TU Dortmund
Lehrstuhl 11: Algorithm Engineering

Succinct Enhanced Suffix Arrays
Eine neue Repräsentation von RMQ-Informationen und Verbesserungen des Enhanced Suffix Arrays

Seminar: AE Datenstrukturen
Prof. Dr. Mutzel

von
Christian Scheffer
17.6.2009

1 Einleitung

Der *Suffix Tree* ist eine Datenstruktur, die effizientes Pattern Matching ermöglicht, aber den Nachteil eines hohen Speicherplatzverbrauches hat. Das Suffix Array hingegen bewältigt diese Aufgabe mit geringerem Platzbedarf. Dies führt nun zu einem erweiterten Suffix Array, dem *Enhanced Suffix Array* (ESA), mit dem eine Top-Down Travesierung des Suffix Trees simuliert werden kann. Damit diese Travesierung effizient durchgeführt werden kann, wird das Enhanced Suffix Array um eine weitere Tabelle erweitert, der *Child Tabelle*. Diese hat einen Speicherplatzverbrauch von $O\,(\mathrm{n\ log\ n})$. Im Folgenden wird das Suchproblem RMQ vorgestellt, mit deren Hilfe sich Anfragen an die Child Tabelle beantworten lassen.
Sei A ein Array der Länge n von einer total geordneten Menge. Ein *Range Minimum Querie* (RMQ) auf A, zwischen zwei Indizes $l, r \in \{0, ..., n-1\}$ ($RMQ_A(l, r)$) ist eine Anfrage nach der Position des minimalen Elementes bzgl. der totalen Ordnung in $A_{[l..r]}$. Eine triviale Lösung des Problems ist der lineare Durchlauf des betrachteten Bereiches. Im Paper hingegen werden nur Ansätze betrachtet, die abgesehen von einem vorhergehenden Pre-Processing nur konstante Anfragezeit benötigen. Im Fokus steht allerdings der hier für zusätzlich benötigte Speicherplatz. Im Folgenden werden zwei Ansätze zur Lösung des Problems unter diesen Anforderungen präsentiert: Es wird der Algorithmus von Berkman and Vishkin zitiert und ein neuer entworfen, wobei dieser die asymptotische Laufzeit des alten Algorithmus nicht verschlechtern soll. Der neue Algorithmus wird einen optimalen Platzverbrauch von $2n + o(n)$ haben. Somit wird also eine Speicherplatzverbesserung von $O\,(\mathrm{n\ log\ n})$ auf $2n + o(n)$ erreicht. Da diese Verbesserung sogar optimal ist, spricht man von dem *Succinct Enhanced Suffix Array*.
Die Grundidee beider vorgestellten Algorithmen, für das RMQ Problem, ist es das zu befragende Array in Blöcke zu unterteilen und dessen Minima vorzuberechnen. Die aktuelle Anfrage wird dann in Teile von zwei Typen unterteilt:

1. Vereinigung verschiedener Blöcke
2. Teilstücke von Blöcken

Durch eine konstante Zugriffszeit auf das Minimum eines Teilstückes eines Blockes, wird dann eine gesamte konstante Zugriffszeit erreicht. Der Kniff besteht also darin, dass Ergebnis des Pre-Processing so effizient wie möglich zu speichern, bzgl. des Speicherplatzverbrauchs. Für beide Algorithmen wird eine Speicherplatzanalyse durchgeführt. Anschließend wird bewiesen, dass der benötigte Speicherplatz des „neuen“Algorithmus asymptotisch optimal ist. Es werden dann verschiedene Anwendungen von RMQ aufgezählt, u.a. das *Enhanced Suffix Array*. Hier wird etwas genauer gezeigt, in wie weit der bentötigte Speicherplatz reduziert wird. Zum Schluss wird gezeigt, dass diese Reduzierung optimal ist.
Succincte Datenstrukturen sind Repräsentationen von (elementaren) Datenstrukturen (z.B. binäre Bäume, Bitvektoren usw.) deren Speicherbedarf nahe an der informationstheoretischen Schranke liegt und dennoch eine effiziente Ausführung bestimmter Operationen zulässt. Es wird also das *Succinct* Enanced Suffix Array betrachtet.

2 Definitionen

Sei A ein Array der Länge n und $l, r \in 0, ..., n-1$ Indizes, dann ist eine $RMQ_A\,(l, r)$ wie folgt definiert:

$$RMQ_A(l,r) := \arg\min_{k \in 0,\ldots,n-1} A[k].$$

Eine Hilfsdatenstruktur bzgl. des RMQ-Problems ist der *kartesische Baum*.

