

Algorithmische Aspekte von Kommunikationsnetzen

IP Prefix Lookup

Lukáš Ličko

Hauptseminar AFS
Institut für Theoretische Informatik
TU Ilmenau

25.Juni 2009

Gliederung

- 1 IP Prefix Lookup mit einem Trie
 - Einleitung & Definitionen
 - Konstruktion, Speicherplatzbedarf, Suchen, Aktualisieren
- 2 IP Prefix Lookup durch Binärsuche nach Präfixlängen
 - Einführung von Längenklassen
 - Binärsuche, Speicherplatzbedarf, Aktualisieren
- 3 Prefix-Expansion für schnelleren IP Prefix Lookup
 - Expansion von Prefixen
 - Minimierung des Speicherplatzes
 - Anhang

Destination-based routing

- anhand einer Zieladresse z wird ein Nachbarknoten gewählt
- Routing-Tabelle R auf jedem Knoten (Router)
- Adresse ist ein Binärstring der Länge 32 (128)
Annahme: Adressen sind Binärstrings der Länge W
- Tabelleneinträge: (x, y)
 x -Binärstring der Länge höchstens W ; y -ausgehender Link
- IP Prefix Lookup: für ein Paket mit Zieladresse z wird (x, y) in R gesucht, wobei x das längste passende Präfix (BMP) von z ist

Destination-based routing

- anhand einer Zieladresse z wird ein Nachbarknoten gewählt
- Routing-Tabelle R auf jedem Knoten (Router)
- Adresse ist ein Binärstring der Länge 32 (128)
Annahme: Adressen sind Binärstrings der Länge W
- Tabelleneinträge: (x, y)
 x -Binärstring der Länge höchstens W ; y -ausgehender Link
- IP Prefix Lookup: für ein Paket mit Zieladresse z wird (x, y) in R gesucht, wobei x das längste passende Präfix (BMP) von z ist

Destination-based routing

- anhand einer Zieladresse z wird ein Nachbarknoten gewählt
- Routing-Tabelle R auf jedem Knoten (Router)
- Adresse ist ein Binärstring der Länge 32 (128)
Annahme: Adressen sind Binärstrings der Länge W
- Tabelleneinträge: (x, y)
 x - Binärstring der Länge höchstens W ; y - ausgehender Link
- IP Prefix Lookup: für ein Paket mit Zieladresse z wird (x, y) in R gesucht, wobei x das längste passende Präfix (BMP) von z ist

Destination-based routing

- anhand einer Zieladresse z wird ein Nachbarknoten gewählt
- Routing-Tabelle R auf jedem Knoten (Router)
- Adresse ist ein Binärstring der Länge 32 (128)
Annahme: Adressen sind Binärstrings der Länge W
- Tabelleneinträge: (x, y)
 x -Binärstring der Länge höchstens W ; y -ausgehender Link
- IP Prefix Lookup: für ein Paket mit Zieladresse z wird (x, y) in R gesucht, wobei x das längste passende Präfix (BMP) von z ist

Destination-based routing

- anhand einer Zieladresse z wird ein Nachbarknoten gewählt
- Routing-Tabelle R auf jedem Knoten (Router)
- Adresse ist ein Binärstring der Länge 32 (128)
Annahme: Adressen sind Binärstrings der Länge W
- Tabelleneinträge: (x, y)
 x -Binärstring der Länge höchstens W ; y -ausgehender Link
- IP Prefix Lookup: für ein Paket mit Zieladresse z wird (x, y) in R gesucht, wobei x das längste passende Präfix (BMP) von z ist

Beispiel

Sei $W=5$. Einträge in Routing-Tabelle: $(0, y_1)$, $(001, y_2)$, $(10, y_1)$, $(110, y_3)$, $(111, y_2)$.

Für Paket mit Zieladresse $z = 00100$ ist $(001, y_2)$ BMP und für $z = 01000$ ist $(0, y_1)$ BMP.

Routing-Tabelle als Trie

- Trie ist ein Suchbaum (V, E)
- Kantenbeschriftungen sind Binärstrings
- Knoten repräsentiert einen String x , der sich durch Konkatenation der Kantenbeschriftungen auf dem Pfad von der Wurzel zu dem Knoten ergibt
- in dem Knoten ist der Link y gespeichert
- jeder Knoten hat höchstens 2 Kinder. Kantenbeschriftung zum ersten Kind beginnt mit 0, zum zweiten Kind mit 1

Routing-Tabelle als Trie

- Trie ist ein Suchbaum (V, E)
- Kantenbeschriftungen sind Binärstrings
- Knoten repräsentiert einen String x , der sich durch Konkatenation der Kantenbeschriftungen auf dem Pfad von der Wurzel zu dem Knoten ergibt
- in dem Knoten ist der Link y gespeichert
- jeder Knoten hat höchstens 2 Kinder. Kantenbeschriftung zum ersten Kind beginnt mit 0, zum zweiten Kind mit 1

Routing-Tabelle als Trie

- Trie ist ein Suchbaum (V, E)
- Kantenbeschriftungen sind Binärstrings
- Knoten repräsentiert einen String x , der sich durch Konkatenation der Kantenbeschriftungen auf dem Pfad von der Wurzel zu dem Knoten ergibt
- in dem Knoten ist der Link y gespeichert
- jeder Knoten hat höchstens 2 Kinder. Kantenbeschriftung zum ersten Kind beginnt mit 0, zum zweiten Kind mit 1

Routing-Tabelle als Trie

- Trie ist ein Suchbaum (V, E)
- Kantenbeschriftungen sind Binärstrings
- Knoten repräsentiert einen String x , der sich durch Konkatenation der Kantenbeschriftungen auf dem Pfad von der Wurzel zu dem Knoten ergibt
- in dem Knoten ist der Link y gespeichert
- jeder Knoten hat höchstens 2 Kinder. Kantenbeschriftung zum ersten Kind beginnt mit 0, zum zweiten Kind mit 1

Routing-Tabelle als Trie

- Trie ist ein Suchbaum (V, E)
- Kantenbeschriftungen sind Binärstrings
- Knoten repräsentiert einen String x , der sich durch Konkatenation der Kantenbeschriftungen auf dem Pfad von der Wurzel zu dem Knoten ergibt
- in dem Knoten ist der Link y gespeichert
- jeder Knoten hat höchstens 2 Kinder. Kantenbeschriftung zum ersten Kind beginnt mit 0, zum zweiten Kind mit 1

Beispiel

Routing Tabelle als Trie mit den Einträgen $(0, y_1)$, $(001, y_2)$,
 $(10, y_1)$, $(110, y_3)$, $(111, y_2)$
-> Tafel

Satz

Mittels eines Trie lässt sich eine Routing-Tabelle in Speicherplatz $O(N)$ so speichern, dass jede Suche und jede Aktualisierung in Zeit $O(W)$ durchgeführt werden kann. Der Zeitbedarf für die Konstruktion des Trie ist $O(N \cdot W)$.

- N - Anzahl der Einträge in R

Satz

Mittels eines Trie lässt sich eine Routing-Tabelle in Speicherplatz $O(N)$ so speichern, dass jede Suche und jede Aktualisierung in Zeit $O(W)$ durchgeführt werden kann. Der Zeitbedarf für die Konstruktion des Trie ist $O(N \cdot W)$.

