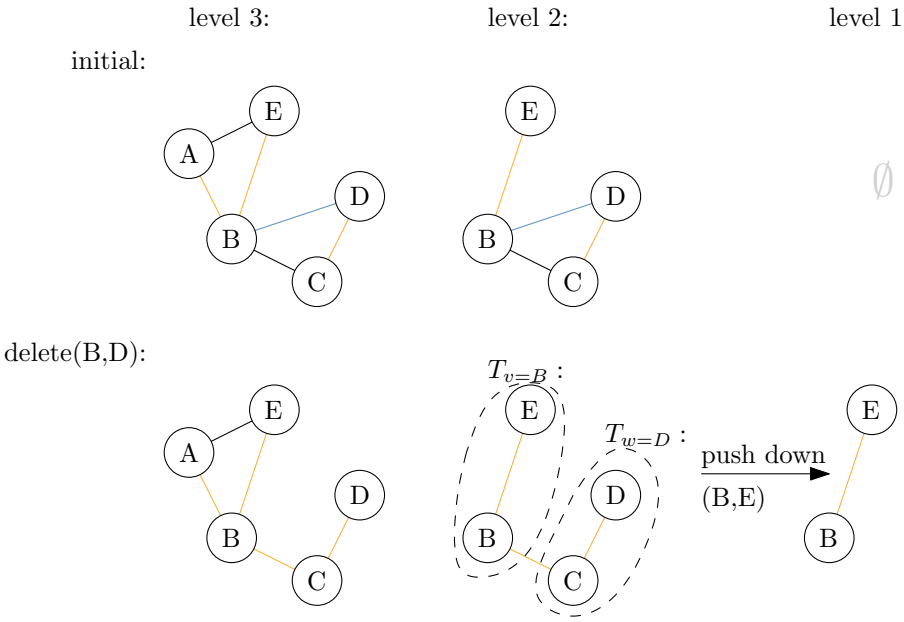


Figure 4 An Example how the delete operation works. The first row depicts the initial state, the second the state after $delete(B, D)$ is called. The blue edge is going to be deleted. The orange edges with the blue one represent the current minimum spanning tree.

the subtree with an incident level-i edge, this necessary for the inner for-loop in algorithm 2. With an incidence list for each level, all level-i edges of a node can be reported.

2.2.2 Analysis

As we have seen in 2.1.6 we can speed up our queries if we store the Euler-Tour in a B-Tree. The queries use the top-level spanning forest only. Therefore, it is sufficient to store $F_{\lg n}$ as a B-Tree based Euler-Tour tree. Then the overall time for the update operations do not decline. The insert operation has one look up in $F_{\lg n}$ and at most one insert operation in this tree. Therefore we need $O(\lg^2 / \lg \lg n) \subset O(\lg^2 n)$ time for the insert operation. The connectivity query uses the findroot operation on $F_{\lg n}$ which can be done in $O(\lg n / \lg \lg n)$ time.

Essentially, the delete operation consists of two parts. First, the edge e has to be deleted in all forests it is contained in. There are at most $\lg n$ forests. The delete operation on the top-layer forest costs roughly $O(\lg^2 n)$ time and for all lower level forests the delete operation can be charged with $O(\lg n)$ time. Therefore this step can be realised in $O(\lg^2 n)$ time. A push down operation is basically a join operation on a lower level Euler-Tour tree and costs $O(\lg n)$ time. With the subtree aggregation on Euler-Tour trees the next level-i edge can be found in $O(\lg n)$ time. Then an edge is either pushed down one level or the edge is used to join T_v and T_w. Overall, an edge can only be decremented $\lg n$ times. Therefore, the amortized time for the delete operation is $O(\lg^2 n)$ time.