Definition 1. *Der kartesische Baum $C_{A[l,r]}$ eines Arrays $A[l,r]$ sei als ein binärer Baum definiert, dessen Wurzel das minimale Element $A[k]$ des Arrays ist. Die Wurzel wird mit dessen Index k annotiert ist. Der linke Teilbaum der Wurzel sei der kartesische Baum $C_{A[l,k-1]}$, falls $l \neq k$ ansonsten leer. Der rechte Teilbaum ist analog definiert.*

Es ist klar, dass für ein Array $A[l,r]$ mit $\exists l \leq k_1 < k_2 \leq k_2 r : A[k_1] = A[k_2]$ der kartesische Baum $C(A)$ i.A. nicht eindeutig definiert ist. Deswegen werden die Elemente des Arrays bzgl. folgender Ordnung verglichen:

$$A[k_1] \prec A[k-2] :\Longleftrightarrow A[k-1] \leq A[k-2] \vee A[k_1] = A[k_2] \wedge k_1 < k_2$$

Der bzgl. $\prec$ konstruierte karthesiche Baum $C^{can}(A)$ ist somit eindeutig und wird deswegen als *kanonisch* bezeichnet.

2.1 Algorithmische Anwendungen

Die Anwendung von RMQ auf ein Algorithmisches Problem A entspricht einer Reduktion des Problems A auf das Problem RMQ. Also wird im Folgendem das betrachtete Problem als eine RMQ Formulierung betrachtet.

2.2 Tiefster gemeinsamer Vorgänger der Knoten v und w in Bäumen T ($LCA_T(v,w)$)

Um in einem gegebenem statischen Baum T den tiefsten gemeinsamen Vorgänger zweier Knoten v und w zu bestimmen, wird zunächst eine in-order Travesierung vorgenommen. Hierbei werden die Höhen bzgl T der Knoten in ihrer besuchten Reihenfolge in H gespeichert. Darüber hinaus wird in $I[i]$ der i-te Knoten der Travesierung gespeichert. Dann definiere R als das inverse Array von I. Dann läßt sich das Problem des kleinsten gemeinsamen Vorgängers wie folgt umformulieren:

$$LCA_T(v,w) = I\left[RMQ_H\left(R(v), R(w)\right)\right]$$

2.2.1 Längster gemeinsamer Präfix zweier Suffix $t[i..n]$ und $t[j..n]$ eines Textes t ($LCE_t(i,j)$)

Gegeben sei eine beliebiger statischer Text t der Länge n und zwei Indizes $i, j \in \{1,\ldots,n\}$. Gesucht ist nun die Länge des längsten gemeinsamen Präfixes von $t[i..n]$ und $t[j..n]$. Das heißt also $LCE_t(i,j)$ ist wie folgt definiert:

$$LCE_t(i,j) = max\left\{k : t_{i,\ldots,i+k-1} = t_{j,\ldots,j+k-1}\right\}$$

Die Reduzierung auf das RMQ Problem sieht nun wie folgt aus:

$$LCE_t(i,j) = LCP\left[RMQ_{LCP}\left(SA^{-1}[i], SA^{-1}[j]\right)\right]$$

2.2.2 Dokumenten Wiederfindungs Anfragen

Gegeben seien n Dokumente. Das Problem besteht in der on-line Anfragebeantwortung aller Dokumente die ein bestimmtes Pattern beinhalten.

2.2.3 Indizes i und j eines Arrays A die größte Summe verursachen $MSSQ_A(l, r)$

Gegeben sei ein Array A der Größe n und zwei Indizes $l, r \in \{1, ..., n\}$. Gesucht sind nun zwei Indizes i und j mit $i, j \in \{l, ..., r\}$, die die Summe aller Einträge zwischen i und j maximieren, also:

$$(i, j) = \arg\max_{l \leq i < j \leq r} \sum_{k=l}^{r} (A[k])$$

Dann definiere das Array C als die Summe aller Elemente bis zum m-ten Element:

$$C[m] = \sum_{k=0}^{m} A[m]$$

Dann berechne x, y als die Indizes für die $C[x]$ bzw. $C[y]$ maximal bzw. minimal wird, wobei $x, y \in \{l, ...r\}$ gelten soll. Diese Berechnung läßt sich auf das RMQ Problem reduzieren. Nun ist $(x + 1, y)$ die Lösung von $MSSQ_A(l, r)$.

2.3 Berkman and Vishkins Algorithmus

Die Idee des Algorithmus von Berkman und Vishkin ist es zunächst das RMQ auf das stark verwandte $\pm 1 - RMQ$ zu reduzieren. Anschließend wird das reduzierte Problem gelöst.

