UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Dokumentation zu Routing-Tabellen und Forwarding

Studienarbeit

im Studiengang Informatik

vorgelegt von

Dirk Steffes-Enn

Universität Koblenz-Landau

Campus Koblenz

Institut für Informatik

AG Rechnernetze und Rechnerarchitektur

Betreuer:

Prof. Dr. Hannes Frey

Dipl. Inf. Frank Bohdanowicz

(Institut für Informatik, AG Rechnernetze und Rechnerarchitektur)

Koblenz, 02. Juli 2013

Erklärung

	Ja	Nein
Mit der Einstellung dieser Arbeit in der Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

_____

Ort, Datum Unterschrift

Eidesstattliche Erklärung

Hiermit versichere ich gemäß der Diplomprüfungsordnung Informatik der Universität Koblenz-Landau, Campus Koblenz, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmitteln und Quellen benutzt habe. Außerdem bestätige ich, dass die Arbeit in gleicher oder ähnlicher Form noch keinem anderen Prüfungsausschuss vorgelegen hat.

Koblenz, 02.07.2013

_____

Dirk Steffes-Enn

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

In den letzten Jahrzenten sind die Anforderungen an Internet-Router deutlich gestiegen. Die Anzahl der Teilnehmer im Internet ist immens angestiegen. So waren von den insgesamt ca. 4,3 Milliarden verfügbaren IPv4-Adressen Ende des Jahres 2009 knapp 95 % registriert [01].

Dadurch sind auch die zu verwaltenden Adress-Bereiche in einem Router in den letzten Jahren umfangreicher geworden und erfordern Strukturierungen, um die Adress-Bereiche verwalten zu können.

Waren in den Anfangszeiten die Einträge in einer Routingtabelle noch vergleichsweise gering, sind diese durch die größere Vernetzung und auch Umstrukturierungen in Zuweisungen der Adressbereichen, z.B. Classless Inter-Domain Routing (**CIDR**), deutlich angewachsen.

Dies erfordert eine effiziente Organisation der Prozesse in einem Router, damit Anfragen nach einer korrekten Paketweiterleitung schnell erfolgen können.

Bei der Grundüberlegung zur Strukturierung der Adressen im Internet wurden anfangs strikt drei verschiedene Klassen definiert. Für Organisationen mit vielen Teilnehmern eine Klasse mit verhältnismäßig wenig verschiedenen Adressbereichen (Klasse A). Für Organisation mit wenigen Teilnehmern umgekehrt gab es eine Klasseneinteilung mit einer hohen Anzahl von verschiedenen Adressbereichen (Klasse C). Und eine mittlere Klasse mit einem Kompromiss zwischen Teilnehmeranzahl und Adressbereichen bezeichnet mit Klasse B. [03] (S. 446)

In den Anfängen des Internets ging man noch davon aus, dass diese Einteilung ausreichend sein würde. So wurde zum Beispiel Firmen oftmals ein größeres Klasse-B-Netz zugewiesen, auch wenn mehrere Klasse-C-Netzwerke ausreichend gewesen wären.

Die Technik wurde Anfang der 90er Jahre zu einem Problem, da die Anzahl der noch freien Klasse-B-Netze geringer wurde. Behoben wurde dies dann mit der Zuteilung von mehreren Klasse-C-Netzen für eine Organisation in Verbindung mit der Option von unterschiedlichen Längen der Subnet-Mask (VLSM = variable length subnet mask). [03] (S.448)

Bei den nun komplexer werdenden Routing-Prozessen mussten neue Strategien zur Adressfindung entworfen werden. Da unterschiedliche Adress-Bereiche nun einem Unternehmen zugeordnet werden konnten, entstanden in den Routing-Tabellen auch wesentlich mehr Einträge, die durchsucht werden müssen.

Dies wirkt sich direkt auf die Leistungsfähigkeit eines Routers aus.

