Frequent String Mining in mehreren Datenbanken

Peter Fricke

5. Mai 2009

Adrian Kügel, Enno Ohlebusch (2008): „A space efficient solution to the frequent string mining problem for many databases.“

Ordnung ist das halbe Leben!
(Graf Lexiko, Anno 1234)

Einführung: Was, warum und wie?
- Problemdefinition („Was?“)
- Motivation („Warum?“)
- Grundidee („Wie?“)

Datenstruktur und Basisalgorithmus
- Struktur in den Daten
- Datenstruktur
- Der Basisalgorithmus

Speicherplatzeffiziente Erweiterung des Basisalgorithmus
- Gemeinsame Sortierung berechnen
- Partnersuche
- Schneiden

Ergebnisse

Problemdefinition: Begriffe

- m Datenbanken $D_1, \ldots D_m$ sind Mengen von Strings über Σ
- Elemente der Datenbank sind die „Originalstrings“ ψ

Problemdefinition: Begriffe

- m Datenbanken $D_1, \ldots D_m$ sind Mengen von Strings über Σ
- Elemente der Datenbank sind die „Originalstrings" ψ
- Häufigkeit eines Strings ϕ:
  $freq(\phi, D) := |\{\psi \in D : \phi \text{ ist Teilstring von } \psi\}|$

Problemdefinition: Frequent String Mining Problem

Frequent String Mining Problem
Gegeben:

- m Datenbanken $D_1, \ldots D_m$ sind Mengen von Strings über Σ
- m Paare von Häufigkeitsschwellwerten $(minf_1, maxf_1), \ldots (minf_m, maxf_m)$

Problemdefinition: Frequent String Mining Problem

Frequent String Mining Problem
Gegeben:

- m Datenbanken $D_1, \ldots D_m$ sind Mengen von Strings über Σ
- m Paare von Häufigkeitsschwellwerten $(minf_1, maxf_1), \ldots (minf_m, maxf_m)$
- Definition: Ein *relevanter Substring* ist ein beliebiger String, dessen Häufigkeit in mindestens einer Datenbank zwischen den Häufigkeitsschwellwerten liegt: $minf_i \leq freq(\phi, D_i) \leq maxf_i$
- Definition Frequent String Mining Problem: Finde Schnittmenge der relevanten Substrings der einzelnen Datenbanken.

Motivation: Huntington's disease

- Beispiel: Huntintons's Disease
- Vermutung: Ursache ist Gendefekt in bestimmtem Bereich
- Gensequenzen (=Strings) gesunder Individuen in einer Datenbank
- Gensequenzen kranker Individuen in anderer Datenbank
- Welche Strings kommen in einer Datenbank sehr häufig, in der anderen sehr selten vor?
- $\rightarrow$ Frequent String Mining Problem mit Häufigkeitsschwellwerten $(minf_1, maxf_1) = (n_{groß}, \infty)$, $(minf_2, maxf_2) = (0, n_{klein})$

Grundidee

- Es gibt schon einen Linearzeitalgorithmus (!)
- Platzbedarf proportional zur *Gesamtgröße* der Datenbanken
- Idee für Verbesserung: Löse Problem mit diesem Algorithmus für jede Datenbank einzeln...
- ...und konstruiere Gesamtlösung durch Schneiden der Teillösungen
- Platzbedarf *dann* proportional zur *Größe der größten Datenbank*

Grundidee

- Es gibt schon einen Linearzeitalgorithmus (!)
- Platzbedarf proportional zur *Gesamtgröße* der Datenbanken
- Idee für Verbesserung: Löse Problem mit diesem Algorithmus für jede Datenbank einzeln...
- ...und konstruiere Gesamtlösung durch Schneiden der Teillösungen
- Platzbedarf *dann* proportional zur *Größe der größten Datenbank*
- ToDo:
- Effizienten Schneidealgorithmus entwickeln

Grundidee

- Es gibt schon einen Linearzeitalgorithmus (!)
- Platzbedarf proportional zur *Gesamtgröße* der Datenbanken
- Idee für Verbesserung: Löse Problem mit diesem Algorithmus für jede Datenbank einzeln...
- ...und konstruiere Gesamtlösung durch Schneiden der Teillösungen
- Platzbedarf *dann* proportional zur *Größe der größten Datenbank*
- ToDo:
- Effizienten Schneidealgorithmus entwickeln
- Implizite Darstellung der Teillösungen (Größe nicht beschränkt!)

Einordnung

Das ist genau wie...

- Wir haben diskutiert:
- Terabyte Daten kann man nicht verschicken
- $\rightarrow$ Lokal Modelle aus *Teilen* der Daten extrahieren, Modelle verschicken, zusammen nutzen oder zusammensetzen.
- Hier: Daten passen nicht in den Hauptspeicher, Informationen aus *Teilen* der Daten extrahieren, zusammensetzen.
- Anderer Grund, ähnliches Vorgehen.

Vorverarbeitung

- Vorverarbeitung für jede Datenbank:
- Verketten der Elemente der Datenbank zu *einem* String
- Trennsymbol zwischen den Originalstrings der Datenbank: $\#$
- Abschlusssymbol \$, beide nicht im Alphabet Σ enthalten.

Vorverarbeitung

- Vorverarbeitung für jede Datenbank:
- Verketten der Elemente der Datenbank zu *einem* String
- Trennsymbol zwischen den Originalstrings der Datenbank: $\#$
- Abschlusssymbol $, beide nicht im Alphabet Σ enthalten.
- Eine Datenbank D ist also eine Menge von Originalstrings ψ und wird dargestellt als Gesamtstring $T^D = aaaa\#baaab\#aba\#\$$, der eine Verkettung von Originalstrings ist.