Definition 2. *Das $\pm 1 - RMQ_H$ Problem entspricht dem RMQ_H, wobei aber benachbarte Elemente von H sich um maximal ein Element unterscheiden dürfen, d.h. also:*

$$\forall k \in 0, ..., n - 1 : H[k] \pm 1 = H[k + 1]$$

Als erstes wird das betrachtete Array A vorbereitet, indem dessen kartesischer Baum $C^{can}(A)$ berechnet wird. Auf diesem wird dann eine Euler-Tour durchgeführt. Bzgl. dieser Euler-Tour werden parallel drei Arrays generiert, die für den Algorithmus wichtige Informationen speichern:

- *Das Label l des i-ten Knotens der Euler-Tour*
  $E[i] = l :\Longleftrightarrow$ der i-te Knoten der Euler-Tour ist mit dem Label l gekennzeichnet.
- *Die Höhe h des i-ten besuchten Knotens*
  $H[i] = h :\Longleftrightarrow$ der i-te Knoten der Euler-Tour hat die Höhe h im kartesischen Baum $C^{can}(A)$
- *Das erste Vorkommen des Labels l*
  $R[l] = i :\Longleftrightarrow$ das Label l taucht in der Euler-Tour zum ersten mal am i-ten Knoten auf.

Da bei einer Euler-Tour die Knoten doppelt betrachtet werden, haben die Arrays H und E eine Länge von $n' = 2n - 1$, wobei $n := |A|$ sei. Die Arrays E und H haben somit doppelte Länge. Das Array $|R|$ hat eine Länge von n.
Da E, H und R alle für den Algorithmus notwendigen Informationen speichern, wird $C^{can}(A)$ anschließend nicht mehr benötigt und kann somit gelöscht werden. In Paper [3] wird nun gezeigt, dass folgendes gilt:

$$RMQ_A(l,r) = E\left[\pm 1 - RMQ_H\left(R[l], R[r]\right)\right].$$

Somit haben wir das eigentliche RMQ-Problem auf ein $\pm 1 RMQ$-Problem reduziert. Um nun eine schnelle Verarbeitung von $\pm 1 - RMQ_H$ zu gewährleisten, wird H in disjunkte Blöcke der Größe $\frac{log(n')}{2}$ aufgeteilt. Das Minimum eines jeden Blockes i wird dann in dem Array A' gespeichert und dessen Position in B.

$$A'[i] := \pm 1 - RMQ_H\left(i \cdot \tfrac{log(n')}{2}, (i+1) \cdot \tfrac{log(n')}{2} - 1\right)$$

Im Folgenden wird eine Anfrage $\pm 1 - RMQ_H(l,r)$ in drei Teile geteilt.

1. $\pm 1 - RMQ_H(l, l')$
2. $\pm 1 - RMQ_H(l', r')$
3. $\pm 1 - RMQ_H(r', r)$

wobei l' bzw. r' die Position des ersten Blockendes rechts bzw. links von l bzw. r bezeichnet. Somit erstreckt sich die Anfrage $\pm 1 - RMQ_H(l', r')$ genau über mehrere Blöcke. Es werden nun in einem weiteren Vorverarbeitungsschritt alle möglichen dieser blockübergreifenden Anfragen berechnet und in der Tabelle M gespeichert:

$$M[i,j] := \min\left\{A'\left[i, i + 2^j - 1\right]\right\} \text{ für } i \in \left\{0, \tfrac{2n'}{log(n')}\right\} \text{ und } j \in \left\{0, log\left(\tfrac{2n'}{log(n')}\right)\right\}$$

Die Tabelle M kann in linearer Zeit in n mit dynamischer Programmierung berechnet werden:

$$M[i,j] = \arg\min_{k \in \{M[i][j-1], M[i+2^{j-1}][j-1]\}} \left(A'[k]\right)$$

Es werden also die Einträge von $M[i][j]$ iterativ berechnet, wobei j mit jeder Iteration inkrementiert wird und $j = 1$ den Iterationsanker darstellt. Dieser ist eine Anfrage, die sich genau über einen Block erstreckt und somit bereits mit A' berechnet wurde. Die Tabelle M kann somit in $O(n)$ Zeit berechnet werden und benötigt

$$O\left(\tfrac{2n'}{log(n)'} \cdot \tfrac{2n'}{log(n')} \cdot \tfrac{2n'}{log(n')}\right) = O\left(n \cdot log(n)\right)$$

Bits Platz zur Kodierung. Der Trick ist es nun eine blockübergreifende Anfrage in zwei sich überlappende blockübergreifende Anfragen zu unterteilen, die bereits in M berechnet wurden. Genauer wählt man $l := \lfloor log\left(j - i\right) \rfloor$. Nun löst man die Anfrage mittels der bereits berechneten Tabelle M:

$$RMQ(i,j) = \arg\min_{k \in \left\{M[i][l], M[j-2^l+1][l]\right\}} \left\{A'k\right\}$$