Drei Faktoren können als Maßstab für die Leistungsfähigkeit eines Netzwerks als verantwortlich gekennzeichnet werden [02]:

- Bandbreite der Übertragungsmedien
- Bandbreite der Router
- Geschwindigkeit von Paket-Weiterleitungen bzw. Forwarding

Dabei steht bei den ersten beiden Faktoren der Begriff **Bandbreite** bei Übertragungsmedien und Routern zusammengefasst für die Menge an Bytes, die innerhalb eines festgelegten Zeitrahmens übertragen werden.

Der dritte Faktor ist abhängig vom implementierten Verfahren in einem Router, um die **Paket-Weiterleitung** durchzuführen. [07]

Forwarding bezeichnet den Vorgang in einem Router, bei vorliegenden Datenpaketen die Zieladresse aus dem Header zu extrahieren, anhand der Adresse in der Weiterleitungstabelle den passenden Ausgangsport zu bestimmen und die korrekte Paketverarbeitung zu veranlassen.

Die angewandten Algorithmen zur Erstellung und Aktualisierungen von Routing-Tabellen haben jeweils deutlichen Einfluss auf die Geschwindigkeit.

2. Ziel der Studienarbeit

In dieser Studienarbeit sollen verschiedene gängige Verfahren aufgelistet und verglichen werden, mit denen eine Routing-Tabelle erstellt und angepasst werden kann. Dazu werden hier nur dynamische Verfahren in Betracht gezogen.

Allgemein wird die Funktionsweise einer Routingtabelle erklärt und drei Verfahren bzw. Algorithmen analysiert und bewertet.

Die Algorithmen werden anhand von Beispielen erläutert und in einem abschließenden Kapitel gegenüber gestellt. Dabei werden die Vor- und Nachteile der einzelnen Verfahren aufgelistet.

3. Routing

Als Routing wird im Allgemeinen die Berechnung von Wegen bezeichnet, um Daten bzw. Datenpaketen in Rechnernetzen zwischen Hosts auszutauschen. Diese Aufgabe wird von Routern übernommen, die verschiedene Netze miteinander verbinden und für die Weiterleitung der ihnen zugesendeten Datenpakete zuständig sind.

Soll beispielsweise von Rechner A ein Datenpaket an Rechner B gesendet werden, gibt es oftmals verschiedene Möglichkeiten, einen Pfad zu finden. Idealerweise wird dabei der kürzeste Pfad genutzt, wobei der Begriff "kürzeste" z.B. für die geringste Anzahl an zurückgelegten Wegstrecken bzw. Routern benutzt werden kann. Dies bedeutet wiederum oftmals eine minimale Latenz und dadurch eine schnellstmögliche Übertragung der Pakete. [03] (S.377 - 381)

In dieser Studienarbeit werden IP-Adressen-basierte Netze betrachtet.

Die Organisation einer effizienten Routenführung erfordert oftmals einen hohen Aufwand zur Aktualisierung der notwendigen Netzwerk-Informationen, da die Netze sich dynamisch z.B. in der Teilnehmerzahl, Größe und Bandbreite ändern können. So

kann eine vorher optimale Route (z.B. kürzeste Verbindung) durch einen Hardware-Defekt nicht mehr verfügbar sein. Dies sollte effizient durch Kontrollmechanismen erkannt werden und mittels einer neuen Route umgangen werden.

Dazu gibt es verschiedene Lösungsstrategien, die sich vor allem in einer zentralen oder dezentralen Organisation unterscheiden:

Bei der zentralen Organisation der Routenführung übernimmt ein spezialisierter Router bzw. Host die komplette Berechnung und Erzeugung der Routen. Diesem müssen alle Teilnehmer und der Aufbau des Netzbereiches bekannt sein und anhand von verschiedenen Metriken werden dann die optimalen Routen berechnet.

Dezentral hingegen bedeutet, dass jeder Router selbstständig aus vorhandenen Informationen zum Netzwerk die Routen individuell berechnet.

Ausführungen zur dezentralen Router-Organisation sind das Distanz-Vektor-Verfahren und das Link-State-Verfahren, die im Kapitel 3.3 genauer beschrieben sind.

Der Prozess zum erstellen von Routen wird durch verschiedene Parameter beeinflusst. Dies können z.B. die kürzeste Route oder die größtmögliche Bandbreite bei der Wegfindung sein.