Ein Beispiel

- Was sind die relevanten Substrings in der Datenbank D, die durch $T^D = aaaa\#baaab\#aba\#\$$ dargestellt wird, für die Häufigkeitsschwellwerte (4,9)?
- $\emptyset$! Warum? $freq(\phi, D) := |\{\psi \in D : \phi \text{ ist Teilstring von } \psi\}|$
- Alle Vorkommen *insgesamt* zu berechnen ist einfacher!
- Idee Basisalgorithmus: Berechne alle Vorkommen $S_D(\phi)$ von ϕ *insgesamt* und ziehe Korrekturterme $C_D(\phi)$ ab für Vorkommen in demselben Originalstring der Datenbank. Hier: Korrekturterm 6 für $freq(a, D)$, weil a im ersten Originalstring vierfach, im zweiten dreifach und im dritten Originalstring doppelt vorkommt.
  $freq(a, D) = S_D(a) - C_D(a) = 9 - 6 = 3 < 4$

Ein Beispiel

- Was sind die relevanten Substrings in der Datenbank D, die durch $T^D = aaaa\#baaab\#aba\#\$$ dargestellt wird, für die Häufigkeitsschwellwerte (4,9)?
- $\emptyset$! Warum? $freq(\phi, D) := |\{\psi \in D : \phi \text{ ist Teilstring von } \psi\}|$
- Alle Vorkommen *insgesamt* zu berechnen ist einfacher!
- Idee Basisalgorithmus: Berechne alle Vorkommen $S_D(\phi)$ von ϕ *insgesamt* und ziehe Korrekturterme $C_D(\phi)$ ab für Vorkommen in demselben Originalstring der Datenbank. Hier: Korrekturterm 6 für $freq(a, D)$, weil a im ersten Originalstring vierfach, im zweiten dreifach und im dritten Originalstring doppelt vorkommt.
  $freq(a, D) = S_D(a) - C_D(a) = 9 - 6 = 3 < 4$
- Für alle ϕ? Problem: Quadratisch viele. Später lösen!

Suche nach Struktur in den Daten

- Eine Datenbank wird dargestellt als String $T^D = aaaa\#baaab\#aba\#\$$

Suche nach Struktur in den Daten

- Eine Datenbank wird dargestellt als String $T^D = aaaa\#baaab\#aba\#\$$
- Erster Versuch: Schreibe die n Suffixe von T^D untereinander
- Ist das Struktur?
- Keine ausreichend interessante Struktur!

Startposition	Suffix
1	aaaa#baaab#aba#$
2	aaa#baaab#aba#$
3	aa#baaab#aba#$
4	a#baaab#aba#$
5	#baaab#aba#$
6	baaab#aba#$
7	aaab#aba#$
8	aab#aba#$
9	ab#aba#$
10	b#aba#$
11	#aba#$
12	aba#$
13	ba#$
14	a#$
15	#$
16	$

Index	Startposition	Suffix
1	16	$
2	15	#$
3	11	#aba#$
4	5	#baaab#aba#$
5	14	a#$
6	4	a#baaab#aba#$
7	3	aa#baaab#aba#$
8	2	aaa#baaab#aba#$
9	1	aaaa#baaab#aba#$
10	7	aaab#aba#$
11	8	aab#aba#$
12	9	ab#aba#$
13	12	aba#$
14	10	b#aba#$
15	13	ba#$
16	6	baaab#aba#$

Ähnlichkeit

- Sortieren hilft!
- Identifiziere Suffix mit seiner Position i in der Liste der sortierten Suffixe
- Mit der Startposition („$SA[i]$“) kann ich wieder auf das Suffix selbst zugreifen:
- $T^D_{SA[i]\dots n}$ ist das lexikografisch i-te Suffix
- SA nennt man auch *Suffix Array*.
- *Ähnlichkeit* von Suffixen ist Länge des längsten gemeinsamen Präfixes: $lcp(x, y)$ („length of longest common prefix“)
- Ähnlichkeit zum lexikografischen Vorgänger: $LCP[i]$

Index	SA	LCP	Suffix
1	16	0	$
2	15	0	#$
3	11	1	#aba#$
4	5	1	#baaab#aba#$
5	14	0	a#$
6	4	2	a#baaab#aba#$
7	3	1	aa#baaab#aba#$
8	2	2	aaa#baaab#aba#$
9	1	3	aaaa#baaab#aba#$
10	7	3	aaab#aba#$
11	8	2	aab#aba#$
12	9	1	ab#aba#$
13	12	2	aba#$
14	10	0	b#aba#$
15	13	1	ba#$
16	6	2	baaab#aba#$

Struktur!

- „Ähnlichkeit mindestens l“ definiert Äquivalenzrelation auf Suffixen
- Äquivalenzklassen entsprechen Intervallen in der sortierten Suffixliste
- Jede Äquivalenzklasse kann ich mit dem gemeinsamen Präfix der Mitglieder markieren. („Das ω-Intervall“)

Struktur!

- „Ähnlichkeit mindestens l" definiert Äquivalenzrelation auf Suffixen
- Äquivalenzklassen entsprechen Intervallen in der sortierten Suffixliste
- Jede Äquivalenzklasse kann ich mit dem gemeinsamen Präfix der Mitglieder markieren. („Das ω-Intervall")
- Ich kann l variieren: Neue Äquivalenzrelation, neue Partitionierung. Also Hierarchie von Äquivalenzklassen.

Struktur!