3 Conclusion

Maintaining (minimum) spanning trees as Euler-Tour trees is a powerful instrument. Holm et al. [7] showed how Euler-Tour trees can be used to handle connectivity queries in fully

Listing 2 delete($e = (v, w)$)

```
remove e from incidence list of v and w
if ( v and w are in different connected component in F_{lg n} ) return
delete e from F_{e.level}, ..., F_{lg n} e.g. a cut operation
for ( i = e.level to F_{lg n}) {
    let T_v, T_w be trees of F_i where v ∈ T_v and w ∈ T_w
    relabel v, w so that |T_v| ≤ |T_w|
    foreach( edge e' = (x, y) in level i where x ∈ T_v){
        if (y ∈ T_w) {
            insert e' into F_i, e.g. join T_v and T_w
            return
        } else {
            push down e
        }
    }
}
```

dynamic graphs efficiently. Different Euler-Tour tree data structures e.g. balanced binary search trees and B-trees allow us to speed up connectivity queries to $O(\lg n / \lg \lg n)$ time without slowing down the update operations. For the insert and connectivity query there is no big conceptual difference to the union-find datastructure. The main challenge to fully dynamic graphs is the find to a replacement edge for a deleted edge. The hierarchally decomposition of the graph and the spanning forest provides a solution with amortized $O(\lg^2 n)$ time.

3.1 Open Problems

It is not known if $O(\lg n)$ amortized time can be achieved for updates and queries. An open problem as well is whether polylog worst-case time for updates and queries is possible or not. It even remains an open problem if we can achieve $o(\lg n)$ updates and polylog queries.

References

1 Stephen Alstrup, Jens P. Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Inf. Process. Lett.*, 64(4):161–164, 1997.

2 Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

3 Eric Demaine. 6.851: Advanced datastructures (spring'12), lecture 20, dynamic graphs. University Lecture, Spring 2012.

4 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification for dynamic planar graph algorithms. In *Proc. 25th Symp. Theory of Computing*, pages 208–217. ACM, May 1993.

5 Shimon Even and Robert E. Tarjan. Network Flow and Testing Graph Connectivity. *SIAM Journal on Computing*, 4(4):507–518, December 1975.

6 Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *STOC*, pages 519–527, 1995.

7 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *Journal of the ACM*, 48(4):723–760, July 2001.

8 Mihai Patrascu and Erik D. Demaine. Lower bounds for dynamic connectivity. In *STOC*, pages 546–553, 2004.

9 Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 343–350, New York, NY, USA, 2000. ACM.

Link-Cut Trees

Alexander Sebastian Weigl[1]

1 **KIT, Institut für Theoretische Informatik, 76131 Karlsruhe, Germany**
Alexander.Weigl@student.kit.edu

Abstract

Link-cut trees is a data structure for maintaining a forest of rooted trees. The data structure allows an efficiently join (*link*) of two trees and disjoint (*cut*) of a subtree. Additionaly a cost function can be maintain during both operations. There are no constraints of the morph of the trees. We introduce the data structure and show two approaches by using a path decomposition into solid and dashed edges defined vertex access. The solid edges can be represented with list or splay trees. Within the runtime analysis we use the concept of solid paths and heavy-light decomposition to an amortized running time $\mathcal{O}(m \log n)$ with m operation and n tree nodes.

1998 ACM Subject Classification E.1 Trees

Keywords and phrases link-cut trees, forest, data structure, splay trees, preferred child decomposition, heavy-light decomposition, trees as set of paths

Digital Object Identifier 10.4230/LIPIcs.9999.9999.9

1 Introduction

With link-cut trees we can maintain a forest of trees. The trees are rooted, unordered and the branching factor is not limited. The trees are managed by the the link-cut data structure in definition 1. A direct editing of a tree is prohibited. The only constraint is the disjoint of the vertices between the trees. For example link-cut trees are used in network algorithms for finding various kinds of network flows after [2].

Normally trees are represented as directed-acyclic graph, there the edges shows from the parent to his children. In our case the edge direction is reverse. The edge shows from the child to his parent (fig. 1). We define the operation on the forest as follows ([4, 1]):

▸ **Definition 1** (Functions for link-cut trees)**.**

- maketree(v): creates a new tree, that contains only the vertex v.
- link(v, w): connect the tree containing v and w by adding the edge (v, w). v is a root and the trees are vertex disjoint.
- cut(v): splits up the tree containing v above by deleting the edge from v to its parent.
- path-aggregate(v): applies an function (e.g. sum, min, max) to all elements on the path from v to the root of v.

Additionally [4, 2] defines a function for manipulating edge or vertex costs. We follow [1] and added path-aggregate(v) to the set of functions. This can be any function application on the whole path that contains v like minimizing, maximizing or sum the costs of the path from v to his root. We do not take care of maintaining the costs during link and cut. Details are in [2].