3.1 Adaptive und nicht-adaptive Routingverfahren

Die grundlegende Funktionsweise des Routing wird in adaptives und nicht adaptives Routing unterteilt.

Nichtadaptives Routing bedeutet ein statisches Verfahren, bei dem die Wegberechnung innerhalb eines Netzbereiches im Voraus berechnet wird und diese Informationen an alle im Netzbereich beteiligten Router weitergegeben wird. [03] (S.377)

Bei kleinen und konstanten Netzwerken ist ein solches Routing effizient, da nicht wie beim adaptiven Routing zusätzlicher Organisationsaufwand innerhalb des Betriebes auftritt, um die bekannte Struktur zu aktualisieren. Ein einmal so organisiertes Netz

arbeitet solange unverändert, bis eine erneute Durchführung (z.B. durch einen Neustart des kompletten Netzbereiches) erfolgt ist.

Vorteile sind ein optimales Anpassen des Routing an die Gegebenheiten des Netzes. Die festgelegten Routen können z.B. genau an die benötigte Bandbreite angepasst werden und bieten so ein vorausberechenbares Modell.

Nachteile sind das unflexible Verhalten auf Änderungen in der Struktur, z.B. durch einen neuen Router und optionalen Wegstrecken als auch durch eine technische Störung innerhalb eines abgegrenzten Bereiches (z.B. Routerdefekt), und der Organisationsaufwand zur Inbetriebnahme des Netzes bei Anpassungen an bestimmte Bedürfnisse.

Adaptives Routing ist im Gegensatz dazu ein dynamisches Verfahren. Aufgrund von Messverfahren (siehe Routingprotokolle) ergeben sich unter Umständen fortlaufend neue Ergebnisse zur Wegstrecken-Berechnung.

Dies bietet sich für größere und dynamische Netzbereiche an, in denen die Teilnehmer-Anzahl variabel ist.

Vorteile sind ein eigenständiges Anpassen des Netzes an geänderte Strukturen. Fällt z.B. ein Bereich durch einen Hardware-Defekt aus, kann dieser neue Zustand erkannt und eine geänderte Wegstrecke berechnet werden. Dieses wird dann den anderen angeschlossenen Routern mitgeteilt bzw. selbst erkannt. Durch optimierte Verfahren lassen sich auch Anpassungen an die Bedürfnisse ändern, wenn z.B. das Hauptinteresse einer kostengünstigen Übermittlung von Paketen gilt und nicht einer schnellstmöglichen.

Nachteile sind ein erhöhter Traffic innerhalb der Router-Kommunikation und mögliche Fehler, bei der nicht korrekte Wegstrecken trotzdem aktuell gehalten werden und an andere Router übermittelt werden.

In diesem Zusammenhang wird oft auch der Zielkonflikt erwähnt: dies bedeutet einen Kompromiss zwischen einer aktuellen Strukturerkennung (bedeutet schnelles Erkennen von defekten Routen, aber auch mehr Traffic) und Bandbreiten-Optimierung für Datentraffic (wenig Bandbreiten-Beanspruchung durch Router-Kommunikation, aber dadurch verspätetes Erkennen von Strukturänderungen im Rechnernetz). [03] (S. 376)

3.2 Routingalgorithmen

Mit den Algorithmen löst man vorrangig eine effiziente Umstrukturierung der Routingtabelle. Angewandt auf diese Tabellen werden z.B. Baumstrukturen erzeugt, in denen die Tiefe und Anzahl der Verzweigungen so gering als möglich gehalten werden, um schnellere Treffer zu liefern.

Die Begriffe Routingalgorithmen und Routingprotokolle werden oft im gleichen Zusammenhang genannt. In dieser Studienarbeit dient die Terminologie wie folgt:

Routingalgorithmen: Lösungsverfahren lokal auf einem Router, um eine vorhandene Routingtabelle in eine optimierte Struktur zu überführen und Aktualisierungen in der Struktur zu speichern. Diese Struktur kann linear als Hash-Tabelle oder auch als Baumstruktur organisiert sein.