- N - Anzahl der Einträge in R

Konstruktion des Trie

- Am Anfang: leerer Trie mit Wurzel
- für $\forall (x,y) \in R$: beginne an der Wurzel und folge dem Pfad abwärts, der durch x festgelegt wird
- ① Der Pfad endet bei einem Trie-Knoten v , wenn x ganz abgearbeitet ist
- ② Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , wenn x ganz abgearbeitet ist
- ③ Der Pfad endet bei einem Trie-Knoten v , bevor x ganz abgearbeitet ist
- ④ Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , bevor x ganz abgearbeitet ist
- Laufzeit für jeden Eintrag: $O(W)$, Gesamtlaufzeit: $O(N \cdot W)$

Konstruktion des Trie

- Am Anfang: leerer Trie mit Wurzel
- für $\forall (x,y) \in R$: beginne an der Wurzel und folge dem Pfad abwärts, der durch x festgelegt wird
- ① Der Pfad endet bei einem Trie-Knoten v , wenn x ganz abgearbeitet ist
- ② Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , wenn x ganz abgearbeitet ist
- ③ Der Pfad endet bei einem Trie-Knoten v , bevor x ganz abgearbeitet ist
- ④ Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , bevor x ganz abgearbeitet ist
- Laufzeit für jeden Eintrag: $O(W)$, Gesamtlaufzeit: $O(N \cdot W)$

Konstruktion des Trie

- Am Anfang: leerer Trie mit Wurzel
- für $\forall (x,y) \in R$: beginne an der Wurzel und folge dem Pfad abwärts, der durch x festgelegt wird
- ① Der Pfad endet bei einem Trie-Knoten v , wenn x ganz abgearbeitet ist
- ② Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , wenn x ganz abgearbeitet ist
- ③ Der Pfad endet bei einem Trie-Knoten v , bevor x ganz abgearbeitet ist
- ④ Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , bevor x ganz abgearbeitet ist
- Laufzeit für jeden Eintrag: $O(W)$, Gesamtlaufzeit: $O(N \cdot W)$

Konstruktion des Trie

- Am Anfang: leerer Trie mit Wurzel
- für $\forall (x,y) \in R$: beginne an der Wurzel und folge dem Pfad abwärts, der durch x festgelegt wird
- ① Der Pfad endet bei einem Trie-Knoten v , wenn x ganz abgearbeitet ist
- ② Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , wenn x ganz abgearbeitet ist
- ③ Der Pfad endet bei einem Trie-Knoten v , bevor x ganz abgearbeitet ist
- ④ Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , bevor x ganz abgearbeitet ist
- Laufzeit für jeden Eintrag: $O(W)$, Gesamtlaufzeit: $O(N \cdot W)$

Konstruktion des Trie

- Am Anfang: leerer Trie mit Wurzel
- für $\forall (x,y) \in R$: beginne an der Wurzel und folge dem Pfad abwärts, der durch x festgelegt wird
- ① Der Pfad endet bei einem Trie-Knoten v , wenn x ganz abgearbeitet ist
- ② Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , wenn x ganz abgearbeitet ist
- ③ Der Pfad endet bei einem Trie-Knoten v , bevor x ganz abgearbeitet ist
- ④ Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , bevor x ganz abgearbeitet ist
- Laufzeit für jeden Eintrag: $O(W)$, Gesamtlaufzeit: $O(N \cdot W)$

Konstruktion des Trie

- Am Anfang: leerer Trie mit Wurzel
- für $\forall (x,y) \in R$: beginne an der Wurzel und folge dem Pfad abwärts, der durch x festgelegt wird
- ① Der Pfad endet bei einem Trie-Knoten v , wenn x ganz abgearbeitet ist
- ② Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , wenn x ganz abgearbeitet ist
- ③ Der Pfad endet bei einem Trie-Knoten v , bevor x ganz abgearbeitet ist
- ④ Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , bevor x ganz abgearbeitet ist
- Laufzeit für jeden Eintrag: $O(W)$, Gesamtlaufzeit: $O(N \cdot W)$

Konstruktion des Trie

- Am Anfang: leerer Trie mit Wurzel
- für $\forall (x,y) \in R$: beginne an der Wurzel und folge dem Pfad abwärts, der durch x festgelegt wird
- ① Der Pfad endet bei einem Trie-Knoten v , wenn x ganz abgearbeitet ist
- ② Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , wenn x ganz abgearbeitet ist
- ③ Der Pfad endet bei einem Trie-Knoten v , bevor x ganz abgearbeitet ist
- ④ Der Pfad endet auf einer Kante (v, w) mit Beschriftung $\alpha\beta$ zwischen α und β , bevor x ganz abgearbeitet ist
- Laufzeit für jeden Eintrag: $O(W)$, Gesamtlaufzeit: $O(N \cdot W)$

Speicherplatzbedarf des Trie

- jeder Knoten und jede Kante benötigt konstanten Speicherplatz
- N Knoten repräsentieren die Einträge der Routing-Tabelle, alle anderen sind innere Knoten mit genau zwei Kindern
- ein Baum mit höchstens N Blättern hat höchstens $N-1$ innere Knoten mit zwei Kindern
- \Rightarrow ein Trie enthält höchstens $2N-1$ Knoten und $2N-2$ Kanten
- Speicherplatzbedarf ist linear: $O(N)$

Speicherplatzbedarf des Trie

- jeder Knoten und jede Kante benötigt konstanten Speicherplatz
- N Knoten repräsentieren die Einträge der Routing-Tabelle, alle anderen sind innere Knoten mit genau zwei Kindern
- ein Baum mit höchstens N Blättern hat höchstens $N-1$ innere Knoten mit zwei Kindern
- \Rightarrow ein Trie enthält höchstens $2N-1$ Knoten und $2N-2$ Kanten
- Speicherplatzbedarf ist linear: $O(N)$

Speicherplatzbedarf des Trie

- jeder Knoten und jede Kante benötigt konstanten Speicherplatz
- N Knoten repräsentieren die Einträge der Routing-Tabelle, alle anderen sind innere Knoten mit genau zwei Kindern
- ein Baum mit höchstens N Blättern hat höchstens $N-1$ innere Knoten mit zwei Kindern
- \Rightarrow ein Trie enthält höchstens $2N-1$ Knoten und $2N-2$ Kanten
- Speicherplatzbedarf ist linear: $O(N)$

Speicherplatzbedarf des Trie

- jeder Knoten und jede Kante benötigt konstanten Speicherplatz
- N Knoten repräsentieren die Einträge der Routing-Tabelle, alle anderen sind innere Knoten mit genau zwei Kindern
- ein Baum mit höchstens N Blättern hat höchstens $N-1$ innere Knoten mit zwei Kindern
- \Rightarrow ein Trie enthält höchstens $2N-1$ Knoten und $2N-2$ Kanten
- Speicherplatzbedarf ist linear: $O(N)$

Speicherplatzbedarf des Trie

- jeder Knoten und jede Kante benötigt konstanten Speicherplatz
- N Knoten repräsentieren die Einträge der Routing-Tabelle, alle anderen sind innere Knoten mit genau zwei Kindern
- ein Baum mit höchstens N Blättern hat höchstens $N-1$ innere Knoten mit zwei Kindern
- \Rightarrow ein Trie enthält höchstens $2N-1$ Knoten und $2N-2$ Kanten
- Speicherplatzbedarf ist linear: $O(N)$

Suchen im Trie

- beginne bei der Wurzel
- folge dem Pfad, der durch die Zieladresse z bestimmt ist
- merke sich immer den letzten besuchten markierten Knoten
- stopp, wenn z abgearbeitet ist oder wenn der Pfad sich nicht um das nächste Bit von z verlängern kann
- der letzte gemerkte Knoten repräsentiert das BMP
- Laufzeit: $O(W)$