- „Ähnlichkeit mindestens l" definiert Äquivalenzrelation auf Suffixen
- Äquivalenzklassen entsprechen Intervallen in der sortierten Suffixliste
- Jede Äquivalenzklasse kann ich mit dem gemeinsamen Präfix der Mitglieder markieren. („Das ω-Intervall")
- Ich kann l variieren: Neue Äquivalenzrelation, neue Partitionierung. Also Hierarchie von Äquivalenzklassen.
- Erhöhe ich l auf $l + 1$, werden die Intervalle an den Positionen mit LCP[i]=l gespalten, die neuen Intervalle sind Teilintervall des alten Intervalls. Elter-Kind-Beziehung definierbar.

lcp-Intervall

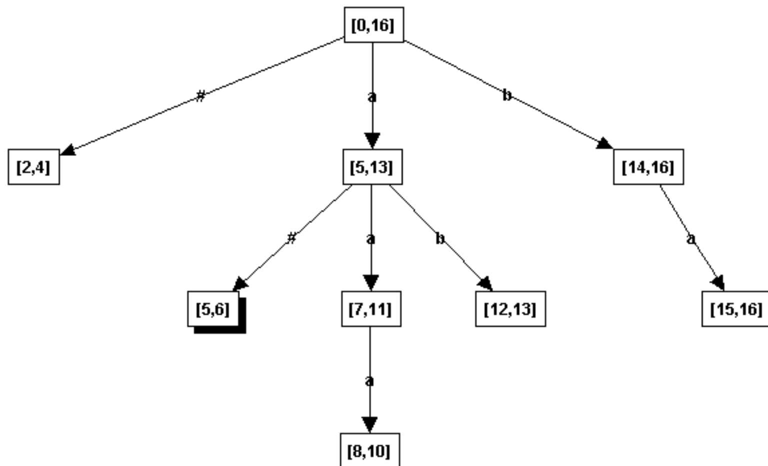
Sei $1 \leq i < j \leq n$. Ein lcp-Intervall $[i, j]$ vom Wert l (auch l-Intervall genannt) erfüllt folgende Bedingungen:

- $LCP[i] < l$ und $LCP[j+1] < l$
- $LCP[k] \geq l$ für alle k mit $i < k \leq j$
- $LCP[k] = l$ für mindestens ein $i < k \leq j$ („l-Index“)

lcp-Intervall

- Ein l'-Intervall $[i', j']$ ist *eingebettet* in ein l-Intervall $[i, j]$, wenn $i \leq i' \leq j' \leq j$ und $l < l'$.
- $[i, j]$ *umschließt* dann $[i', j']$.
- Wenn es kein anderes lcp-Intervall $[i'', j'']$ gibt, das von $[i, j]$ eingebettet ist und $[i', j']$ umschließt, ist $[i', j']$ *Kind* von $[i, j]$.

lcp-Intervall-Baum



lcp-Intervall-Baum

- Präfixbaum wie bei FPgrowth.
- Einzufügende Daten (Suffixe, geordnete Mengen) sind Pfade.
- Naiver Aufbau: Pfade übereinanderlegen.
- Struktur: Gemeinsamkeiten sind als gemeinsame Teilpfade zu erkennen
- Hier kompaktere Darstellung: Bei FPgrowth wurde pro Kante *ein* Item abgearbeitet, hier können pro Kante auch mehrere Buchstaben abgearbeitet werden (Hinweis: Im Bild auf der vorigen Seite wurden die ursprünglichen Blätter (=Intervalle der Länge eins) gestrichen, um eine noch kompaktere Darstellung zu erreichen).

Verwenden effizienter Algorithmen

- Wir müssen nicht alle Teilprobleme selbst lösen, sondern können existierende Algorithmen verwenden:
- Algorithmus für „Konstruktion“ des lcp-Intervall-Baums: Linear in Zeit und Raum.
- Virtuelle Datenstruktur/Kapselung, intern Arrays/Tabellen.
- Intervallgrenzen lassen sich sehr effizient finden: Range Minimum Queries $RMQ_{LCP}(i,j) := \arg\min_{i<m\leq j} LCP[m]$ sind nach linearer Vorverarbeitung in konstanter Zeit ausführbar. Später nützlich.

Quadratisch viele Häufigkeiten in Linearzeit berechnen?

- Häufigkeit $freq(\omega, D)$ müssen wir nur für die ω berechnen, die einem Knoten des Intervallbaums zugeordnet sind (dem Knoten, der das ω-Intervall repräsentiert). Warum?
- Beispiel: Suffix 16. Es reicht, die Häufigkeit des Gesamtstrings zu kennen, schon kenne ich die Häufigkeiten für alle Präfixe mit Länge mindestens 3 (Häufigkeit: 1).
- Jeder Knoten, der nicht Blatt ist, hat mindestens zwei Kinder (Intervalle der Länge 1 sind Blätter).
- n Blätter, also hat Baum lineare Größe.
- Also berechnen wir nur linear viele Häufigkeiten.
- Platzsparend speichern. Später!

Wo sind wir?

Das waren Datenstruktur und Vorüberlegungen. Folgendes ist noch zu tun:

- Wir müssen für jeden Knoten im lcp-Intervallbaum, der ein ω-Intervall repräsentiert, die Anzahl $S_D(\omega)$ der Vorkommen von ω in T^D *insgesamt* berechnen.
- Die eigentlich gesuchten Häufigkeiten berechnen wir so nicht ganz korrekt, weil wir Strings, die in einem Originalstring mehrfach vorkommen, fälschlich mehrfach zählen. Wir müssen also für jedes dieser ω einen Korrekturterm $C_D(\omega)$ berechnen, so dass wir $freq(\omega, D) = S_D(\omega) - C_D(\omega)$ berechnen können.
- Wir müssen die Ergebnisse, die (möglicherweise quadratisch vielen) relevanten Substrings platzsparend darstellen.