Als zweites müssen die beiden Enden $\pm 1 - RMQ_H(l, l')$ und $\pm 1 - RMQ_H(r', r)$ der $\pm 1 - RMQ_H(l, r)$-Anfrage behandelt werden. Dies sind also Anfragen innerhalb eines Blockes. Das Preprocessing besteht hier aus der Vorberechnung aller möglichen $\pm 1 - RMQ_H$ innerhalb eines Blockes und dies für jeden Block. Dies wird mit dem *4-Russian-Trick* getan, der besagt, dass alle möglichen Berechnungen gemacht werden sollen, wenn dessen Anzahl ausreichend klein ist.
Als erstes wird die Anzahl der zu bearbeitenden Blöcke eingschränkt. In [3] wurde gezeigt, dass aufgrund der ± 1 Eigenschaft maximal $\sqrt{n'}$ unterschiedliche Blöcke vorberechnet werden müssen. Ein einzelner Block bleibt bzgl. einer RMQ äquivalent, wenn man den initialen Wert von allen

Einträgen des Blocks subtrahiert. Somit lässt sich ein Block durch eine ± 1 Vektor der Länge $2^{\frac{1}{2}\cdot(log(n')+1)}$ darstellen. Es gibt $\frac{1}{2}\frac{log(n')}{2} \cdot \left(\frac{log(n')}{2}+1\right)$ mögliche Wahlen für $i,j \in \left\{0, \frac{log(n')}{2}\right\}$ mit $i \neq j$. Somit gibt es genau so viele mögliche RMQs auf einem Block. Das heiß die Tabelle P, die alle notwendigen Informationen für eine beliebige Block-RMQ speichert, hat folgende Gestalt:

$$P\left[1, 2^{\frac{1}{2}\cdot(log(n')-1)}\right]\left[1, \frac{1}{2}\cdot\frac{log(n')}{2}\cdot\left(\frac{log(n')}{2}+1\right)\right]$$

Da aber nur $\sqrt{n'}$ viele verschiedene Blöcke betrachtet werden müssen, benötigt P

$$O\left(\sqrt{n'}log^2(n')\cdot log\left(\frac{log(n')}{2}\right)\right) = o(n)$$

Bits Platz zur Kodierung. Um innerhalb von P einen Block eindeutig identifiziren zu können, wird für jeden Block, der *Typ* des Blockes berechnet. Dieser ist als die Binärzahlinterpretation des folgenden Bitvektors definiert:

$$BV(i) := 0 \text{ falls } H[i+1] = H[i]+1 \text{ und } BV(i) := 1 \text{ sonst.}$$

2.3.1 Zusammenfassung

Die $RMQ_A(l,r)$-Anfrage wird durch eine Euler-Tour auf eine $\pm 1 - RMQ$-Anfrage reduziert. Das jetzt zu untersuchende Intervall von H wir in drei Teile unterteilt:

1. die startende Block-interne Anfrage
2. eine Blockübergreifende Anfrage
3. die endende Block-interne Anfrage

In den drei Teilen wird jeweils das Minimum bestimmt und schließlich wird dessen Minimum als globales Minimum bestimmt. Da M, E und R $O(n)$ Speicherplatz benötigen, folgt ein gesamter Speicherbedarf von $O\left(n \cdot log(n)\right)$ Bits.

3 Der neue Algorithmus

Die Idee dieses neuen Algorithmus ist, auch wie bei dem Vorherigen, dass der zu untersuchende Bereich in Blöcke unterteilt wir. Allerdings wird hier eine strukturiertere Unterteilung in sogenannte Superblöcke $B'_1, ..., B'_{\frac{n}{s'}}$ der Größe $s' := log^{2+\epsilon}n$ und konzeptionelle Blöcke $B_1, ..., B_{\frac{n}{s}}$ der Größe $s := \frac{log(n)}{2+\sigma}$, wobei $\epsilon > 0$ und $\sigma > 0$ Konstanten sind, vorgenommen. Der Verständlichkeit und Einfachheit halber wird im Folgenden angenommen, dass s' ein Vielfaches von s ist. Ähnlich wie im obigen Algorithmus wird nun eine Anfrage auf dem Intervall $A[l,r]$ in folgende Teilanfragen unterteilt:

1. eine Anfrage innerhalb eines konzeptionellen Blockes
2. eine Anfrage über mehrere konzeptionelle Blöcke, aber keinen Superblock
3. eine Anfrage über mehrere Superblöcke
4. eine Anfrage über mehrere konzeptionelle Blöcke, aber keinen Superblock
5. eine Anfrage innerhalb eines konzeptionellen Blockes

3.1 Eine platzsparende Datenstruktur für das Behandeln langer Anfragen

Als erstes werden Anfragen betrachtet, die sich genau über mehrere Superblöcke erstrecken. Hier wird analog zu 2.1 eine Tabelle $M'[0, \frac{n}{s'} - 1]\,[0, log\,(\frac{n}{s'})]$ aufgebaut, wobei $M'[i][j]$ den Index des Minimums von $A[is', (i + 2^j)\,s' - 1]$ speichert. Die Spalte $M[.][0]$ kann durch einen linearen Durchlauf des Arrays A gefüllt werden. Die weiteren Spalten $M[.][j]$ für $j > 0$ können iterativ über j mittels

$$M'[i][j] = \arg\,\min_{k \in M'[i][j-1], M'[i+2^{j-1}][j-1]}$$

berechnet werden.
Analog werden RMQs vorberechnet, die sich genau über mehrere konzeptionelle Blöcke aber keinen Superblock erstrecken. Die Positionen dieser RMQ werden in $M[0, \frac{n}{s} - 1][0, log\frac{s'}{s}]$ gespeichert, wobei $M'[i][j]$ das Minimum von $A[is, (i + 2^j)\,s - 1]$ speichert.