Suchen im Trie

- beginne bei der Wurzel
- folge dem Pfad, der durch die Zieladresse z bestimmt ist
- merke sich immer den letzten besuchten markierten Knoten
- stopp, wenn z abgearbeitet ist oder wenn der Pfad sich nicht um das nächste Bit von z verlängern kann
- der letzte gemerkte Knoten repräsentiert das BMP
- Laufzeit: $O(W)$

Suchen im Trie

- beginne bei der Wurzel
- folge dem Pfad, der durch die Zieladresse z bestimmt ist
- merke sich immer den letzten besuchten markierten Knoten
- stopp, wenn z abgearbeitet ist oder wenn der Pfad sich nicht um das nächste Bit von z verlängern kann
- der letzte gemerkte Knoten repräsentiert das BMP
- Laufzeit: $O(W)$

Suchen im Trie

- beginne bei der Wurzel
- folge dem Pfad, der durch die Zieladresse z bestimmt ist
- merke sich immer den letzten besuchten markierten Knoten
- stopp, wenn z abgearbeitet ist oder wenn der Pfad sich nicht um das nächste Bit von z verlängern kann
- der letzte gemerkte Knoten repräsentiert das BMP
- Laufzeit: $O(W)$

Suchen im Trie

- beginne bei der Wurzel
- folge dem Pfad, der durch die Zieladresse z bestimmt ist
- merke sich immer den letzten besuchten markierten Knoten
- stopp, wenn z abgearbeitet ist oder wenn der Pfad sich nicht um das nächste Bit von z verlängern kann
- der letzte gemerkte Knoten repräsentiert das BMP
- Laufzeit: $O(W)$

Suchen im Trie

- beginne bei der Wurzel
- folge dem Pfad, der durch die Zieladresse z bestimmt ist
- merke sich immer den letzten besuchten markierten Knoten
- stopp, wenn z abgearbeitet ist oder wenn der Pfad sich nicht um das nächste Bit von z verlängern kann
- der letzte gemerkte Knoten repräsentiert das BMP
- Laufzeit: $O(W)$

Aktualisieren des Trie

- Einfügen und Löschen des Eintrags (x,y) oder Link y ändern
- Lokalisierung der richtigen Stelle, des richtigen Knotens
- Laufzeit: $O(W)$

Aktualisieren des Trie

- Einfügen und Löschen des Eintrags (x,y) oder Link y ändern
- Lokalisierung der richtigen Stelle, des richtigen Knotens
- Laufzeit: $O(W)$

Aktualisieren des Trie

- Einfügen und Löschen des Eintrags (x,y) oder Link y ändern
- Lokalisierung der richtigen Stelle, des richtigen Knotens
- Laufzeit: $O(W)$

Längenklassen

- Ziel: Laufzeitverbesserung von $O(W)$ auf $O(\log W)$
- Idee: wenn alle Einträge von R dieselbe Länge haben, benötigt man für IP Prefix Lookup konstante Zeit
- Einteilung der Einträge anhand ihrer Länge in höchstens W Klassen
- Längenkategorie L_i speichert alle Einträge (x, y) , bei denen das Präfix x die Länge i hat
- sei N_i Anzahl solcher Einträge, dann benötigt man zur Speicherung von L_i Speicherplatz $O(N_i)$ und Zeit $O(1)$ zum Entscheiden, ob (k, y) in L_i enthalten ist \rightarrow Hash-Tabelle

Längenklassen

- Ziel: Laufzeitverbesserung von $O(W)$ auf $O(\log W)$
- Idee: wenn alle Einträge von R dieselbe Länge haben, benötigt man für IP Prefix Lookup konstante Zeit
- Einteilung der Einträge anhand ihrer Länge in höchstens W Klassen
- Längenkategorie L_i speichert alle Einträge (x, y) , bei denen das Präfix x die Länge i hat
- sei N_i Anzahl solcher Einträge, dann benötigt man zur Speicherung von L_i Speicherplatz $O(N_i)$ und Zeit $O(1)$ zum Entscheiden, ob (k, y) in L_i enthalten ist \rightarrow Hash-Tabelle

Längenklassen

- Ziel: Laufzeitverbesserung von $O(W)$ auf $O(\log W)$
- Idee: wenn alle Einträge von R dieselbe Länge haben, benötigt man für IP Prefix Lookup konstante Zeit
- Einteilung der Einträge anhand ihrer Länge in höchstens W Klassen
- Längenkategorie L_i speichert alle Einträge (x, y) , bei denen das Präfix x die Länge i hat
- sei N_i Anzahl solcher Einträge, dann benötigt man zur Speicherung von L_i Speicherplatz $O(N_i)$ und Zeit $O(1)$ zum Entscheiden, ob (k, y) in L_i enthalten ist \rightarrow Hash-Tabelle

Längenklassen

- Ziel: Laufzeitverbesserung von $O(W)$ auf $O(\log W)$
- Idee: wenn alle Einträge von R dieselbe Länge haben, benötigt man für IP Prefix Lookup konstante Zeit
- Einteilung der Einträge anhand ihrer Länge in höchstens W Klassen
- Längenkategorie L_i speichert alle Einträge (x, y) , bei denen das Präfix x die Länge i hat
- sei N_i Anzahl solcher Einträge, dann benötigt man zur Speicherung von L_i Speicherplatz $O(N_i)$ und Zeit $O(1)$ zum Entscheiden, ob (k, y) in L_i enthalten ist \rightarrow Hash-Tabelle

Längenklassen

- Ziel: Laufzeitverbesserung von $O(W)$ auf $O(\log W)$
- Idee: wenn alle Einträge von R dieselbe Länge haben, benötigt man für IP Prefix Lookup konstante Zeit
- Einteilung der Einträge anhand ihrer Länge in höchstens W Klassen
- Längenkategorie L_i speichert alle Einträge (x, y) , bei denen das Präfix x die Länge i hat
- sei N_i Anzahl solcher Einträge, dann benötigt man zur Speicherung von L_i Speicherplatz $O(N_i)$ und Zeit $O(1)$ zum Entscheiden, ob (k, y) in L_i enthalten ist \rightarrow Hash-Tabelle

Binärsuche

- Suche in einer Längenkategorie: $O(1)$
- Suche nach BMP für eine gegebene Zieladresse z
- beginne in 'mittlerer' Längenkategorie, falls
- ① erfolgreich, suche bei längeren Prefixen
- ② sonst, suche bei kürzeren Prefixen
- höchstens W Längenkategorien \Rightarrow Laufzeit $O(\log W)$

Binärsuche

- Suche in einer Längenkategorie: $O(1)$
- Suche nach BMP für eine gegebene Zieladresse z
- beginne in 'mittlerer' Längenkategorie, falls
- ① erfolgreich, suche bei längeren Prefixen
- ② sonst, suche bei kürzeren Prefixen
- höchstens W Längenkategorien \Rightarrow Laufzeit $O(\log W)$

Binärsuche

- Suche in einer Längerklasse: $O(1)$
- Suche nach BMP für eine gegebene Zieladresse z
- beginne in 'mittlerer' Längerklasse, falls
 - 1 erfolgreich, suche bei längeren Prefixen
 - 2 sonst, suche bei kürzeren Prefixen
- höchstens W Längerklassen \Rightarrow Laufzeit $O(\log W)$