Tree operations normally include insert, deleting and finding a vertex (binary search trees). At link-cut trees we join two whole trees. The morph of these both trees are not allowed to be changed like re-balancing.

1st Symposium on Breakthroughs in Advanced Data Structures (BADS'13).
Editor: J. Fischer; pp. 62–71

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

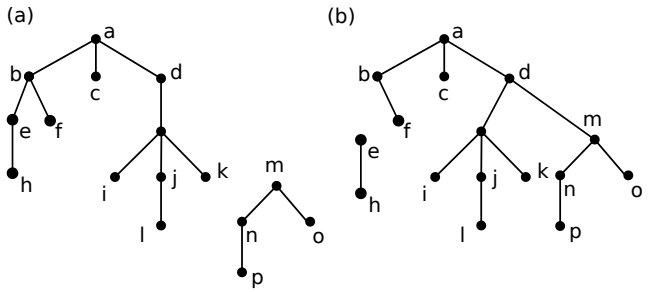


Figure 1 Tree operations. (a) Two trees. Notice the edge direction to the parent. (b) is after link(m, g) and cut(e). Figure after [4]

A simple approach is to store for every vertex v its parent $parent(v)$. In this simple link-cut-tree we can link, cut and maketree in $\mathcal{O}(1)$ by adding or removing edges in $parent(\cdot)$. The edges are given directly as arguments. But the path-aggregate$(\cdot)$ depends on the length from v to its root – in worst-case $\mathcal{O}(n)$ (see [4, 2]). We improve path-aggregate$(\cdot)$ to $\mathcal{O}(m \log n)$[1] and get a worse run time for link and cut.

The main idea is the partitioning of the paths within the trees. These paths can be handle like list (section 2) or by trees (section 3). For both representation we do a estimation of running time.

References This article is based on [2], [4] from R. Tarjan and the lecture 19 by E. Demaine hold in spring term 2012 at MIT [1]. All three sources have a differences, for example costs are bound in [2] to edges, in [4] in vertices and in [1] there are out of scope. Our goal is an easy to understand fusion result with focus on the decomposition. We abandon the costs from trees and refer to [2].

2 Solid paths as list

We partitioned a tree into solid paths (fig. 2) determined by access of the vertices. Then we can map the link-cut tree with his operations to a set of paths and operations on the list and the set. We realize the link-cut tree by defining algorithms above a set of paths. In section 3 we exchange this data structure for variant with splayed trees.

▸ **Definition 2** (Data structure *PathSet*)**.** The data structure *PathSet* is a set of paths P with the following operations for $p := (p_1, \ldots, p_n)$ and $q := (q_1, \ldots, q_m)$ paths with $m, n \geqslant 0$ and v a vertex.

- makepath$(v) =_{\text{def}} (v)$
- tail$(p) =_{\text{def}} p_n$
- join$(p, v, q) =_{\text{def}} (p_1, \ldots, p_n, v, q_1, \ldots, q_m)$
- split$(v) =_{\text{def}} (\underbrace{a_1, \ldots, a_n}_{p}, v, \underbrace{w_1, \ldots, w_m}_{q})$
- findpath$(v) =_{\text{def}} (a_1, \ldots, a_i, v, a_{i+2}, \ldots, a_n)$

With join(p, v, q) we glue the p, v and q to a new path. split(v) destroys paths and returns p, q without the vertex v. findpath(v) is to find the path for a vertex v. Operation join is

[1] m denotes the amount of operations including n maketree-calls

the dual to split: $\text{join}(\text{split}(v)_p, v, \text{split}(v)_q) = \text{findpath}(v)$. split does a findpath internally. In each operation the set of paths P is maintained, e.g. the splitted path is removed and p, q and (v) are added to P. A more concrete view of the data structure is in appendix A.

The solid paths are determined by tree operations done so far. [1] calls this *preferred paths*, build up by preferred child. The preferred child w (incident solid edge) of a vertex v is

- *none* if we access v itself
- else w iff. the access a vertex in w's subtree.

The head of a path is bottommost, the tail is the topmost vertex. The decomposition change every time we access two different vertices. We call the edge between the *preferred child* and his parent solid. The other edges are called dashed.