3.2 Eine platzsparende Datenstruktur für das Behandeln kurzer Anfragen

Als nächstes werden Anfragen betrachtet, die sich nur über ein Teilintervall eines konzeptionellen Blocks erstrecken. Zunächst wird die maximale Anzahl aller möglichen Teilintervalle der Länge s bestimmt:

Thereom 1. *Seien A und B zwei Arrays der Länge s, so sind folgende Aussagen äquivalent:*

1. $\forall 0 \leq l \leq r < s : RMQ_A\,(l, r) = RMQ_B\,(l, r)$
2. $C^{can}(A) = C^{can}(B)$

Das heiß es gibt genau so viele mögliche zu betrachtende Teilintervalle der Größe s, wie es kanonische kartesische Bäume der Größe s gibt.
Die Anzahl aller möglichen binären Bäume der Größe s ist mit C_s wohl bekannt:

$$C_s = \tfrac{1}{s+1} \cdot \left(\tbinom{2s}{s}\right) = \tfrac{4^s}{\sqrt{\pi}^{\frac{3}{2}}(1+O(s^{-1}))}$$

Auch hier findet der 4-Russian-Trick wieder seine Anwendung. Nachdem die Anzahl aller zu betrachtenden unterschiedlichen Blöcke mit C_s beschränkt wurden, werden für jeden konzeptionellen Block alle möglichen RMQs vorberechnet. Die ensprechenden Positionen der Minimas werden in der Tabelle P gespeichert. Da die Anzahl aller möglichen paarweise verschiedenen konzeptionellen Blöcken durch C_s beschränkt ist, müssen nur für so viele Blöcke Vorberechnungen gemacht werden. Um P bzgl. der Blocktypen zu indizieren muss eine Surjektion folgender Form berechnet werden:

$$t : A_s \longrightarrow \{0, ..., C_s - 1\} \text{ mit}$$
$$t\,(B_i) = t\,(B_j) \text{ falls } C^{can}\,(B_i) = C^{can}\,(B_j),$$

wobei A_s die Menge aller möglichen Arrays der Größe s sei. Hier wird also die Aussage von Theorem 1 ausgenutzt, dass der gleiche kanonische kartesiche Baum die gleichen RMQ impliziert. Die Surjektion t kann in linearer Zeit berechnet werden.
Für die Berechnung, bzw. die Speicherung der Vorberechnungen, des Minimums einer beliebigen RMQ innerhalb eines konzeptionellen Blockes wird auf das Paper [2] verwiesen. Hier wird

eine RMQ zunächst auf die Berechnung des kleinsten gemeinsamen Vorgängers innherhalb eines binären Baumes reduziert. Hier entspricht jedes Element des betrachteten Arrays einem Knoten und das Minimum zweier Elemente dem kleinstem gemeinsamen Vorgänger der entsprechenden Knoten. Des Weiteren wird ein Theorem vorgerstellt dass Vorberechnungen auf einem binären Baum ermöglicht, die die Abfrage des kleinsten gemeinsamen Vorgängers zweier Knoten in Konstanter Zeit zulassen. Diese Vorberechnungen benötigen insgesamt nur $s \cdot s$ Bits.

3.3 Platzverbrauch des Algorithmus

Nun wird der Platzverbrauch des oben vorgestellten Algorithmus analysiert. Der Algorithmus benötigt ingesamt vier Arrays zur Speicherung bereits vorberechneter Informationen:

1. M' zur Speicherung des Minimums eines superblockübergreifenden Intervalls
2. M zur Speicherung des Minimums eines Intervalls über mehrere konzeptionelle Blöcke
3. P zu Speicherung aller möglichen RMQ Ergebnisse innerhalb aller Blöcke
4. T zur Speicherung des Typs jedes konzeptionellen Blockes

3.3.1 Platzverbrauch von M'

Die Tabelle M' speichert in $M'[i][j]$ die Position des Minimums von $A\left[is', \left(i + 2^j\right) s' - 1\right]$ für $i \in \{0, ..., \frac{n}{s'}\}$ und $j \in \{0, ..., log\left(\frac{n}{s'}\right)\}$. Somit hat M' die Dimension:

$$\frac{n}{s'} \cdot log\left(\frac{n}{s'}\right) = \frac{n}{log^{2+\epsilon}} \cdot log\left(\frac{n}{log^{2+\epsilon}}\right).$$