Binärsuche

- Suche in einer Längerkategorie: $O(1)$
- Suche nach BMP für eine gegebene Zieladresse z
- beginne in 'mittlerer' Längerkategorie, falls
- ① erfolgreich, suche bei längeren Prefixen
- ② sonst, suche bei kürzeren Prefixen
- höchstens W Längerklassen \Rightarrow Laufzeit $O(\log W)$

Binärsuche

- Suche in einer Längenkategorie: $O(1)$
- Suche nach BMP für eine gegebene Zieladresse z
- beginne in 'mittlerer' Längenkategorie, falls
- ① erfolgreich, suche bei längeren Prefixen
- ② sonst, suche bei kürzeren Prefixen
- höchstens W Längenkategorien \Rightarrow Laufzeit $O(\log W)$

Binärsuche

- Suche in einer Längenkategorie: $O(1)$
- Suche nach BMP für eine gegebene Zieladresse z
- beginne in 'mittlerer' Längenkategorie, falls
- ① erfolgreich, suche bei längeren Prefixen
- ② sonst, suche bei kürzeren Prefixen
- höchstens W Längenkategorien \Rightarrow Laufzeit $O(\log W)$

So einfach ist es nicht

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00\}$, $L_3 = \{111\}$

$z=11101$

Suche in L_2 erfolglos, weitersuchen in L_1 , obwohl BMP in L_3 ist.

Lösung

- Marker in Längenklassen einfügen
- sei x ein Prefix der Länge i und sei $x[1..j]$ das Prefix der ersten j Zeichen von x mit $1 \leq j \leq i$
- wenn x in Längenkategorie L_i , dann speichere in $\forall L_j$ für $j < i$, die $x[1..j]$ nicht enthalten, einen Marker für $x[1..j]$

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00, 11\}$, $L_3 = \{111\}$

$z=11101$

Einfügen von Marker 11 in L_2 . Suche in L_2 erfolgreich, weitersuchen in L_3 .

Lösung

- Marker in Längenklassen einfügen
- sei x ein Prefix der Länge i und sei $x[1..j]$ das Prefix der ersten j Zeichen von x mit $1 \leq j \leq i$
- wenn x in Längenkategorie L_i , dann speichere in $\forall L_j$ für $j < i$, die $x[1..j]$ nicht enthalten, einen Marker für $x[1..j]$

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00, 11\}$, $L_3 = \{111\}$

$z=11101$

Einfügen von Marker 11 in L_2 . Suche in L_2 erfolgreich, weitersuchen in L_3 .

Lösung

- Marker in Längenklassen einfügen
- sei x ein Prefix der Länge i und sei $x[1..j]$ das Prefix der ersten j Zeichen von x mit $1 \leq j \leq i$
- wenn x in Längenkategorie L_i , dann speichere in $\forall L_j$ für $j < i$, die $x[1..j]$ nicht enthalten, einen Marker für $x[1..j]$

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00, 11\}$, $L_3 = \{111\}$

$z=11101$

Einfügen von Marker 11 in L_2 . Suche in L_2 erfolgreich, weitersuchen in L_3 .

Lösung

- Marker in Längenklassen einfügen
- sei x ein Prefix der Länge i und sei $x[1..j]$ das Prefix der ersten j Zeichen von x mit $1 \leq j \leq i$
- wenn x in Längenkategorie L_i , dann speichere in $\forall L_j$ für $j < i$, die $x[1..j]$ nicht enthalten, einen Marker für $x[1..j]$

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00, 11\}$, $L_3 = \{111\}$

$z=11101$

Einfügen von Marker 11 in L_2 . Suche in L_2 erfolgreich, weitersuchen in L_3 .

So einfach ist es nicht

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00, 11\}$, $L_3 = \{111\}$

$z=11010$

Marker 11 in L_2 . Weitersuche in L_3 erfolglos, 11 in L_2 wäre das BMP, obwohl BMP ist 1 in L_1 und 11 nur ein Marker.

Lösung

- um die Laufzeit $O(\log W)$ zu erreichen ist backtracking (Weitersuche in L_1) nicht möglich
- speichere bei jedem Marker x einen Verweis $x.BMP$ ab
- Verweis auf $(x',y') \in R$, x' ist Prefix von x mit maximaler Länge

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00, 11\}$, $L_3 = \{111\}$

$z=11010$

Marker 11 in L_2 mit dem Verweis 11.BMP auf den Eintrag für das Prefix 1.

Lösung

- um die Laufzeit $O(\log W)$ zu erreichen ist backtracking (Weitersuche in L_1) nicht möglich
- speichere bei jedem Marker x einen Verweis x .BMP ab
- Verweis auf $(x', y') \in R$, x' ist Prefix von x mit maximaler Länge

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00, 11\}$, $L_3 = \{111\}$

$z = 11010$

Marker 11 in L_2 mit dem Verweis 11.BMP auf den Eintrag für das Prefix 1.

Lösung

- um die Laufzeit $O(\log W)$ zu erreichen ist backtracking (Weitersuche in L_1) nicht möglich
- speichere bei jedem Marker x einen Verweis x .BMP ab
- Verweis auf $(x', y') \in R$, x' ist Prefix von x mit maximaler Länge

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00, 11\}$, $L_3 = \{111\}$

$z = 11010$

Marker 11 in L_2 mit dem Verweis 11.BMP auf den Eintrag für das Prefix 1.

Lösung

- um die Laufzeit $O(\log W)$ zu erreichen ist backtracking (Weitersuche in L_1) nicht möglich
- speichere bei jedem Marker x einen Verweis x .BMP ab
- Verweis auf $(x', y') \in R$, x' ist Prefix von x mit maximaler Länge

Beispiel

Routing-Tabelle mit Einträgen für Prefixe 0, 1, 00 und 111.

Längenklassen $L_1 = \{0, 1\}$, $L_2 = \{00, 11\}$, $L_3 = \{111\}$

$z = 11010$

Marker 11 in L_2 mit dem Verweis 11.BMP auf den Eintrag für das Prefix 1.

Speicherplatzbedarf

- eine Tabelle für L_j benötigt $O(N_j)$
- insgesamt $O(N)$, ohne Marker
- bis zu $O(W)$ Marker pro Prefix (in allen L_j mit $j < i$),
Speicherplatzbedarf für gesamte Datenstruktur $O(N \cdot W)$
- $O(\log W)$ Marker, wenn Marker in Längenklassen, wo sie
tatsächlich nötig sind, Speicherplatzbedarf für gesamte
Datenstruktur $O(N \cdot \log W)$

Speicherplatzbedarf

- eine Tabelle für L_j benötigt $O(N_j)$
- insgesamt $O(N)$, ohne Marker
- bis zu $O(W)$ Marker pro Prefix (in allen L_j mit $j < i$),
Speicherplatzbedarf für gesamte Datenstruktur $O(N \cdot W)$
- $O(\log W)$ Marker, wenn Marker in Längenklassen, wo sie
tatsächlich nötig sind, Speicherplatzbedarf für gesamte
Datenstruktur $O(N \cdot \log W)$

Speicherplatzbedarf

- eine Tabelle für L_j benötigt $O(N_j)$
- insgesamt $O(N)$, ohne Marker
- bis zu $O(W)$ Marker pro Prefix (in allen L_j mit $j < i$),
Speicherplatzbedarf für gesamte Datenstruktur $O(N \cdot W)$
- $O(\log W)$ Marker, wenn Marker in Längerklassen, wo sie
tatsächlich nötig sind, Speicherplatzbedarf für gesamte
Datenstruktur $O(N \cdot \log W)$