We connect the solid paths together via $pathparent(t)$[2] (dashed edges). This denotes to a vertex in the upper solid path. $pathparent(t)$ is *null* if t contains the root of the tree.

The preferences are change by the expose-operation[3]. The operation expose(v) transfer the way from vertex v to the root of v into a solid path. This means, that each *pathparent*-link (dashed lines) on the way becomes solid and other solid lines becomes dashed. The solid incident edge of v becomes dashed. After the application of expose(v) the path of v starts at v and ends at the $root(v)$.

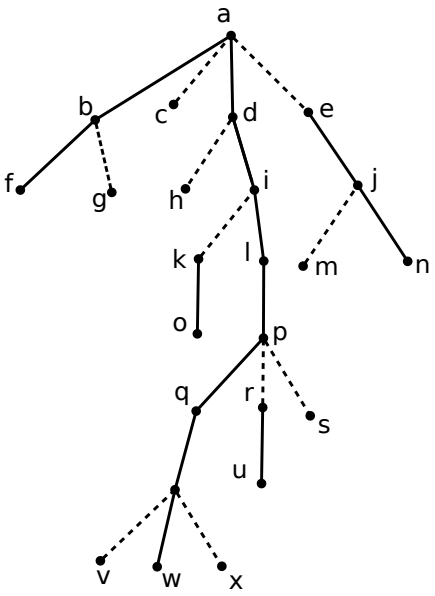


Figure 2 A tree partitioned into solid paths (t, q, p, l, i, d) has head t and tail d [4]

The algorithms 1 to 4 are from [4] with slightly modifications. The maketree operation (alg. 1) is simple. A new tree with a node v is just the solid path with v and with no $pathparent(v)$. link(v, w) uses join to create a new solid path together by gluing the solid path of v and w together. The operation assumes that v and w are vertex disjoint. In cut(v) (alg. 3) we generate solid path with the path findpath(v) and split up the path at v. The two parts p, q are added to the set of paths. The most complicated algorithm is expose(v)

[2] [4]: *successor*, [2]: *dparent*.
[3] In [1] as *access*(v)

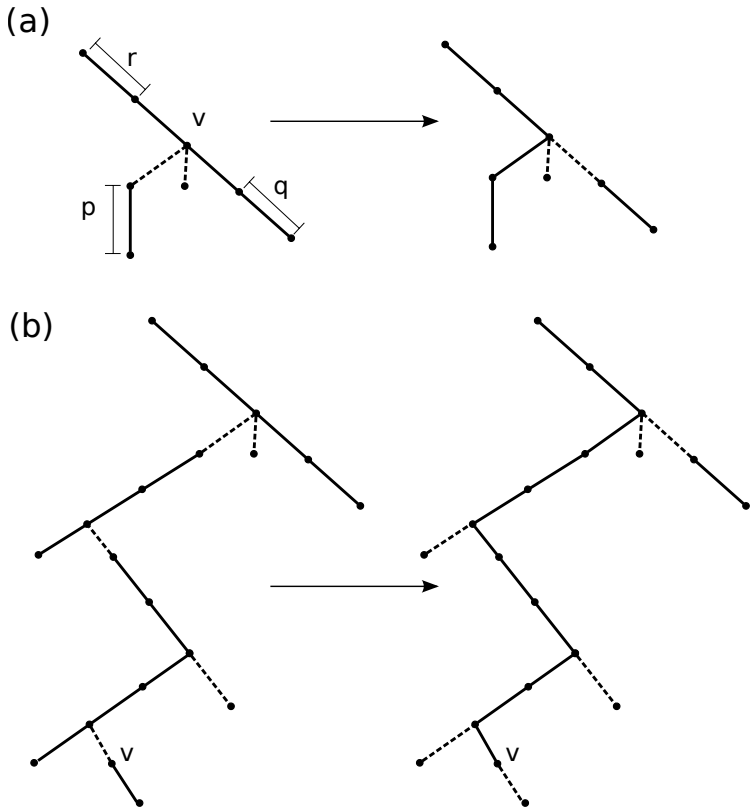


Figure 3 Expose operation of one dashed edge and in a tree [4]

(alg. 4). We start at v and $pathparent(v)$, if v is within a solid then q is not *null*. Then the incident edge to v from q is set to dashed. In the next step we glue the solid path p below v to the new lower solid path together and repeat this for the $pathparent(v)$. The join repairs the split operation. Each iteration in expose is called a *splice*.