Häufigkeiten berechnen

Berechne alle Vorkommen $S_D(\phi)$ von ϕ *insgesamt* und ziehe Korrekturterme $C_D(\phi)$ ab für mehrfaches Vorkommen in demselben Originalstring der Datenbank.
Erinnerung: $T^D = aaaa\#baaab\#aba\#\$$,
$freq(a, D) = S_D(a) - C_D(a) = 9 - 6 = 3$

Häufigkeiten berechnen

Berechne alle Vorkommen $S_D(\phi)$ von ϕ *insgesamt* und ziehe Korrekturterme $C_D(\phi)$ ab für mehrfaches Vorkommen in demselben Originalstring der Datenbank.
Erinnerung: $T^D = aaaa\#baaab\#aba\#\$$,
$freq(a, D) = S_D(a) - C_D(a) = 9 - 6 = 3$

- s^k ist der k-te Originalstring.
- $S_D(\phi) = |\{(j,k) : s^k_{j \cdots j+|\phi|-1} = \phi\}|$
- $C_D(\phi) = \sum_{s^k \in D : \phi \preceq s^k} (|\{j : s^k_{j \cdots j+|\phi|-1} = \phi\}| - 1)$
- $freq(\phi, D) = S_D(\phi) - C_D(\phi)$

Häufigkeiten berechnen

Berechne alle Vorkommen $S_D(\phi)$ von ϕ *insgesamt* und ziehe Korrekturterme $C_D(\phi)$ ab für mehrfaches Vorkommen in demselben Originalstring der Datenbank.
Erinnerung: $T^D = aaaa\#baaab\#aba\#\$$,
$freq(a, D) = S_D(a) - C_D(a) = 9 - 6 = 3$

- s^k ist der k-te Originalstring.
- $S_D(\phi) = |\{(j,k) : s^k_{j \cdots j+|\phi|-1} = \phi\}|$
- $C_D(\phi) = \sum_{s^k \in D : \phi \preceq s^k} (|\{j : s^k_{j \cdots j+|\phi|-1} = \phi\}| - 1)$
- $freq(\phi, D) = S_D(\phi) - C_D(\phi)$
- Für ϕ-Intervall [l,r] ist $S_D(\phi) = r - l + 1$

Korrekturterme berechnen 1

Korrekturterme berechnen - Idee:

- Mehrfaches Auftreten von Strings in demselben Originalstring soll gezählt werden.
- Jedes Auftreten des Strings ordnen wir dem Suffix zu, dessen Präfix er ist.
- Uns interessieren also Paare von Suffixen, die in demselben Originalstring beginnen und ein gemeinsames Präfix ϕ der Länge $k > 0$ haben.

Korrekturterme berechnen 1

Korrekturterme berechnen - Idee:

- Mehrfaches Auftreten von Strings in demselben Originalstring soll gezählt werden.
- Jedes Auftreten des Strings ordnen wir dem Suffix zu, dessen Präfix er ist.
- Uns interessieren also Paare von Suffixen, die in demselben Originalstring beginnen und ein gemeinsames Präfix ϕ der Länge $k > 0$ haben.
- Wenn wir ein solches Paar gefunden haben, wollen wir *alle* Probleme, die dieses Paar verursacht (auch kürzere Präfixe des gemeinsamen Präfixes), „notieren“.
- Suffixpaar mit Präfix ϕ hat Mehrfachzählungen nur für Strings der Länge $\leq k$ verursacht, also soll die Notierung für genau diese sichtbar sein.

Korrekturterme berechnen 2

- Benutze Hilfsarray C' der Länge n
- Iteriere lexikografisch aufsteigend über Suffixe
- Finde heraus, in welchem Originalstring ψ das aktuelle Suffix $T_{SA[e]\ldots n}$ beginnt (Zeilen färben im Beispiel auf S. 42)
- Finde den gespeicherten Index d des letzten Vorkommens eines im Originalstring ψ beginnenden Präfixes.
- Für dieses Paar $T_{SA[d]\ldots n}, T_{SA[e]\ldots n}$ von lexikografisch adjazenten Suffixen, die im selben Originalstring beginnen, erhöhe $C'[m]$ um eins für $m = \arg\min_{i<m\leq j} LCP[m]$

Korrekturterme berechnen 2

- Benutze Hilfsarray C' der Länge n
- Iteriere lexikografisch aufsteigend über Suffixe
- Finde heraus, in welchem Originalstring ψ das aktuelle Suffix $T_{SA[e]\dots n}$ beginnt (Zeilen färben im Beispiel auf S. 42)
- Finde den gespeicherten Index d des letzten Vorkommens eines im Originalstring ψ beginnenden Präfixes.
- Für dieses Paar $T_{SA[d]\dots n}, T_{SA[e]\dots n}$ von lexikografisch adjazenten Suffixen, die im selben Originalstring beginnen, erhöhe $C'[m]$ um eins für $m = \arg\min_{i<m\leq j} LCP[m]$
- Sei $\{T_{SA[i]\dots n} : l \leq i \leq r\}$ die Menge der Suffixe, die ϕ als Präfix haben:
- $C_D(\phi) = \sum_{i=l+1}^{r} C'[i]$

Berechnung des Hilfsarrays C'

Algorithm 1: Berechnung des Hilfsarrays C'