Routingprotokoll: Lösungsverfahren verteilt in einem Rechnernetz, um die Struktur dieses Netzes zu erfassen und durch einen bzw. mehreren Router organisieren zu lassen. [10] (S. 271). Die beteiligten Knoten erfassen Daten wie z.B. die Entfernung zu benachbarten Knoten und tauschen diese Informationen mit anderen Knoten aus. Als Parameter werden beispielsweise die Bandbreite einer Verbindung oder die Verzögerung einer bestätigten Anfrage zwischen zwei Knoten gemessen.

RIP (Routing Information Protocol) ist ein Distanzvektor-Algorithmus und der Internet-Standard zu frühen Zeiten des Internets [10] (S. 271). Dabei erstellt jeder Knoten in einem Netzwerkbereich ein Array, in dem die Entfernung zu allen anderen Netzen gelistet wird. Diese Daten werden nur den unmittelbaren Nachbar-Knoten mitgeteilt. Jeder Router in einem Netzwerk hat dadurch Kenntnis über die anderen im Netzwerk vorhandenen Router.

Nachteilig ist damit die Dauer der Aktualisierung, die sich bei Änderungen im Netzwerk, z.B. bei Ausfall eines Routers, ergeben, da jeder Router die Informationen nur an seinen Nachbar-Router weiterleitet[13].

OSPF (Open Shortest Path First) hingegen als Beispiel für Link-State-Routing führt nur detaillierte Informationen über direkte Nachbar-Knoten, teilt diese Information aber allen Knoten im Netzwerk durch fluten von Aktualisierungs-Paketen mit. Dadurch lässt sich eine optimale Route (z.B. geringste Anzahl an beteiligten Knoten) für jeden Knoten separat berechnen.

Vorteilshaft ist die Speicherung der aktuellen Daten von anderen im Netzwerk beteiligten Routern. So kann bei einem Ausfall einer Route mit den vorliegenden Link-State-Paketen eine alternative Route berechnet werden, ohne dass eine Neuerfassung des Netzwerkaufbaus notwendig wäre. [14]

Interior / Exterior Gateway Protocol

Das Internet besteht aus einem Zusammenschluss von unzähligen Netzwerken, die unabhängig voneinander sind. Als Begriff für diese Netzwerkabschnitte wird autonomes System (AS) verwendet, wobei ein AS jeweils unter einer eigenen administrativen Verwaltung steht.[10] (S.312)

Die zuvor aufgeführten Beispiel-Protokolle RIP und OSPF werden intern in einem Netzwerk eingesetzt [10] (S.269). Externe Protokolle werden allgemein als **EGP** (Exterior Gateway Protocol) bezeichnet und unterstützen auch zusätzliche Regeln, um z.B. bestimmte Routen aus ökonomischen Gründen zu bevorzugen. [10] (S.311)

Grundsätzlich wird dabei unterschieden in der Kommunikation außerhalb autonomer Systeme (exterior) und innerhalb autonomer Systeme (interior).

Dazu wird als **BGP** (Border Gateway Protocol) als Standard-Protokoll benutzt. BGP ist ein Pfadvektor-Protokoll und berücksichtigt im Gegensatz zu den anderen Protokollen wie RIP oder OSPF zusätzliche Attribute, um eine Route zu beeinflussen.

In der Kommunikation zwischen verschiedenen AS wird das **eBGP** (exterior Border Gateway Protocol) verwendet. So kann z.B. bei der Paketweiterleitung ein bestimmtes AS gemieden werden und anhand der Attribute die Route über gewünschte AS bevorzugt werden. Weiterhin werden auch alternative Routen zu einem Zielsystem vorgehalten, um Attribute wie das vorherige genannte zu unterstützen. [15]

Innerhalb eines autonomen Systems wird **iBGP** (interior Border Gateway Protocol) eingesetzt [10] (S.317). Dies ist bedingt durch unterschiedliche Anforderungen an das Routing und berücksichtigt dadurch auch eine bessere Skalierbarkeit bei der Verbindung von autonomen Systemen. Dies soll hier aber nicht weiter erläutert werden.