Algorithm 1: maketree(v)

```
1 begin
2 |  p := makepath(v);
3 |  pathparent(v) := null;
4 |  return p;
5 end
```

Algorithm 2: link(v, w)

```
1 begin
2 |  pathparent(join((), expose(v), expose(w))) := null;
3 end
```

Algorithm 3: cut(v)

```
1 begin
2 |  expose(v);
3 |  (p, q) := split(v);
4 |  pathparent(v) := pathparent(q) := null;
5 end
```

runtime estimation Each algorithm tree operation takes $\mathcal{O}(1)$ path operations and at most two expose operation. The *splice* takes $\mathcal{O}(1)$ path operations. We only need to count the number of *splice* operations.

▸ **Theorem 3.** *For a sequence of m tree operation including n* maketree *operations*
- *we need $\mathcal{O}(m)$ path operations*
- *and $\mathcal{O}(m \log n)$* splices *(with $\mathcal{O}(1)$ path operations)*

The first part of theorem 3 is clear for the second part we need to analyze the expose operation. We need the *heavy-light decomposition* for estimation:

▸ **Definition 4** (heavy-light decomposition)**.** An edge e outgoing from vertex v is *heavy* iff. $size(v) > \frac{1}{2} size(parent(v))$ and *light* otherwise.

With $size(v)$ we refer to the number of descendant vertex of v including v.

▸ **Lemma 5.** *For any vertex v is at most one* heavy *incident edge to v and at most $\lfloor \lg n \rfloor$* light *edges to root(v) with n vertices on the whole tree.*

There can be only one vertex that have more than 50 % of the size of his parent. The second results that the size is halved in each level. We can differ between heavy or light and solid or dashed.

Algorithm 4: expose(v)

```
1  begin
2      path p, q, r; vertex w;
3      v' := v;
4      p := null;
5      while v ≠ null do
6          w = pathparent(findpath(v));
7          (q, r) := split(v);
8          if q ≠ null then
9              pathparent(q) := v;
10         end
11         p := join(p, v, r);
12         v := w;
13     end
14     pathparent(p) := null;
15     return v';
16 end
```

▸ **Lemma 6.** *Each* expose, link *and* cut *requires* $\mathcal{O}(\lg n)$ *amortized changes of preferred childs.*

Proof. We investigate the number of changes solid to dashed in expose($\cdot$), link($\cdot$) and cut($\cdot$) for a sequence of m operation. A change of a solid edge results in a destruction and creation. Additionally we can divide the solid edge change with the heavy-light decomposition. So we can describe the number of changes as:

$$\#changes \leqslant \text{light solid edges creation} \tag{1}$$
$$+ \text{ heavy solid edges destruction} \tag{2}$$

If we change an edge it is a creation and destruction. For light edges the creation is easier to estimate, for heavy edge the destruction.

expose(v) convert at most $\lfloor \lg n \rfloor$ light dashed edges from v to $root(v)$ into solid (lemma). If we destroy a heavy solid edge, then we can only created solid light edge. At the beginning of expose we destroy incident solid edge into v, so we can summarize that we have $\log n + 1 \in \mathcal{O}(\log n)$ changes.

link(v, u) The path from $root(u)$ to v can be become heavy. That destroys heavy edges along the path, but they are not solid. We have no change.

cut(v) Nodes on the path $root(v)$ to v becomes lighter and so light solid edges can be created, but at most $\lg n$. The destroyed edge $(v, parent(v))$ can be heavy and destroyed. cut(v) has at most $\mathcal{O}(\lg n)$ changes.

◂

3 Solid path as tree

In previous section we show an implementation for link-cut tree with $\mathcal{O}(m \log n) \cdot T_{\text{Path}}$ amortized cost with T_{Path} costs per path operation. We achieve a better result by representing

the solid paths with balanced trees (fig 4). Within *PathSet* the set of paths is replaced with a set of balanced splay trees T'. We call these trees solid the trees in the link-cut trees are called actual[4].