Da in M' nur die Positionen innerhalb von A zu speichern sind und somit der maximale Wert n ist, werden zur Speicherung eines Eintrags $log(n)$ Bits benötigt. Daraus ergibt sich ein gesamter Speicherplatzbedarf von:

$$\frac{n}{log^{2+\epsilon}} \cdot log\left(\frac{n}{log^{2+\epsilon}}\right) \cdot log(n) = o(n).$$

3.3.2 Platzverbrauch von M

Die Tabelle M hat analog zu M' eine Dimension von

$$\frac{n}{s} \cdot log\left(\frac{n}{s}\right) = \frac{(2+\sigma)n}{log(n)} \cdot log\left((2 + \sigma)\, log^{1+\sigma} n\right).$$

In dem man jetzt jeweils in einem Block immer nur den offset zum Minimum des betrachteten Blocks betrachtet, haben die in M zu speichernden Indizes einen maximalen Wert von s'. Somit reichen für die Kodierung eines Eintrags in M $log(s')$ Bits aus. Also ergibt sich für M ein Gesamtspeicherbedarf von:

$$O\left(\frac{n}{log(n)} \cdot log\left(log^{1+\epsilon n}\right) \cdot log\left(log^{2+\epsilon} n\right)\right) = o(n)$$

3.3.3 Platzverbrauch von T

Insgesamt sind Typen für $\frac{n}{s}$ Blöcke zu speichern. Mit Theorem 1 folgt das eine Nummerierung aller benötigten Typen bis maximal $O\left(\frac{4^s}{s^{\frac{3}{2}}}\right)$ läuft. Somit werden:

$$\frac{n}{s} \cdot log \cdot \left(O\left(\frac{4^s}{s^{\frac{3}{2}}}\right)\right) = \frac{n}{s} \cdot (2s - O(logs)) = 2n - O\left(\frac{n}{log(n)log(n)log(n)}\right) = 2n - o(n)$$

Bits zur Kodierung von T benötigt.

3.3.4 Platzverbrauch von P

Da für die Kodierung aller Anfragen innerhalb eines Blockes $O(s \cdot s)$ Bits benötigt werden, kann P mit:

$$O\left(\frac{4^s}{s^{\frac{3}{2}}} \cdot s \cdot s\right) = O\left(n^{\frac{2}{2+\sigma}} \cdot \sqrt{log(n)}\right) = o\left(\frac{n}{log(n)}\right)$$

Bits kodiert werden.

3.3.5 Der Gesamtspeicherverbrauch und dessen Optimalität

Somit ergibt sich in der Summe der Speichergrößen ein gesamter Speicherverbrauch von $2n-o(n)$, d.h. der Platzverbrauch zur Kodierung der konzeptionellen Blöcke dominiert den Platzverbrauch der restlichen Arrays. Für folgende Untersuchungen sei das *Min-Probe* Model gegeben, in dem nur Vergleiche der Form $\arg\min\{A[i], A[j]\}$ gezählt werden. Das folgende Theorem zeigt, dass ein Platzverbrauch von $2n - o(n)$ im *Min-Probe* Model optimal ist.

Thereom 2. *Im min-probe Model wird für jedes Array der Größe n eine Hilfsdatenstruktur mindestens der Größe $2n - o(n)$ benötigt, um $RMQ_A(l, r)$s für alle $0 \leq l \leq r < n$ in konstanter Zeit zu beantworten.*

Beweis. Sei D_A diese Hilfsdatenstruktur. Da eine konstante Laufzeit $O(k)$ erlaubt ist, kann die Hilfsdatenstruktur k mal befragt werden. Somit ergeben sich $2^k =: K$ viele Befragungsergebnisse. Angenommen es gäbe weniger als $\frac{C_n}{K+1}$ solcher D_As. Dann müssten $K + 1$ Arrays $\{A_0, ..., A_K\}$ der Länge n existieren, wobei die jeweiligen D_{A_i} alle gleich sind, aber mit paarweise verschiedenen kanonischen kartesischen Bäumen. Da der Algorithmus für maximal K dieser Hilfsarrays verschiedene Lösungen für RMQ Anfragen geben kann, gibt er für zwei Arrays jeweils die gleiche Lösung zurück. Diese seien A_i und A_j. Da aber A_i und A_j verschiedene kanonische kartesische Bäume haben ist dies eine Wiederspruch. Also müssen mindestens $\frac{C_n}{K+1}$ verschiedene Entscheidungen für den Inhalt der Hilfsdatenstrukur getroffen werden. Somit verbraucht diese mindestens $log\left(\frac{C_n}{K+1}\right) = 2n - o(n) - O(1)$ Bits. □