Input: Suffix-Array SA und *lcp*-Array LCP der Größe n

Output: Hilfsarray C' der Größe n

last sei ein Array der Größe $|D| := |\{\psi : \psi \in D\}|$

for $i = 1, \ldots, n$ **do**
  Setze k so, dass $T^D_{SA[i]\ldots n}$ im k-ten Originalstring beginnt.
  if $last[k] \neq 0$ **then**
    $l = RMQ_{LCP}(last[k] + 1, i)$
    $C'[l] + +$
  $last[k] = i$

Korrekturterme berechnen 3

- Wir können nun berechnen: $C_D(\phi) = \sum_{i=l+1}^{r} C'[i]$
- Zweites Hilfsarray C'' mit $C''[i] = \sum_{j=1\ldots i} C'[j]$
- Nun ist $C_D(\phi) = \sum_{i=l+1}^{r} C'[i] = C''[r] - C''[l]$
- Speicher sparen: C' wieder verwenden.
- Das Beispiel auf der nächsten Seite zeigt den Ablauf des Algorithmus. Der Algorithmus berechnet nur C' und C'', die drei bunten Spalten werden nicht explizit berechnet.

i	LCP	Suffix	aba#$	baaab#	aaaa#	C'	C''
1	0	$					0
2	0	#$	x			1	1
3	1	#aba#$					1
4	1	#baaab#aba#$					1
5	0	a#$	x	x	x	3	4
6	2	a#baaab#aba#$					4
7	1	aa#baaab#aba#$	x		x	2	6
8	2	aaa#baaab#aba#$			x	1	7
9	3	aaaa#baaab#aba#$			x	1	8
10	3	aaab#aba#$					8
11	2	aab#aba#$		x		1	9
12	1	ab#aba#$		x		1	10
13	2	aba#$					10
14	0	b#aba#$	x	x		2	12
15	1	ba#$		x		1	13
16	2	baaab#aba#$					13

Ergebnisse implizit speichern

Ergebnisse implizit speichern:

- Wir speichern relevante Strings an dem lexikografisch kleinsten Suffix, dessen Präfix sie sind.
- Die Mengen von relevanten Strings an einem Suffix sind nicht zersplittert, sondern bilden ein Intervall. Grund: Apriori-Eigenschaft.
- Wir speichern nur die Mindestlänge f und die Maximallänge g des Präfixes des Suffixes.
  $[f, g]_{T^D_{SA[i]\ldots n}} := \{T^D_{SA[i]\ldots SA[i]+k-1} : f \leq k \leq g\}$

Ergebnisse implizit speichern

Ergebnisse implizit speichern:

- $T^D = aaaa\#baaab\#aba\#\$$
- $(SA[i] = 6, f = 2, g = 6)$ ergibt?

Ergebnisse implizit speichern

Ergebnisse implizit speichern:

- $T^D = aaaa\#baaab\#aba\#\$$
- $(SA[i] = 6, f = 2, g = 6)$ ergibt?
- $[2, 6]_{T^D_{6\dots n}} = \{ba, baa, baaa, baaab, baaab\#\}$
- (Relevante Mengen mit Trennsymbol werden in der Nachverarbeitung beschnitten: $f = 2$, $g = 5$)

Basisalgorithmus

- Vorverarbeitung: Verkette die Originalstrings der Datenbank zu Gesamtstring T, berechne Suffixarray, LCP-Array und Informationen zum schnellen Berechnen von RMQ_{LCP}
- Berechne Array C'
- Durchlaufe den lcp-Intervall-Baum (Postorder): Für jedes ω-Intervall [l,r] berechne $freq(\omega, D) = S_D(\omega) - C_D(\omega)$
- Speichere jeden relevanten String am lexikografisch kleinsten Suffix, dessen Präfix er ist.
- Entferne alle relevanten Strings mit Trennsymbol $\#$

Speichern der relevanten Substrings

- Initialisiere Arrays der Intervallgrenzen: $f[i] = \infty$, $g[i] = 0$ für $1 \leq i \leq n$
- „Postorder-Durchlauf“ durch den virtuellen lcp-Intervallbaum
- Für jeden Knoten, der (l,r)-Intervall (ω-Intervall) darstellt:

```
if minf ≤ r − l + 1 − (C''[r] − C''[l]) ≤ maxf then
    if f[l] = ∞ then
        if l = r then
            g[l] ← n − SA[l] + 1  //Blatt
        else
            g[l] ← LCP[RMQ_LCP(l + 1, r)]  //|ω|
    f[l] ← max{LCP[l], LCP[r + 1]} + 1 //|φ_Elter| + 1
```

Wo sind wir?

- Das war der Basisalgorithmus.
- Der Basisalgorithmus wird für jede der m Datenbanken mit jeweils eigenem Parametersatz $(minf_i, maxf_i)$ ausgeführt. Ergebnis ist jeweils eine Ergebnistabelle, die die relevanten Substrings in impliziter Darstellung enthält.
- Das Gesamtergebnis ergibt sich durch Schneiden dieser Ergebnistabellen. Gesucht sind die relevanten Substrings, die in *allen* m Ergebnistabellen vorkommen.
- Weil wir eine implizite Darstellung gewählt haben, ist das nicht trivial.

Gesamtergebnis berechnen

Überblick

- Ausgabe des Basisalgorithmus ist eine Tabelle von Tupeln $(SA[i], LCP[i], f[i], g[i])$.

Gesamtergebnis berechnen

Überblick

- Ausgabe des Basisalgorithmus ist eine Tabelle von Tupeln $(SA[i], LCP[i], f[i], g[i])$.
- Wir schneiden die Ergebnistabelle L_1 von D_1 mit der Ergebnistabelle L_2 von D_2, die entstehende Ergebnistabelle mit der Ergebnistabelle von D_3 usw.