3.3 Routingtabellen

In Routingtabellen werden von Routern die Informationen vorgehalten, wie mit ankommenden Datenpaketen zu verfahren ist. Unterschieden wird dabei zwischen einer Routingtabelle, in der Zieladresse und dem dazugehörigen next-hop, also den auf dieser Wegstrecke als nächsten zu adressierenden Router, gespeichert werden und der Weiterleitungstabelle, in der zusätzlich die physikalische Adresse des next-hop gespeichert wird.

Je komplexer ein Netzwerk ist, desto wichtiger ist eine zielgerichtete Weiterleitung von Datenpaketen. Ist es in kleinen Netzwerken noch möglich, ein Datenpaket per Broadcast zu versenden, ohne das Netz zu überlasten, würde dieses Vorgehen bei größeren Netzwerken eine viel zu große Bandbreite aufbrauchen. In den Weiterleitungstabellen wird von einem ankommenden Datenpaket die Zieladresse (bzw. ein Teil der Zieladresse) gelesen und mit einem Eintrag in der Tabelle abgeglichen, wohin das Paket weitergeleitet werden soll.

Für die Überprüfung gibt es verschiedene Lösungsstrategien [3] (S. 377), da bei größeren Netzwerken zahlreiche Einträge in der Routingtabelle entstehen.

Zum Aufbau der Tabellen gibt es Algorithmen, die eine Speicherstruktur als Baumstruktur oder Hashtabelle realisieren.

Als grundlegende Verfahren gelten die Baum- und die Hash-Struktur, die abhängig von der Anzahl der zu abspeichernden Adressen zur Durchführung des IP-Lookup Unterschiede in der Effizienz des verwendeten Verfahrens aufweisen. Eine Auflistung mit Unterschieden bei der Aufwandsberechnung findet sich im Fazit dieser Studienarbeit.

Zur Durchführung des IP-Lookup werden folgende Verfahren üblicherweise eingesetzt, von denen in dieser Studienarbeit das Trie-basierte Suchverfahren und die Hash-basierende Speicherung von Präfixen genauer betrachtet werden:

Trie-basierte Suchverfahren

Das am häufigsten verwendete Verfahren zum IP lookup basiert auf dem Radix-Baum. Problematisch sind der Speicherbedarf und die vergleichsweise hohe Anzahl von Speicherzugriffen (memory access). Diese liegen für IPv4 bei bis zu 32 und für IPv6 bei bis zu 128 (also entsprechend der Bit-Anzahl). Der Aufwand wird mit $\mathbf{O(W^2)}$ angegeben [02] für Umsetzungen der Suche ohne Optimierung. W entspricht der Adresslänge. Durch Verbesserungen in der Suche kann der Aufwand auf **O(W)** reduziert werden [02].

Modifikationen der exakten Suche

Da sich eine exakte Suche nicht für die Suche nach dem längsten Prefix eignet, aber eine schnelle Suche versprechen, gibt es einige Veränderungen, um das grundlegende Verfahren (binäre Suche) auf IP lookup anzuwenden.

Der Aufwand dieser Änderung ist $\mathbf{O(\log_2 (2N))}$ [02]. N entspricht der Anzahl der Einträge in der Routingtabelle.

Hardware-basierend

Eine Lösung mit einer Hardware-basierenden Suche vergleicht eine eingehende Adresse mit einer in einem Speicher hinterlegten Adresse. Dies kann parallel erfolgen und ermöglicht so eine sehr schnelle Suche. Diese Technik ist aber nur für kleine Datenstrukturen effizient, bei IPv6 werden zum Vergleich der Adresse inklusive Maske und den verschiedenen Prefix-Möglichkeiten unrentabel viele Hardware-Module benötigt[02].

Protokoll-basierend

Bei diesem Ansatz werden jedem IP-Packet zusätzliche Informationen, oft als Tags bezeichnet, mitgegeben, die der Router zur schnelleren Suche auswertet. Dies kann zum Beispiel mit einem Zahlenwert erfolgen, der vom Router eine direkte Treffermöglichkeit bei der Suche in der Routing-Tabelle liefert. Vorteile sind eine etwas schnellere und gezieltere Suche. Nachteile sind hingegen die umfangreichere Aktualisierung der notwendigen Daten im Netzwerk. Damit dies ausreichend funktioniert, müssen große Teile des Netzwerks einen gleichen Datenbestand besitzen[02].