Speicherplatzbedarf

- eine Tabelle für L_j benötigt $O(N_j)$
- insgesamt $O(N)$, ohne Marker
- bis zu $O(W)$ Marker pro Prefix (in allen L_j mit $j < i$),
Speicherplatzbedarf für gesamte Datenstruktur $O(N \cdot W)$
- $O(\log W)$ Marker, wenn Marker in Längerklassen, wo sie
tatsächlich nötig sind, Speicherplatzbedarf für gesamte
Datenstruktur $O(N \cdot \log W)$

Aktualisieren der Datenstruktur

- Eintrag (x, y) in L_i , ggf. Marker x' in L_j für $j < i$ und Verweise $x'.BMP$ in L_j mit $j > i$ einfügen
- potentiell sehr teuer, da viele Marker aktualisiert werden müssen
- keine gute Zeitschranke für Einfügen und Löschen
- besser: Änderungen über längere Zeit sammeln und dann die Datenstruktur komplett neu aufbauen
- Zeit für die Konstruktion ist linear in ihrer Größe (mittels Hash-Tabellen)

Aktualisieren der Datenstruktur

- Eintrag (x, y) in L_i , ggf. Marker x' in L_j für $j < i$ und Verweise $x'.BMP$ in L_j mit $j > i$ einfügen
- potentiell sehr teuer, da viele Marker aktualisiert werden müssen
- keine gute Zeitschranke für Einfügen und Löschen
- besser: Änderungen über längere Zeit sammeln und dann die Datenstruktur komplett neu aufbauen
- Zeit für die Konstruktion ist linear in ihrer Größe (mittels Hash-Tabellen)

Aktualisieren der Datenstruktur

- Eintrag (x, y) in L_i , ggf. Marker x' in L_j für $j < i$ und Verweise $x'.BMP$ in L_j mit $j > i$ einfügen
- potentiell sehr teuer, da viele Marker aktualisiert werden müssen
- keine gute Zeitschranke für Einfügen und Löschen
- besser: Änderungen über längere Zeit sammeln und dann die Datenstruktur komplett neu aufbauen
- Zeit für die Konstruktion ist linear in ihrer Größe (mittels Hash-Tabellen)

Aktualisieren der Datenstruktur

- Eintrag (x, y) in L_i , ggf. Marker x' in L_j für $j < i$ und Verweise $x'.BMP$ in L_j mit $j > i$ einfügen
- potentiell sehr teuer, da viele Marker aktualisiert werden müssen
- keine gute Zeitschranke für Einfügen und Löschen
- besser: Änderungen über längere Zeit sammeln und dann die Datenstruktur komplett neu aufbauen
- Zeit für die Konstruktion ist linear in ihrer Größe (mittels Hash-Tabellen)

Aktualisieren der Datenstruktur

- Eintrag (x, y) in L_i , ggf. Marker x' in L_j für $j < i$ und Verweise $x'.BMP$ in L_j mit $j > i$ einfügen
- potentiell sehr teuer, da viele Marker aktualisiert werden müssen
- keine gute Zeitschranke für Einfügen und Löschen
- besser: Änderungen über längere Zeit sammeln und dann die Datenstruktur komplett neu aufbauen
- Zeit für die Konstruktion ist linear in ihrer Größe (mittels Hash-Tabellen)

Laufzeitverbesserung der BMP-Suche

- gesehen: Suche von BMP hat die Laufzeit $O(\log W)$
- falls nur l verschiedene Prefixlängen, $l < W$, dann ist die Laufzeit $O(\log l)$
- Reduzierung von Längenklassen mittels Prefix-Expansion
- Prefix x (Länge $k < W$) wird durch zwei Prefixe x_0 und x_1 (Länge $k+1$) ersetzt, d.h. in längere Klasse eingetragen
- Eintrag (x,y) wird durch (x_0,y) und (x_1,y) ersetzt, falls es (x_0,y') bzw. (x_1,y') noch nicht gibt
- durch j -fache Expansion von x (Länge k) erhält man höchstens 2^j neue Prefixe (Länge $k+j$)

Laufzeitverbesserung der BMP-Suche

- gesehen: Suche von BMP hat die Laufzeit $O(\log W)$
- falls nur l verschiedene Prefixlängen, $l < W$, dann ist die Laufzeit $O(\log l)$
- Reduzierung von Längenklassen mittels Prefix-Expansion
- Prefix x (Länge $k < W$) wird durch zwei Prefixe x_0 und x_1 (Länge $k+1$) ersetzt, d.h. in längere Klasse eingetragen
- Eintrag (x,y) wird durch (x_0,y) und (x_1,y) ersetzt, falls es (x_0,y') bzw. (x_1,y') noch nicht gibt
- durch j -fache Expansion von x (Länge k) erhält man höchstens 2^j neue Prefixe (Länge $k+j$)

Laufzeitverbesserung der BMP-Suche

- gesehen: Suche von BMP hat die Laufzeit $O(\log W)$
- falls nur l verschiedene Prefixlängen, $l < W$, dann ist die Laufzeit $O(\log l)$
- Reduzierung von Längenklassen mittels Prefix-Expansion
- Prefix x (Länge $k < W$) wird durch zwei Prefixe x_0 und x_1 (Länge $k+1$) ersetzt, d.h. in längere Klasse eingetragen
- Eintrag (x,y) wird durch (x_0,y) und (x_1,y) ersetzt, falls es (x_0,y') bzw. (x_1,y') noch nicht gibt
- durch j -fache Expansion von x (Länge k) erhält man höchstens 2^j neue Prefixe (Länge $k+j$)

Laufzeitverbesserung der BMP-Suche

- gesehen: Suche von BMP hat die Laufzeit $O(\log W)$
- falls nur l verschiedene Prefixlängen, $l < W$, dann ist die Laufzeit $O(\log l)$
- Reduzierung von Längenklassen mittels Prefix-Expansion
- Prefix x (Länge $k < W$) wird durch zwei Prefixe x_0 und x_1 (Länge $k+1$) ersetzt, d.h. in längere Klasse eingetragen
- Eintrag (x,y) wird durch (x_0,y) und (x_1,y) ersetzt, falls es (x_0,y') bzw. (x_1,y') noch nicht gibt
- durch j -fache Expansion von x (Länge k) erhält man höchstens 2^j neue Prefixe (Länge $k+j$)

Laufzeitverbesserung der BMP-Suche

- gesehen: Suche von BMP hat die Laufzeit $O(\log W)$
- falls nur l verschiedene Prefixlängen, $l < W$, dann ist die Laufzeit $O(\log l)$
- Reduzierung von Längenklassen mittels Prefix-Expansion
- Prefix x (Länge $k < W$) wird durch zwei Prefixe x_0 und x_1 (Länge $k+1$) ersetzt, d.h. in längere Klasse eingetragen
- Eintrag (x,y) wird durch (x_0,y) und (x_1,y) ersetzt, falls es (x_0,y') bzw. (x_1,y') noch nicht gibt
- durch j -fache Expansion von x (Länge k) erhält man höchstens 2^j neue Prefixe (Länge $k+j$)