Gesamtergebnis berechnen

Überblick

- Ausgabe des Basisalgorithmus ist eine Tabelle von Tupeln $(SA[i], LCP[i], f[i], g[i])$.
- Wir schneiden die Ergebnistabelle L_1 von D_1 mit der Ergebnistabelle L_2 von D_2, die entstehende Ergebnistabelle mit der Ergebnistabelle von D_3 usw.
- Keine neue Tabelle, sondern L_1 behalten und bei Bedarf Ergebnisintervalle verkleinern. Schnitt bedeutet: Elemente entfernen, die keinen Partner in einem Ergebnisintervall in L_2 finden. Wegen Apriori-Eigenschaft keine Löcher!

Gesamtergebnis berechnen

Überblick

- Ausgabe des Basisalgorithmus ist eine Tabelle von Tupeln $(SA[i], LCP[i], f[i], g[i])$.
- Wir schneiden die Ergebnistabelle L_1 von D_1 mit der Ergebnistabelle L_2 von D_2, die entstehende Ergebnistabelle mit der Ergebnistabelle von D_3 usw.
- Keine neue Tabelle, sondern L_1 behalten und bei Bedarf Ergebnisintervalle verkleinern. Schnitt bedeutet: Elemente entfernen, die keinen Partner in einem Ergebnisintervall in L_2 finden. Wegen Apriori-Eigenschaft keine Löcher!
- Suche für Ergebnisintervall eines Suffixes aus L_1 Partner in L_2. Verwalte dazu Menge(n) von Partnerkandidaten.
- Bearbeite die Tupel der beiden Tabellen in der Reihenfolge ihrer *gemeinsamen* lexikografischen Sortierung.

Vorarbeiten 1

Vorarbeiten: Alles bitte linear in $n_1 + n_2$ ($n_1 = |T^{D_1}|$, $n_2 = |T^{D_2}|$).

- Wir berechnen Hilfsvariablen, die wir für den Schnittvorgang benötigen.
- Beim Berechnen dieser Hilfsvariablen fallen Informationen an, die wir zum Bearbeiten der Suffixe von T^{D_1} und T^{D_2} in der Reihenfolge ihrer gemeinsamen lexikografischen Sortierung benutzen.
- Reihenfolge der gemeinsamen lexikografischen Sortierung ergibt sich aus dem Ablauf des Algorithmus, keine neuen Indizes!

Vorarbeiten 2

Gemeinsame Sortierung berechnen

- Für Suffix $T^{D_2}_{i \ldots n_2}$ $(1 \leq i \leq n_2)$ ist das Suffix $T^{D_1}_{SA[p(i)] \ldots n_1}$ der lexikografisch kleinste Nachfolger aus T^{D_1}: $T^{D_1}_{SA[p(i)-1] \ldots n_1} \leq T^{D_2}_{i \ldots n_2} < T^{D_1}_{SA[p(i)] \ldots n_1}$
- $T^{D_1}_{SA[p(i)] \ldots n_1}$ ist „Anker“ von $T^{D_2}_{i \ldots n_2}$.
- Für $1 \leq j \leq n_1$ ist $c[j]$ die Anzahl der Suffixe von T^{D_2}, die in der gemeinsamen Sortierung zwischen Suffix $T^{D_1}_{SA[j] \ldots n_1}$ und dessen lexikografischem Vorgänger (in der L_1-Sortierung) $T^{D_1}_{SA[j-1] \ldots n_1}$ eingeordnet werden („Größe der Lücke“).

Vorarbeiten 2

Gemeinsame Sortierung berechnen

- Für Suffix $T^{D_2}_{i \dots n_2}$ $(1 \leq i \leq n_2)$ ist das Suffix $T^{D_1}_{SA[p(i)] \dots n_1}$ der lexikografisch kleinste Nachfolger aus T^{D_1}:
  $T^{D_1}_{SA[p(i)-1] \dots n_1} \leq T^{D_2}_{i \dots n_2} < T^{D_1}_{SA[p(i)] \dots n_1}$
- $T^{D_1}_{SA[p(i)] \dots n_1}$ ist „Anker“ von $T^{D_2}_{i \dots n_2}$.
- Für $1 \leq j \leq n_1$ ist $c[j]$ die Anzahl der Suffixe von T^{D_2}, die in der gemeinsamen Sortierung zwischen Suffix $T^{D_1}_{SA[j] \dots n_1}$ und dessen lexikografischem Vorgänger (in der L_1-Sortierung) $T^{D_1}_{SA[j-1] \dots n_1}$ eingeordnet werden („Größe der Lücke“).
- Berechnung: Sobald $p(i)$ gefunden erhöhe $c[p(i)]$ um eins.

Ablauf des Schnittvorgangs

T^{D_1}		T^{D_2}	
$T^{D_1}_{SA[j-1]\ldots n_1} = \text{abcd}$	$[1,4]$		
		abcdx	$[2,5]$
		abce	$[4,4]$
		abcex	$[\infty,0]$
		abcfe	$[4,5]$
$T^{D_1}_{SA[j]\ldots n_1} = \text{abcfx}$	$[4,5]$		

Vorarbeiten 3

Hilfsvariablen:

- Berechne für jedes Suffix von T^{D_2} die Ähnlichkeit des ähnlichsten Substrings aus T^{D_1}

Vorarbeiten 3

Hilfsvariablen:

- Berechne für jedes Suffix von T^{D_2} die Ähnlichkeit des ähnlichsten Substrings aus T^{D_1}
- Man nennt das: Matching statistics $ms(i) :=$ Länge des längsten Präfixes des Suffixes $T^{D_2}_{i \ldots n_2}$, das einem Substring von T^{D_1} entspricht.
- Naiv: Lege jedes Suffix von T^{D_2} entlang passender Pfade von der Wurzel aus in den lcp-Intervallbaum für T^{D_1}, bis es keine passende Fortsetzung mehr gibt (mismatch).