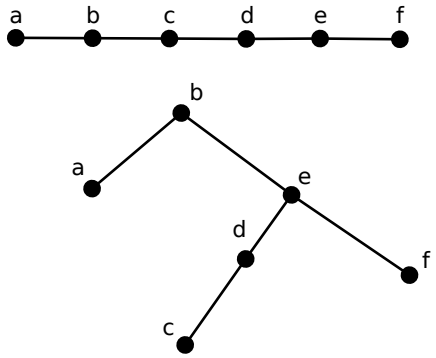


Figure 4 A solid path represented by a solid tree [4].

The splay tree offers the splay operation. The call splay(x) let become x the root of the tree by multiple rotations. The splay operation keeps the order of the nodes at inorder traversing. The operations upon splay are $\mathcal{O}(\log n)$ amortized [3]. Intuitively we can implement link-cut trees in $\mathcal{O}(m \log^2 n)$ (theorem 3). This seems more worse, but we can assume that $T_{\text{Path}} = \mathcal{O}(n)$. With a finer look on the splay operation we get a better runtime estimation.

The figure 5 (a) is the actual tree, that is represented by solid tree (b). The root v of a solid tree has the *pathparent*(v) pointer to the vertex in the upper solid tree. We get forest of the solid trees for one tree in the link-cut structure.

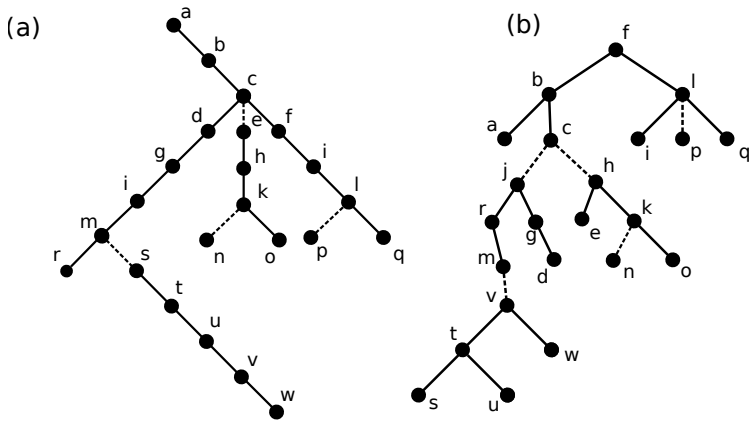


Figure 5 (a) The link-cut tree (b) One possible morph of the solid trees in (a), the dashed line is the *pathparent* pointer.

The result operation of *PathSet* are the sames as in definition 2, but defined upon trees. The operations makepath(v) and findpath(t) are trivial. Additionally findpath(v) call splay(v) to make v the root of his solid tree. tail(v) (topmost vertex of the solid path) is rightest node in this tree (splay preserves the inorder). The join(p, v, q) merges the solid tree

[4] Demaine refers with auxillary tree to solid trees

p, q together via the one-node tree v. The vertex v becomes the root of p (left child) and q (right child). The split(v) splays v and return his left and right child.

We start with the runtime analysis.

▸ **Theorem 7.** *For a sequence of m tree operation including n* maketree *operations and using solid trees we achieve $\mathcal{O}(m \log n)$ time.*

We use the proof from [1]. This very similar to [4]. Let $W(v)$ the size of v's solid subtree. This is the same as:

$$W(v) := |\Delta_v| + \sum_{u=pathparent(w), w \in \Delta_v} size(u)$$

with Δ_v the set of nodes in v's subtree. Notice that the second term does not change during the splay-operation.

We can declare with the access lemma of splay trees [3, 4] and $rank(v) = \log W(v)$ as the splay potential:

$$3(rank^{\text{after}}(u) - rank^{\text{before}}(v)) + 1$$

there u is the root of v's solid tree. We need that to show $0 \leqslant rank(v) \leqslant \lg n$, for any vertex v. We achieve $\mathcal{O}(\log n)$ credits[5] for any path operation.