Laufzeitverbesserung der BMP-Suche

- gesehen: Suche von BMP hat die Laufzeit $O(\log W)$
- falls nur l verschiedene Prefixlängen, $l < W$, dann ist die Laufzeit $O(\log l)$
- Reduzierung von Längenklassen mittels Prefix-Expansion
- Prefix x (Länge $k < W$) wird durch zwei Prefixe x_0 und x_1 (Länge $k+1$) ersetzt, d.h. in längere Klasse eingetragen
- Eintrag (x,y) wird durch (x_0,y) und (x_1,y) ersetzt, falls es (x_0,y') bzw. (x_1,y') noch nicht gibt
- durch j -fache Expansion von x (Länge k) erhält man höchstens 2^j neue Prefixe (Länge $k+j$)

Beispiel

Routing-Tabelle mit 7 nicht-leeren Längenklassen (8 Einträge)

$$L_1 = \{(0, y_1), (1, y_2)\}$$

$$L_2 = \{(10, y_3)\}$$

$$L_3 = \{(111, y_4)\}$$

$$L_4 = \{(1000, y_5)\}$$

$$L_5 = \{(11001, y_6)\}$$

$$L_6 = \{(100000, y_7)\}$$

$$L_7 = \{(1000000, y_8)\}.$$

Reduzierung auf 3 Längenklassen (13 Einträge)

$$l_1 = 2, l_2 = 5, l_3 = 7$$

$$L_2 = \{(00, y_1), (01, y_1), (10, y_3), (11, y_2)\}$$

$$L_5 = \{(11100, y_4), (11101, y_4), (11110, y_4), (11111, y_4), (10000, y_5), (10001, y_5), (11001, y_6)\}$$

$$L_7 = \{(1000000, y_8), (1000001, y_7)\}$$

Wahl der Längenklassen

- Speicherplatzbedarf der Datenstrukturen ist abhängig von der Anzahl der Prefixe
- optimale Wahl der r Längenklassen mit minimalen Gesamtanzahl von Prefixen
- realisiert mittels dynamischer Programmierung, siehe Anhang
- Laufzeit der Berechnung $O(rW^2)$

Wahl der Längenklassen

- Speicherplatzbedarf der Datenstrukturen ist abhängig von der Anzahl der Prefixe
- optimale Wahl der r Längenklassen mit minimalen Gesamtanzahl von Prefixen
- realisiert mittels dynamischer Programmierung, siehe Anhang
- Laufzeit der Berechnung $O(rW^2)$

Wahl der Längenklassen

- Speicherplatzbedarf der Datenstrukturen ist abhängig von der Anzahl der Prefixe
- optimale Wahl der r Längenklassen mit minimalen Gesamtanzahl von Prefixen
- realisiert mittels dynamischer Programmierung, siehe Anhang
- Laufzeit der Berechnung $O(rW^2)$

Wahl der Längenklassen

- Speicherplatzbedarf der Datenstrukturen ist abhängig von der Anzahl der Prefixe
- optimale Wahl der r Längenklassen mit minimalen Gesamtanzahl von Prefixen
- realisiert mittels dynamischer Programmierung, siehe Anhang
- Laufzeit der Berechnung $O(rW^2)$

Multibit-Tries

- Suche von BMP im Trie $O(W)$, vgl. Abschnitt 1
- Verbesserung: nicht nur 1 Bit, sondern mehrere Bits betrachten -> Multibit-Tries
- statt Beschriftungen an Kanten, verwendet man im Knoten v ein Feld mit 2^{k_v} Elementen für jede Kantenbeschriftung der Länge k_v
- jede Schicht entspricht einer Längenkategorie, r nicht-leere Längenkategorien nach der Prefix-Expansion
- Laufzeit für die Suche $O(r)$, da r Speicherzugriffe nötig

Multibit-Tries

- Suche von BMP im Trie $O(W)$, vgl. Abschnitt 1
- Verbesserung: nicht nur 1 Bit, sondern mehrere Bits betrachten -> Multibit-Tries
- statt Beschriftungen an Kanten, verwendet man im Knoten v ein Feld mit 2^{k_v} Elementen für jede Kantenbeschriftung der Länge k_v
- jede Schicht entspricht einer Längenkategorie, r nicht-leere Längenklassen nach der Prefix-Expansion
- Laufzeit für die Suche $O(r)$, da r Speicherzugriffe nötig

Multibit-Tries

- Suche von BMP im Trie $O(W)$, vgl. Abschnitt 1
- Verbesserung: nicht nur 1 Bit, sondern mehrere Bits betrachten -> Multibit-Tries
- statt Beschriftungen an Kanten, verwendet man im Knoten v ein Feld mit 2^{k_v} Elementen für jede Kantenbeschriftung der Länge k_v
- jede Schicht entspricht einer Längenkategorie, r nicht-leere Längenkategorien nach der Prefix-Expansion
- Laufzeit für die Suche $O(r)$, da r Speicherzugriffe nötig

Multibit-Tries

- Suche von BMP im Trie $O(W)$, vgl. Abschnitt 1
- Verbesserung: nicht nur 1 Bit, sondern mehrere Bits betrachten -> Multibit-Tries
- statt Beschriftungen an Kanten, verwendet man im Knoten v ein Feld mit 2^{k_v} Elementen für jede Kantenbeschriftung der Länge k_v
- jede Schicht entspricht einer Längenkategorie, r nicht-leere Längenklassen nach der Prefix-Expansion
- Laufzeit für die Suche $O(r)$, da r Speicherzugriffe nötig

Multibit-Tries

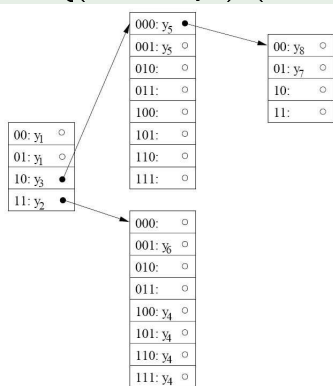
- Suche von BMP im Trie $O(W)$, vgl. Abschnitt 1
- Verbesserung: nicht nur 1 Bit, sondern mehrere Bits betrachten -> Multibit-Tries
- statt Beschriftungen an Kanten, verwendet man im Knoten v ein Feld mit 2^{k_v} Elementen für jede Kantenbeschriftung der Länge k_v
- jede Schicht entspricht einer Längenkategorie, r nicht-leere Längenkategorien nach der Prefix-Expansion
- Laufzeit für die Suche $O(r)$, da r Speicherzugriffe nötig

Beispiel

$L_2 = \{(00, y_1), (01, y_1), (10, y_3), (11, y_2)\}$

$L_5 = \{(11100, y_4), (11101, y_4), (11110, y_4), (11111, y_4), (10000, y_5), (10001, y_5), (11001, y_6)\}$

$L_7 = \{(1000000, y_8), (1000001, y_7)\}; z = 1000010$



Minimierung des Speicherplatzes für Multibit-Tries

- Speicherplatzbedarf für jeden Knoten ist die Anzahl seiner Feldelemente
- im oberen Bsp. $2^2 + 2^3 + 2^3 + 2^2 = 24$
- r vorgegeben, Längenklassen so wählen, dass der Speicherplatzbedarf minimal wird
- mittels dynamischer Programmierung (analog zur früheren Berechnung)
- Laufzeit $O(NW + rW^2)$
- mit $O(NW)$ für zusätzliche Berechnung des Speicherplatzes der Schicht für Längenkategorie L_l , wenn die nächstkleinere nicht-leere Längenkategorie $L_{l'}$ mit $l' < l$ ist