Vorarbeiten 3

Hilfsvariablen:

- Berechne für jedes Suffix von T^{D_2} die Ähnlichkeit des ähnlichsten Substrings aus T^{D_1}
- Man nennt das: Matching statistics $ms(i) :=$ Länge des längsten Präfixes des Suffixes $T^{D_2}_{i\ldots n_2}$, das einem Substring von T^{D_1} entspricht.
- Naiv: Lege jedes Suffix von T^{D_2} entlang passender Pfade von der Wurzel aus in den lcp-Intervallbaum für T^{D_1}, bis es keine passende Fortsetzung mehr gibt (mismatch).
- Die Arbeit erledigt ein cleverer Matchingalgorithmus für uns, wir lehnen uns zurück, sehen zu und erkennen den Anker des Suffixes.

Vorarbeiten 4

Wir beobachten das Matching von Suffix $T^{D_2}_{i \ldots n_2}$ und warten auf Mismatch, um p(i) zu erkennen:

Vorarbeiten 4

Wir beobachten das Matching von Suffix $T^{D_2}_{i \dots n_2}$ und warten auf Mismatch, um p(i) zu erkennen:

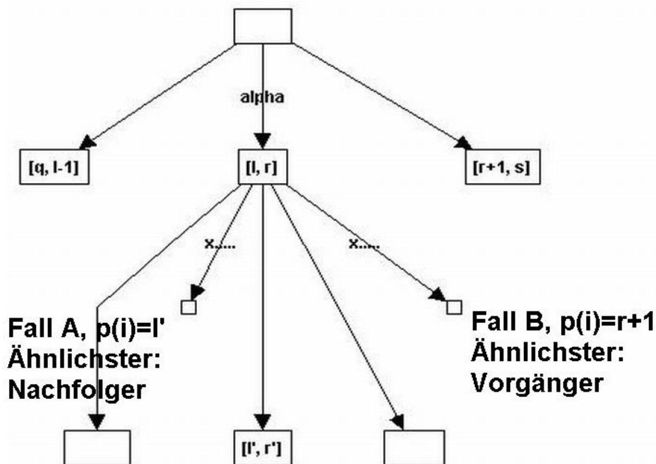
- Sei (l,r) das α-Intervall nach Matching von $\alpha\beta$, wenn beim nächsten Zeichen x der Mismatch auftritt.
- Entweder ist $|\beta| = 0$: Wir sind am Knoten
- Fall A: Es gibt ein Kind (l', r'), so dass auf der Kante zum Kind das erste Zeichen y ist mit $y > x \Rightarrow p(i) = l'$
- Fall B: Es gibt kein solches Kind: $\Rightarrow p(i) = r + 1$

Vorarbeiten 4

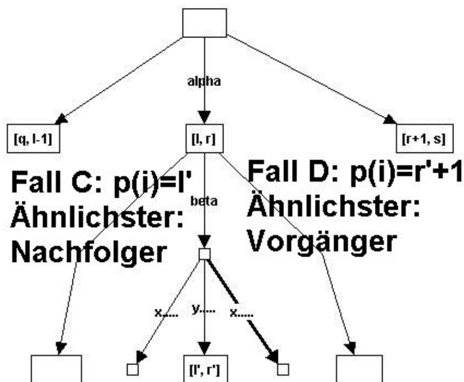
Wir beobachten das Matching von Suffix $T^{D_2}_{i \dots n_2}$ und warten auf Mismatch, um p(i) zu erkennen:

- Sei (l,r) das α-Intervall nach Matching von $\alpha\beta$, wenn beim nächsten Zeichen x der Mismatch auftritt.
- Entweder ist $|\beta| = 0$: Wir sind am Knoten
- Fall A: Es gibt ein Kind (l', r'), so dass auf der Kante zum Kind das erste Zeichen y ist mit $y > x \Rightarrow p(i) = l'$
- Fall B: Es gibt kein solches Kind: $\Rightarrow p(i) = r + 1$
- Oder es ist $|\beta| \geq 1$: Wir sind auf einer Kante, β ist Präfix einer Kantenbeschriftung zu Kind (l', r'), das nächste Zeichen auf der Kante ist y
- Fall C: $x < y \Rightarrow p(i) = l'$
- Fall D: $x > y \Rightarrow p(i) = r' + 1$

Vorarbeiten - Fälle A und B



Vorarbeiten - Fälle C und D



Vorarbeiten 5

Ähnlichkeiten zwischen Suffixen von T^{D_1} und T^{D_2}

- Wie man leicht sieht, gilt für Fall B und D:
  $lcp(T^{D_1}_{SA[p(i)-1]\dots n_1}, T^{D_2}_{i\dots n_2}) = ms(i)$
  $lcp(T^{D_2}_{i\dots n_2}, T^{D_1}_{SA[p(i)]\dots n_1}) = LCP[p(i)]$
- Sowie für Fall A und C (auch am linken Intervallrand):
  $lcp(T^{D_1}_{SA[p(i)-1]\dots n_1}, T^{D_2}_{i\dots n_2}) = LCP[p(i)]$
  $lcp(T^{D_2}_{i\dots n_2}, T^{D_1}_{SA[p(i)]\dots n_1}) = ms(i)$
- Welcher Fall vorliegt, können wir im Vorzeichenbit von $ms(i)$ speichern.