Hash-basierend

Als Datenstruktur zum Abspeichern größerer Datenmengen eignen sich auch Hashtabellen. Daher gab es schon seit Beginn der Routingalgorithmen-Entwicklung die Idee, die Routinginformationen mittels Hash-Werten abzulegen.

Der grundsätzliche Ansatz zum Durchsuchen von Hash-Werten ist ohne Anpassung ineffizient. Denn bei der Hash-Suche wird generell nach einem exakten Wert gesucht, was aber den Anforderungen der längsten Prefix-Suche widerspricht, bei dem Teile eines Wertes als Suchtreffer gelten[02].

Bei den hier aufgeführten Verfahren ist der Ansatz mit Hardware-basierenden Suchvorgängen zu vernachlässigen, da Änderungen bei abgespeicherten Adressen softwareseitig schneller umsetzbar sind und die Anzahl der benötigten Hardware-Komponenten bei vielen Adressen nicht mehr effizient genug sind.

Als **Baumstrukturen** werden zur Speicherung der Adressen vor allem folgende drei Typen benutzt:

1. **Trie**: (aus dem engl. Wort retrieval)

   Baumstruktur, bei der durch jede Kante die Information im jeweils anschließenden Knoten erweitert wird. An jeder Kante befindet sich eine

einzelne Information (z.B. eine Ziffer), die dann in einem Blatt eine Zahl ergibt, die durch die Verkettung von Kanten definiert ist. [11] (S. 390)

2. **PATRICIA**: (Practical Algorithm To Retrieve Information Coded In Alphanumeric)

   binäre Baumstruktur, die wie bei Trie Informationen an den Kanten zu Ergebnissen in den Knoten verkettet. Dabei wird aber bei Patricia die Möglichkeit gegeben, mehr als eine einzelne Information an einer Kante zu hinterlegen, also mehrere Kanten zu einer Kante zu kombinieren. [12]

3. **Prefix Tree**:

   Pro Knoten wird ein Prefix einer IP-Adresse hinterlegt. Dadurch ist die Gesamtanzahl der Knoten gegenüber anderen Baumstrukturen geringer, da Ergebnisse nicht nur in Blättern abgespeichert sind. Ein Suchvorgang kann somit weniger Schritte in einem Baum benötigen. [04]

Als Vergleich dient die folgende Abbildung. Dabei sind beispielsweise die sechs zu speichernden Adressen bzw. deren Prefixe 011, 001, 0101, 01101, 01100 und 00110 vorhanden.

Dies ergibt folgende Baumstrukturen:

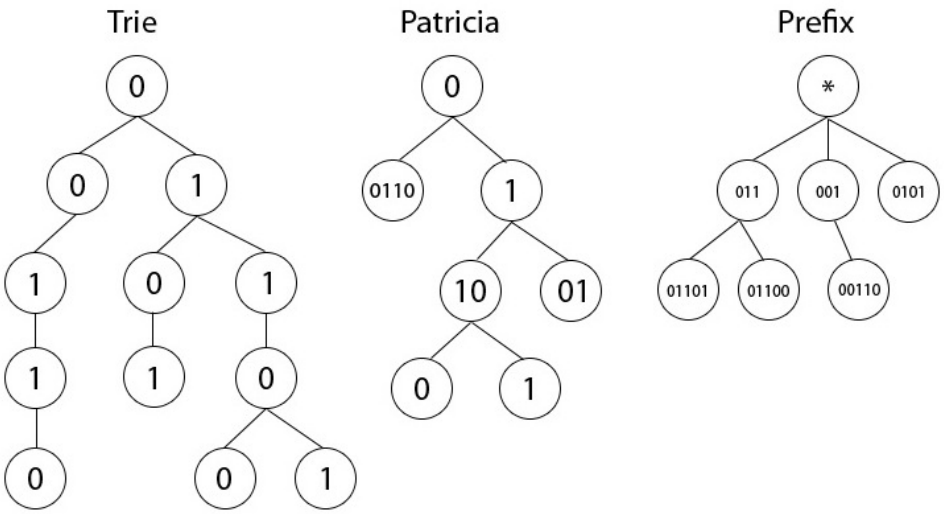


Abbildung 1, Vergleichsbeispiel Baumstrukturen

Die Tiefe der Baumstruktur nimmt in diesem Beispiel beim Übergang von Trie zur Prefix-Speicherung immer weiter ab, auch die Anzahl der Knoten ist beim Trie mit 12 gegenüber 7 beim Patricia und Prefix deutlich höher.