Proof. We investigate our operation expose($\cdot$), link($\cdot$) and cut($\cdot$) for maintaining the access lemma:

expose This function is the center of the analysis. We show that a splice only need $\mathcal{O}(1)$ credits and $\mathcal{O}(\lg n)$ exposes. A typical splice starts with findpath(v). So v becomes the root of the solid tree via a splay(v). After this operation the split(v) and join(p, v, r) takes $\mathcal{O}(1)$. The number of $W(v)$ are not changed during these operations. So we pay one additional credit for split, join and the rest of the splice operation.
Over all splices we get $3(rank^{\text{after}}(u) - rank^{\text{before}}(v)) + 1$ plus two per splice.
We get $\mathcal{O}(\lg n) + \mathcal{O}(\#changes of solid/dashed)$. With lemma 6 we get a bound of $\mathcal{O}(m \log n)$.

link and cut The link(v, w) operation increase the weight $W(v)$, so we place $\mathcal{O}(\log n)$ additional credits on v. cut(v) decreases the size of v's ancestors and only releases credits.

◂

4 Remarks

Tarjan and Sleator present more complex solid trees in [2]. There a solid path is represented only by the external nodes in the solid tree. Our version of solid trees refers to [4].

The aspect of storing with the nodes cost are not discussed in this paper. With the simple representation in section 4 we introduce path operations like findcost or addpathcost for a solid path. With solid trees you should care about the cost during the splays [1]. Hints are in [4].

You can use the heavy-light decomposition for determining solid and dashed edges. This is represented in [2]. There the heavy edges becomes solid. Sleator and Tarjan showed a worst-case case upper bound $\mathcal{O}(\log n)$ time per operation.

[5] This comes the credit-debit accounting scheme used in splay tree runtime estimation

5 Conclusion

We defined the data structure for linking and cutting trees within a forest. With different backends of the data structure *PathSet* we achieved different runtime estimation. The best achieved was $\mathcal{O}(m \log n)$ amortized over a sequence of m operation (theorem 7). A version with $\mathcal{O}(\log n)$ worst case is shown in [2] with a different solid-dashed decomposition.

References

1 Erik Demaine. Dynamic graphs: link-cut trees, heavy-light decomposition. `http://courses.csail.mit.edu/6.851/spring12/lectures/L19.html`, Spring 2012.

2 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362 – 391, 1983.

3 Robert Endre Tarjan. 4 search trees. In *Data Structures and Network Algorithms*, Cbms-Nsf Regional Conference Series in Applied Mathematics, chapter 5, pages 59–70. Society for Industrial & Applied, Muray Hill, New Jersey, 1983.

4 Robert Endre Tarjan. 5 linking and cutting trees. In *Data Structures and Network Algorithms*, Cbms-Nsf Regional Conference Series in Applied Mathematics, chapter 5, pages 59–70. Society for Industrial & Applied, Muray Hill, New Jersey, 1983.

A Implemenation of 2

```
class LinkCutTree(object):
    def __init__(self):
        self.paths = PathSet()
        self.successor = dict()

    def maketree(self, id):
        p = self.paths.makepath(id)
        self.successor[p] = None

    def link(self, v, w):
        a,b = self._expose(v), self._expose(w)
        p = self.paths.join(b,None,a)
        self.successor[p] = None

    def cut(self, id):
        self._expose(id)
        p, q = self.paths.split(id)
        self.successor[p] = None
        self.successor[q] = None

    def _expose(self, v):
        exposed , p = v , None
        while v is not None:
            w = self.successor[self.paths.findpath(v)]
            [q, r] = self.paths.split(v)
            if q is not None: self.successor[q] = v
            p = self.paths.join(p, v, r)
            v = w
        self.successor[p] = None
        return self.paths.findpath(exposed)
```

```
class PathSet(object):
    def __init__(self):
        self._paths = set()

    def makepath(self, v):
        p = (v,)
        self._paths.add(p)
        return p

    def findpath(self, v):
        for p in self._paths:
            if v in p:  return p

    def join(self, p, v, q):
        if p is None: p = tuple()
        if q is None: q = tuple()

        if v is None:
            r = p+q
        else:
            r = p + (v,) + q

        for val in (p, (v,), q):
            self._paths.discard(val)

        self._paths.add(r)
        return r

    def split(self, v):
        r = self.findpath(v)
        i = r.index(v)
        t = lambda tpl: tpl if len(tpl) >= 1 else None
        self._paths.discard(r)

        p, q = tuple(r[:i]), tuple(r[i+1:])
        for a in (p , q, (v,)):
            self._paths.add(a)
        return t(p), t(q)
```