4 Verbesserungen des Enhanced Suffix Arrays

In diesem Kapitel wird eine Anwendung von RMQ besprochen, das String Matching. Das Problem besteht darin ein Muster in einem bekannten Text zu suchen. Um diese Suche zu beschleunigen, wird der zu untersuchende Text aufbereitet und Hilfsdatenstrukturen angelegt. Eine dieser

Datenstrukturen ist der Suffix Tree. Jedoch ein großer Nachteil dieser Hilfsdatenstruktur ist der erhöhte Speicherverbrauch. Das *Suffix Array* hingegen ist solch eine Datenstruktur, die einen verbesserten Speicherplatzverbrauch hat. In [4] wird gezeigt, wie Algorithmen, die auf einer bottom-up Travesierung des Suffix Trees beruhen, durch ein parallelen Scan des Suffix- und LCP Arrays simuliert werden können.
Mit Hilfe einer erweiterten Version des Suffix Arrays hingegen, dem *Enhanced Suffix Array*, können Algorithmen simuliert werden, die auf einer top-down Travesierung des Suffix Trees beruhen. Um dies effizient zu ermöglichen, wird eine weitere Tabelle benötigt. Diese kann nun mit den RMQ-Anfragen auf dem zu untersuchenden Text simuliert werden. Da der Platzverbrauch des zuvor besprochenen RMQ-Algorithmus einen geringeren Speicherplatzverbrauch als die im Enhanced Suffix Array zusätzlich benötigte Tabelle hat, wird hiermit eine Speicherplatzreduzierung des Enhanced Suffix Arrays erreicht.

4.1 Das Enhanced Suffix Array

Die Grundform des Enhanced Suffix Arrays besteht aus zwei Arrays:

- das eigentliche *Suffix Array* und
- das *LCP Array*

Sei SA das Suffix Array. Dann ist das LCP Array wie folgt definiert:

$$LCP[i] := LCP(SA[i], SA[i+1]),$$

wobei $LCP(A, B)$ die Länge des längsten gemeinsamen Präfixes der Strings A und B sei. Die grundlgende Idee des Enhanced Suffix Arrays ist, dass jeder innere Knoten des Suffix Trees einem speziellem Intervall des LCP Arrays entspricht. Diese Abhängigkeit zeigt sich genauer in folgendem Theorem:

Thereom 3. *Sei T, SA und LCP der Suffix Tree bzgl. t, dessen Suffix Array und dessen LCP Array. Dann sind die beiden folgenden Aussagen äquivalent:*

- *Es existiert ein innerer Knoten in T, der ein Teilwort ϕ von t repräsentiert*
- *es existieren Indizes $1 \leq l < r \leq n$, so dass folgendes gilt:*
  1. $LCP[l] < |\phi|$ *und* $LCP[r+1] < |\phi|$
  2. $\forall i \in \{l+1, ..., r\} : LCP[i] \geq |\phi|$ *und* $t_{SA[q]..SA[q]+|\phi|-1}$
  3. $\exists q \in \{l+1, ..., r\} : LCP[q] = |\phi|$

Ein Knoten in einem Suffix Tree repräsentiert den Präfix einer bestimmten Menge von Suffixes. Wenn der Algorithmus an diesem Knoten angekommen ist, heißt dies also, dass diese Präfixe möglicher Kandidaten gesuchter Muster sind. Der besuchte Knoten repräsentiert den Präfix ϕ. Somit ist der Präfix der möglichen Kandidaten gesuchter Muster gleich ϕ.
Die zweite Aussage besagt, dass in dem zu untersuchenden Text t eine Menge von Suffixes existiert, die einen Präfix ϕ haben. Diese Menge entspricht einem Intervall natürlicher Zahlen. Diese stellen die Indizes des Suffix Arrays dar. Des Weiteren soll der nächst kleinere bzw. größere Suffix, bzgl. der lexikographischen Ordnung nicht mehr den Präfix ϕ haben. Als letztes gilt noch,

dass mindestens einer dieser Suffixe mit seinem lexikographischen Nachfolger in dem $|\phi|+1$-ten Zeichen nicht übereinstimmt.
Diese Menge von Suffixes wird im Theorem durch die Indizes l und r des Suffix Arrays angegeben. Das heißt also, dass genau die Suffixe $SA[i]$ für $i = l, ..., r$ den gemeinsamen Präfix ϕ haben. Dies ist der maximale gleiche Präfix. Das Intervall hat die maximale Breite bzgl. dieses Präfixes. Solche Intervalle werden $|\phi|$-Intervall genannt und werden speziell mit $|\phi| - [l : r]$ angegeben. Jeder Index i innerhalb dieses Intervalles, für den $LCP[i] = |\phi|$ gilt, wird $|\phi|$-Index gennant. Das heißt ein $|\phi|$-Index bestimmt den maximalen gleichen Präfix eines Intervalls.
Wenn man nun zu einem gegebenen $k-[l : r]$ Intervall ein k'-Intervall mit $k' > k$ betrachtet und dieses in $[l : r]$ enthalten ist, und so heißt diese Intervall *Child Intervall* von $[l : r]$, falls es kein weiters Child Intervall von $[l : r]$ gibt, welches das k'-Intervall einschließt. Das $k-[l : r]$ Intervall heißt *Super Intervall* vom k'-Intervall. Diese direkte Super- bzw. Child-Beziehung entspricht genau der direkten Vorgänger- bzw. Nachfolger-Beziehung zweier Knoten im Suffix Tree.
Die Child Intervalle eines gegebenen k-Intervalles können mit Hilfe des folgenden Lemmas bestimmt werden:

Lemma 1. *Sei ein $k - [l, r]$ Intervall gegeben und seien $i_1 < i_2 < ... < i_m$ die entsprechenden k-Indizes, so sind $[l : i_1], [i_1 + 1 : i_2], ..., [i_m : r]$ die zugehörigen Child Intervalle.*

Mit Hilfe dieses Lemmas ist nun möglich, eine Top Down Travesierung eines Suffix Trees auf Basis des Enhanced Suffix Arrays durchzuführen. Um den Zugriff auf die Child Intervalle eines k-Intervalls, also auf die zugehörigen k-Indizes, in konstanter Zeit zu gewährleisten, werden diese in einem zusätzlichen Array $C[1..n]$ gespeichert, der Child Tabelle.

4.2 Eine RMQ basierte Repräsentation der Child Tabelle

Das folgende Lemma zeigt, wie sich das Problem der Berechnung der k-Indizes auf das Problem RMQ effizient reduzieren läßt:

Lemma 2. *Sei $[l : r]$ ein k-Intervall, dann sind die enstsprechenden k-Indizes:*

$$i_1 = RMQ_{LCP}\,(l + 1, r)\,; ...; i_m = RMQ_{LCP}\,(i_{m-1}, r)$$

Dies ergibt sich aus der Definition der k-Indizes, die besagt, dass ein k-Index einen Präfix der Länge k mit seinem Nachfolger gemein hat. Da für alle Indizes innerhalb des k-Intervalls gilt, dass sie eine Präfix mit mindestens der Größe k mit ihrem Nachfolger gemein haben, ist die Menge der Einträge in dem Intervall $[l : r]$ gleich der Menge der k Intervalle. Darüber hinaus ist das Minimum bzgl. einer RMQ Anfrage eindeutig bestimmt, indem im Falle einer Gleichheit zusätzlich der minimale Index genommen wird. Hieraus ergibt sich die Behauptung.

Das folgende Theorem fasst die Ergebnisse dieses Kapitels zusammen:

Thereom 4. *Innerhalb eines beliebigen Algorihtmus, der auf einer Top-Down Travesierung eines Suffix Trees t basiert, kann t durch eine Datenstruktur ersetzt werden, die einen Platzverbrauch von $|SA| + 4n + o(n)$ hat. Diese Ersetzung verschlechtert nicht die asymptotische Laufzeit*

des Algorithmus.

Mit Lemma 1 und 2 folgt, dass sich die Child Intervalle mit RMQ Anfragen auf dem LCP-Array berechnen lassen. Dies ist in konstanter Zeit möglich. Mit Theorem 4 folgt, dass sich somit implizit die Nachfolgerknoten im Suffix Tree berechnen lassen. Der Platzverbrauch für die RMQ Berechnungen ist laut Theorem 3 $2n + o(n)$. In [5] wurde gezeigt, dass das LCP-Array einen Platzverbrauch von $2n + o(n)$ hat. Somit folgt die Behauptung

4.3 Fazit

Es wurden zwei Algorithmen betrachtetet, die beide dass RMQ Problem in linearer Vorverarbeitungszeit und konstanter Anfragezeit lösen können. Spezieller wurde ein neuer Algorithmus entworfen, der bzgl. des Speicherplatzes asymptotisch optimal arbeitet.
Im zweiten Teil wurde die Anwendung von RMQs bzw. des neuen Algorithmus in dem Enhanced Suffix Array besprochen. Hier wurde das Child Array durch RMQs simuliert, was mit der RMQ-Optimalität eine Speicherplatzreduzierung zur Folge hat.

Literatur

[1] Johannes Fischer und Volker Heun, *A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array*

[2] Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T., *Nearest common ancestors: A survey and a new distributed algorithm.*

[3] Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P., *Lowest common ancestors in trees and directed acyclic graphs*

[4] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K., *Linear-time longest common-prefix computation in suffix arrays and its applications*

[5] Gabow, H.N., Bentley, J.L., Tarjan, R.E., *Scaling and related techniques for geometry problems*