Zusammenfassend gesehen gibt es grundlegend verschiedene Verfahren, um Speicherstrukturen für den IP-Lookup zur Verfügung zu stellen. Die einfachste Struktur, die Liste, bietet bei einer großen Anzahl von Adress-Einträgen keine effektive Alternative, da die Suche linear erfolgt und der Aufwand ungünstig im Vergleich zu anderen Möglichkeiten wird. Dies wird im letzten Kapitel noch genauer gegenüber gestellt.

Die ideale Speicherstruktur für den IP-Lookup sollte also eine schnelle Suche ermöglichen, mit geringem Aufwand änderbar sein (für Neueinträge, Updates von Einträgen, Löschen von Einträgen) und einen geringen Speicherbedarf aufweisen. [17]

In den folgenden Kapiteln werden verschiedene Beispiele genauer betrachtet, die jeweils Vor- und Nachteile auf die zuvor genannten Wunschattribute aufweisen.

4. Baumstrukturen

Radix und Radish sind Beispiele für binäre Baumstrukturen, die eine kompakte Speicherung der Präfixe zur Verfügung stellen.

Mit **Baumstruktur** wird die Struktur der Speicherung kategorisiert: Daten werden von einem Startpunkt aus (Wurzel) in Abzweigungen (Knoten als Entscheidungspunkte) sortiert. Ein Ergebnis wird erzielt beim Erreichen eines Endpunktes (Blatt) im Baum

Der Radix-Algorithmus ist eine Trie-Variante. In den 90er Jahren wurde er eingesetzt, um die steigende Anzahl von Adress-Einträgen in Routern schneller zu kategorisieren, indem variable Adresslängen möglich sind und auch unterschiedliche Adressbereiche unterstützt wurden, die nicht zusammenhängend sind. Ein Nachteil entstand aber in der Komplexität der Realisierung dieses Algorithmus, aufgrund dessen die Vereinfachung Radish im Jahr 1995 vorgestellt wurde [05], der hier im Kapitel 5.6 vorgestellt wird.

Anhand von verschiedenen Beispielen soll hier zuerst die Funktionsweise des Radix vereinfacht dargestellt werden. Darauf erfolgen die Beschreibung des Radish und die Auflistung der Vorteile mittels der Beispiele.

4.1 Radix

Der Radix-Algorithmus ist eine Variante der Trie-basierten Strukturen. Dieser erlaubt eine Suche nach einem bestimmten Prefix variabler Länge und auch die Verarbeitung von unterschiedlichen, aber zusammengehörigen Adressbereichen. [05] Die Baumstruktur beim Radix ist binär. Ein Knoten steht dabei für eine einzelne Bit-Position, und es gibt in jedem Knoten eine Option zur Speicherung von einer Netz-Maske.

Als Suchverfahren wird "longest prefix match" eingesetzt. Dies bedeutet, dass bei einer Suche als Treffer der Eintrag mit der längsten übereinstimmenden Maske geliefert wird. Gibt es also zum Beispiel in einem Baum die Einträge 10.5.0.0 und 10.5.16.0, und es wird nach einer Route für 10.5.16.4 gesucht, so ist der Eintrag mit 10.5.16.0 der längere von beiden übereinstimmenden Masken. [05]

Radix Lookup

Die Grundlage für das Suchverfahren ist das "downward search and backtrack". Dabei wird die initiale Suche an der Wurzel gestartet und die Knoten bitweise untersucht, bis die Suche an einem Blatt ankommt.