Minimierung des Speicherplatzes für Multibit-Tries

- Speicherplatzbedarf für jeden Knoten ist die Anzahl seiner Feldelemente
- im oberen Bsp. $2^2 + 2^3 + 2^3 + 2^2 = 24$
- r vorgegeben, Längenklassen so wählen, dass der Speicherplatzbedarf minimal wird
- mittels dynamischer Programmierung (analog zur früheren Berechnung)
- Laufzeit $O(NW + rW^2)$
- mit $O(NW)$ für zusätzliche Berechnung des Speicherplatzes der Schicht für Längenkategorie L_l , wenn die nächstkleinere nicht-leere Längenkategorie $L_{l'}$ mit $l' < l$ ist

Minimierung des Speicherplatzes für Multibit-Tries

- Speicherplatzbedarf für jeden Knoten ist die Anzahl seiner Feldelemente
- im oberen Bsp. $2^2 + 2^3 + 2^3 + 2^2 = 24$
- r vorgegeben, Längenklassen so wählen, dass der Speicherplatzbedarf minimal wird
- mittels dynamischer Programmierung (analog zur früheren Berechnung)
- Laufzeit $O(NW + rW^2)$
- mit $O(NW)$ für zusätzliche Berechnung des Speicherplatzes der Schicht für Längenkategorie L_l , wenn die nächstkleinere nicht-leere Längenkategorie $L_{l'}$ mit $l' < l$ ist

Minimierung des Speicherplatzes für Multibit-Tries

- Speicherplatzbedarf für jeden Knoten ist die Anzahl seiner Feldelemente
- im oberen Bsp. $2^2 + 2^3 + 2^3 + 2^2 = 24$
- r vorgegeben, Längenklassen so wählen, dass der Speicherplatzbedarf minimal wird
- mittels dynamischer Programmierung (analog zur früheren Berechnung)
- Laufzeit $O(NW + rW^2)$
- mit $O(NW)$ für zusätzliche Berechnung des Speicherplatzes der Schicht für Längenkategorie L_l , wenn die nächstkleinere nicht-leere Längenkategorie $L_{l'}$ mit $l' < l$ ist

Minimierung des Speicherplatzes für Multibit-Tries

- Speicherplatzbedarf für jeden Knoten ist die Anzahl seiner Feldelemente
- im oberen Bsp. $2^2 + 2^3 + 2^3 + 2^2 = 24$
- r vorgegeben, Längensklassen so wählen, dass der Speicherplatzbedarf minimal wird
- mittels dynamischer Programmierung (analog zur früheren Berechnung)
- Laufzeit $O(NW + rW^2)$
- mit $O(NW)$ für zusätzliche Berechnung des Speicherplatzes der Schicht für Längensklasse L_l , wenn die nächstkleinere nicht-leere Längensklasse $L_{l'}$ mit $l' < l$ ist

Minimierung des Speicherplatzes für Multibit-Tries

- Speicherplatzbedarf für jeden Knoten ist die Anzahl seiner Feldelemente
- im oberen Bsp. $2^2 + 2^3 + 2^3 + 2^2 = 24$
- r vorgegeben, Längenklassen so wählen, dass der Speicherplatzbedarf minimal wird
- mittels dynamischer Programmierung (analog zur früheren Berechnung)
- Laufzeit $O(NW + rW^2)$
- mit $O(NW)$ für zusätzliche Berechnung des Speicherplatzes der Schicht für Längenkategorie L_l , wenn die nächstkleinere nicht-leere Längenkategorie $L_{l'}$ mit $l' < l$ ist

Dynamische Programmierung zur Wahl der Längenklassen

- m - maximale Länge eines Prefix, r - Anzahl von Längenklassen
- für $l \in \{0, 1, \dots, m\}$ und $k \in \{1, 2, \dots, r\}$ definiert man den Wert $M[l][k]$ als minimale Anzahl von Prefixen, wenn man Längenklassen L_1, \dots, L_l in höchstens k nicht-leere Längenklassen umwandelt
- $M[m][r]$ gibt optimale Prefix-Anzahl an, die man am Ende erhält, wenn man durch Prefix-Expansion r nicht-leere Längenklassen konstruiert
- $E[k_1][k_2]$ für $1 \leq k_1 \leq k_2 \leq m$ gibt an, wie viele Prefixe in L_{k_2} entstehen, wenn man alle Prefixe aus L_{k_1} bis L_{k_2-1} auf Länge k_2 expandiert (in obigem Beispiel $E[3][5] = 7$)

Dynamische Programmierung zur Wahl der Längenklassen

- m - maximale Länge eines Prefix, r - Anzahl von Längenklassen
- für $l \in \{0, 1, \dots, m\}$ und $k \in \{1, 2, \dots, r\}$ definiert man den Wert $M[l][k]$ als minimale Anzahl von Prefixen, wenn man Längenklassen L_1, \dots, L_l in höchstens k nicht-leere Längenklassen umwandelt
- $M[m][r]$ gibt optimale Prefix-Anzahl an, die man am Ende erhält, wenn man durch Prefix-Expansion r nicht-leere Längenklassen konstruiert
- $E[k_1][k_2]$ für $1 \leq k_1 \leq k_2 \leq m$ gibt an, wie viele Prefixe in L_{k_2} entstehen, wenn man alle Prefixe aus L_{k_1} bis L_{k_2-1} auf Länge k_2 expandiert (in obigem Beispiel $E[3][5] = 7$)

Dynamische Programmierung zur Wahl der Längenklassen

- m - maximale Länge eines Prefix, r - Anzahl von Längenklassen
- für $l \in \{0, 1, \dots, m\}$ und $k \in \{1, 2, \dots, r\}$ definiert man den Wert $M[l][k]$ als minimale Anzahl von Prefixen, wenn man Längenklassen L_1, \dots, L_l in höchstens k nicht-leere Längenklassen umwandelt
- $M[m][r]$ gibt optimale Prefix-Anzahl an, die man am Ende erhält, wenn man durch Prefix-Expansion r nicht-leere Längenklassen konstruiert
- $E[k_1][k_2]$ für $1 \leq k_1 \leq k_2 \leq m$ gibt an, wie viele Prefixe in L_{k_2} entstehen, wenn man alle Prefixe aus L_{k_1} bis L_{k_2-1} auf Länge k_2 expandiert (in obigem Beispiel $E[3][5] = 7$)

Dynamische Programmierung zur Wahl der Längenklassen

- m - maximale Länge eines Prefix, r - Anzahl von Längenklassen
- für $l \in \{0, 1, \dots, m\}$ und $k \in \{1, 2, \dots, r\}$ definiert man den Wert $M[l][k]$ als minimale Anzahl von Prefixen, wenn man Längenklassen L_1, \dots, L_l in höchstens k nicht-leere Längenklassen umwandelt
- $M[m][r]$ gibt optimale Prefix-Anzahl an, die man am Ende erhält, wenn man durch Prefix-Expansion r nicht-leere Längenklassen konstruiert
- $E[k_1][k_2]$ für $1 \leq k_1 \leq k_2 \leq m$ gibt an, wie viele Prefixe in L_{k_2} entstehen, wenn man alle Prefixe aus L_{k_1} bis L_{k_2-1} auf Länge k_2 expandiert (in obigem Beispiel $E[3][5] = 7$)

- $M[l][1] = E[1][l]$ für alle $l \geq 1$
- $M[0][k] = 0$ für alle $k \geq 1$
- $M[l][k] = \min_{0 \leq l' < l} M[l'][k-1] + E[l'+1][l]$ für $l > 0, k > 1$
- für jedes l' setzt sich optimale Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in k nicht-leere Längenklassen zusammen aus
 - 1 Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in $k-1$ Längenklassen ($M[l'][k-1]$ Prefixe)
 - 2 Prefix-Expansion für Längenklassen $l'+1, \dots, l$ in eine Längenkategorie L_l ($E[l'+1][l]$ Prefixe)
- Laufzeit $O(rW^2)$ für alle Werte $M[l][k]$