Partnersuche: Wer kommt in Frage?

Lemma: Sei $p(i)$ so definiert, dass $T^{D_1}_{SA[p(i)-1]\dots n_1} \leq T^{D_2}_{i\dots n_2} < T^{D_1}_{SA[p(i)]\dots n_1}$. Suffix $T^{D_2}_{i\dots n_2} (1 \leq i \leq n_2)$ kann nur gemeinsame relevante Strings mit Suffixen an Positionen $\leq p(i)$ im Suffixarray von T^{D_1} haben.

- Wenn wir in absteigender lexikografischer Reihenfolge die Suffixe T^{D_1} bearbeiten, sind wir mit $T^{D_1}_{SA[p(i)]\dots n_1}$ fertig, sobald wir $T^{D_1}_{SA[p(i-1)]\dots n_1}$ erreichen.
- Berechne gleichzeitig alle Informationen, die unsere lexikografischen Vorgänger über diese gerade bearbeiteten Suffixe aus T^{D_2} benötigen $\rightarrow$ zwei Kandidatenmengen, eine für $T^{D_1}_{SA[p(i)]\dots n_1}$, eine für alle Vorgänger.

Partnersuche: Wer kommt in Frage?

- Wir suchen Partner für die relevanten Substrings von T^{D_1}, die in einem Ergebnisintervall an Suffix $T^{D_1}_{SA[p(i)]\dots n_1}$ gespeichert werden: $[f,g]_{T^{D_1}_{SA[p(i)]\dots n_1}} = \{T^{D_1}_{SA[p(i)]\dots SA[p(i)]+k-1} : f \leq k \leq g\}$. Diese Partner sind relevante Substrings von T^{D_2}, die wir in einem Ergebnisintervall $[f_{cur}, g_{cur}]_{T^{D_1}_{SA[p(i)]\dots n_1}}$ verwalten.
- Zusätzlich suchen wir vorsorglich Partner für die relevanten Substrings von T^{D_1}, die in Ergebnisintervallen an Suffixen $T^{D_1}_{m\dots n_1}$ mit $T^{D_1}_{m\dots n_1} < T^{D_1}_{SA[p(i)]\dots n_1}$ gespeichert werden. Diese Partner sind relevante Substrings von T^{D_2}, die wir in einem Ergebnisintervall $[f_{prev}, g_{prev}]_{T^{D_1}_{SA[p(i)-1]\dots n_1}}$ an Suffix $T^{D_1}_{SA[p(i)-1]\dots n_1}$ verwalten.

Der Schnittvorgang: Bezeichnungen

Notation Pseudocode:

- Wir haben zwei Ergebnislisten L_1 und L_2, Einträge sind Tupel $(SA[i], LCP[i], f[i], g[i])$
- p_1: Position in der lexikografisch sortierten Ergebnisliste L_1
- p_2: Position in der lexikografisch sortierten Ergebnisliste L_2
- Zugriff auf die Tupel in der Ergebnisliste: $L_i[p_i].f = f[p_i]$ aus $L_i, 1 \leq i \leq 2$, andere Einträge analog.

Weitergabe der Kandidatenmengen

Wir betrachten die Abarbeitung des Tupels aus $L_1[p_1]$ und der Tupel aus L_2, die zwischen $L_1[p_1]$ und $L_1[p_1 - 1]$ liegen. Zunächst übernehmen wir die Kandidatenmengen des Nachfolgers (der bereits bearbeitet wurde).

Algorithm 2: Fragment - Weitergabe der Kandidatenmengen

$f_{cur} = f_{prev}, g_{cur} = g_{prev}$
if $L_1[p_1].lcp < f_{cur}$ **then**
  $f_{prev} = \infty, g_{prev} = 0$
else
  $f_{prev} = f_{cur}, g_{prev} = min\{g_{cur}, L_1[p_1].lcp\}$

Der Schnittvorgang

Algorithm 3: Fragment - Erweitern der Kandidatenmengen

if $L_2[p_2].f \leq L_2[p_2].g$ **then**
- **if** $lcp_1 \geq L_2[p_2].f$ **then**
  - $f_{prev} = min\{f_{prev}, L_2[p_2].f\}$
  - $g_{prev} = max\{g_{prev}, min\{lcp_1, L_2[p_2].g\}\}$
- **if** $lcp_2 \geq L_2[p_2].f$ **then**
  - $f_{cur} = min\{f_{cur}, L_2[p_2].f\}$
  - $g_{cur} = max\{g_{cur}, min\{lcp_2, L_2[p_2].g\}\}$

$$lcp_1 := lcp(T^{D_2}_{SA_{D_2}[p_2]\dots n_2}, T^{D_1}_{SA_{D_1}[p(SA_{D_2}[p_2])-1]\dots n_1})$$
$$lcp_2 := lcp(T^{D_2}_{SA_{D_2}[p_2]\dots n_2}, T^{D_1}_{SA_{D_1}[p(SA_{D_2}[p_2])]\dots n_1})$$

Der Schnittvorgang

Algorithm 4: Fragment - Der Schnitt

$L_{out}[p_1].f = max\{f_{cur}, L_1[p_1].f\}$
$L_{out}[p_1].g = min\{g_{cur}, L_1[p_1].g\}$

Ergebnisse

Kurze Zusammenfassung der Ergebnisse

- Alles wie erwartet:
- Speicherbedarf nur noch von der größten Datenbank abhängig, Konstanten ähnlich.
- Speicherersparnis steigt mit Anzahl der Datenbanken.
- Aber: Etwa verdoppelte Laufzeit.