- $M[l][1] = E[1][l]$ für alle $l \geq 1$
- $M[0][k] = 0$ für alle $k \geq 1$
- $M[l][k] = \min_{0 \leq l' < l} M[l'][k-1] + E[l'+1][l]$ für $l > 0, k > 1$
- für jedes l' setzt sich optimale Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in k nicht-leere Längenklassen zusammen aus
 - 1 Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in $k-1$ Längenklassen ($M[l'][k-1]$ Prefixe)
 - 2 Prefix-Expansion für Längenklassen $l'+1, \dots, l$ in eine Längenkategorie L_l ($E[l'+1][l]$ Prefixe)
- Laufzeit $O(rW^2)$ für alle Werte $M[l][k]$

- $M[l][1] = E[1][l]$ für alle $l \geq 1$
- $M[0][k] = 0$ für alle $k \geq 1$
- $M[l][k] = \min_{0 \leq l' < l} M[l'][k-1] + E[l'+1][l]$ für $l > 0, k > 1$
- für jedes l' setzt sich optimale Prefix-Expansion für Längenklassen $1, 2, \dots, l$ in k nicht-leere Längenklassen zusammen aus
 - 1 Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in $k-1$ Längenklassen ($M[l'][k-1]$ Prefixe)
 - 2 Prefix-Expansion für Längenklassen $l'+1, \dots, l$ in eine Längenkategorie L_l ($E[l'+1][l]$ Prefixe)
- Laufzeit $O(rW^2)$ für alle Werte $M[l][k]$

- $M[l][1] = E[1][l]$ für alle $l \geq 1$
- $M[0][k] = 0$ für alle $k \geq 1$
- $M[l][k] = \min_{0 \leq l' < l} M[l'][k-1] + E[l'+1][l]$ für $l > 0, k > 1$
- für jedes l' setzt sich optimale Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in k nicht-leere Längenklassen zusammen aus
 - 1 Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in $k-1$ Längenklassen ($M[l'][k-1]$ Prefixe)
 - 2 Prefix-Expansion für Längenklassen $l'+1, \dots, l$ in eine Längenkategorie L_l ($E[l'+1][l]$ Prefixe)
- Laufzeit $O(rW^2)$ für alle Werte $M[l][k]$

- $M[l][1] = E[1][l]$ für alle $l \geq 1$
- $M[0][k] = 0$ für alle $k \geq 1$
- $M[l][k] = \min_{0 \leq l' < l} M[l'][k-1] + E[l'+1][l]$ für $l > 0, k > 1$
- für jedes l' setzt sich optimale Prefix-Expansion für Längenklassen $1, 2, \dots, l$ in k nicht-leere Längenklassen zusammen aus
 - 1 Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in $k-1$ Längenklassen ($M[l'][k-1]$ Prefixe)
 - 2 Prefix-Expansion für Längenklassen $l'+1, \dots, l$ in eine Längenkategorie L_l ($E[l'+1][l]$ Prefixe)
- Laufzeit $O(rW^2)$ für alle Werte $M[l][k]$

- $M[l][1] = E[1][l]$ für alle $l \geq 1$
- $M[0][k] = 0$ für alle $k \geq 1$
- $M[l][k] = \min_{0 \leq l' < l} M[l'][k-1] + E[l'+1][l]$ für $l > 0, k > 1$
- für jedes l' setzt sich optimale Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in k nicht-leere Längenklassen zusammen aus
 - 1 Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in $k-1$ Längenklassen ($M[l'][k-1]$ Prefixe)
 - 2 Prefix-Expansion für Längenklassen $l'+1, \dots, l$ in eine Längenkategorie L_l ($E[l'+1][l]$ Prefixe)
- Laufzeit $O(rW^2)$ für alle Werte $M[l][k]$

- $M[l][1] = E[1][l]$ für alle $l \geq 1$
- $M[0][k] = 0$ für alle $k \geq 1$
- $M[l][k] = \min_{0 \leq l' < l} M[l'][k-1] + E[l'+1][l]$ für $l > 0, k > 1$
- für jedes l' setzt sich optimale Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in k nicht-leere Längenklassen zusammen aus
 - 1 Prefix-Expansion für Längenklassen $1, 2, \dots, l'$ in $k-1$ Längenklassen ($M[l'][k-1]$ Prefixe)
 - 2 Prefix-Expansion für Längenklassen $l'+1, \dots, l$ in eine Längenkategorie L_l ($E[l'+1][l]$ Prefixe)
- Laufzeit $O(rW^2)$ für alle Werte $M[l][k]$

Algorithmus

Berechnung der $M[l][k]$ mittels dynamischer Programmierung

Eingabe: Werte $E[k_1][k_2]$ für $1 \leq k_1 \leq k_2 \leq m$, Anzahl r von Längenklassen

Ausgabe: Werte $M[l][k]$ und $P[l][k]$ für $0 \leq l \leq m$, $1 \leq k \leq r$

```
for  $l=0$  to  $m$  do
```

```
  for  $k = 1$  to  $r$  do
```

```
    if  $l = 0$  then  $M[l][k] := 0$ ;  $P[l][k] := 0$ ;
```

```
    else if  $k = 1$  then  $M[l][k] := E[1][l]$ ;  $P[l][k] := 0$ ;
```

```
    else
```

```
       $M[l][k] := \min_{0 \leq l' < l} M[l'][k-1] + E[l'+1][l]$ ;
```

```
       $P[l][k] :=$  das  $l'$ , für das sich in der vorigen Zeile
```

```
das Minimum ergab;
```

```
    fi;
```

```
  od;
```

```
od;
```

Algorithmus

Berechnung der $M[l][k]$ mittels dynamischer Programmierung

- am Ende der Berechnung gibt der Wert $M[m][r]$ die optimale Prefix-Anzahl an
- die r zugehörigen Prefix-Längen (in absteigender Reihenfolge) sind:
- $m, P[m][r], P[P[m][r]][r-1], P[P[P[m][r]][r-1]][r-2], \dots$

Algorithmus

Berechnung der $M[l][k]$ mittels dynamischer Programmierung

- am Ende der Berechnung gibt der Wert $M[m][r]$ die optimale Prefix-Anzahl an
- die r zugehörigen Prefix-Längen (in absteigender Reihenfolge) sind:
 - $m, P[m][r], P[P[m][r]][r-1], P[P[P[m][r]][r-1]][r-2], \dots$

Algorithmus

Berechnung der $M[l][k]$ mittels dynamischer Programmierung

- am Ende der Berechnung gibt der Wert $M[m][r]$ die optimale Prefix-Anzahl an
- die r zugehörigen Prefix-Längen (in absteigender Reihenfolge) sind:
- $m, P[m][r], P[P[m][r]][r-1], P[P[P[m][r]][r-1]][r-2], \dots$

- Datenstrukturen und Algorithmen für schnellen IP Prefix Lookup
- Tries, Längenklassen, Prefix Expansion

- Datenstrukturen und Algorithmen für schnellen IP Prefix Lookup
- Tries, Längenklassen, Prefix Expansion



T. Erlebach

Algorithmen für Kommunikationsnetze

Skript WS 2002/03, EHT Zürich