Sollte dort ein Treffer vorliegen, ist die Suche beendet. Andernfalls wird sich innerhalb des Baumes rückwärts zu einem vorherigen Knoten bewegt. Beim Test mit einem Knoten wird per gesetztem Bit entschieden, ob die Suche rechtsseitig erfolgt oder bei ungesetztem Bit linksseitig erfolgen soll.

Anhand der Beispiele werden drei unterschiedliche Suchergebnisse erläutert:

Suchtreffer "host match"

Ein Suchvorgang führt zu einem Blatt. Ist der Vergleich der Zieladresse mit der im Blatt hinterlegten Adresse identisch, wird dieses als "host match" bezeichnet.

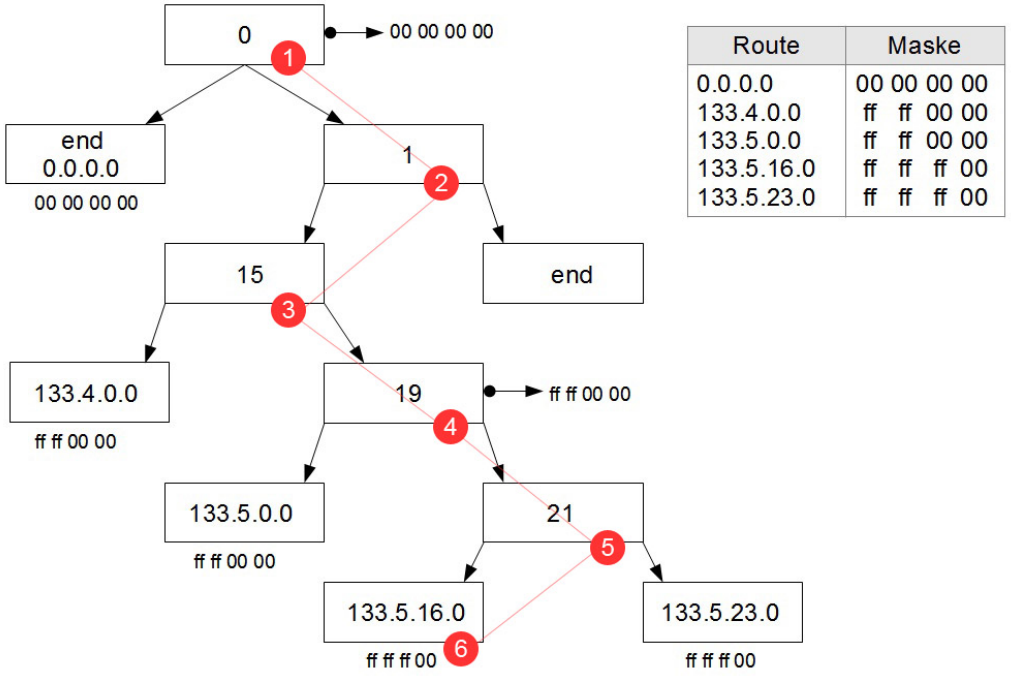


Abbildung 2, Radix Suchtreffer “host match“ [05]

Beispiel: die Zieladresse soll 133.5.16.0 sein. Beim Start wird diese mit dem Wurzel-Knoten verglichen. Bit 0 ist gesetzt, daher wird rechtsseitig im Baum weiter gesucht. Der nächste Knoten testet auf Bit 1, welches aber bei der Zieladresse ungesetzt ist. Die Suche geht linksseitig weiter. Bit 15 und Bit 19 sind gesetzt, Bit 21 aber nicht. Daher endet die Suche vorläufig bei Blatt "133.5.16.0". Beim Vergleich der Zieladresse mit der des Blattes ergibt dies einen "host match", da die Adressen identisch sind und die Suche ist erfolgreich[05].

Suchtreffer "network match"

Sollte ein Vergleich bei einem Blatt mit der Zieladresse nicht erfolgreich sein, wird die Zieladresse mit der Netzwerkmaske der IP-Adresse aus dem Blatt (bzw. einer der hinterlegten Masken) AND-verknüpft und wiederum verglichen. Ist der Vergleich ein Treffer, so wird dies "network match" genannt.

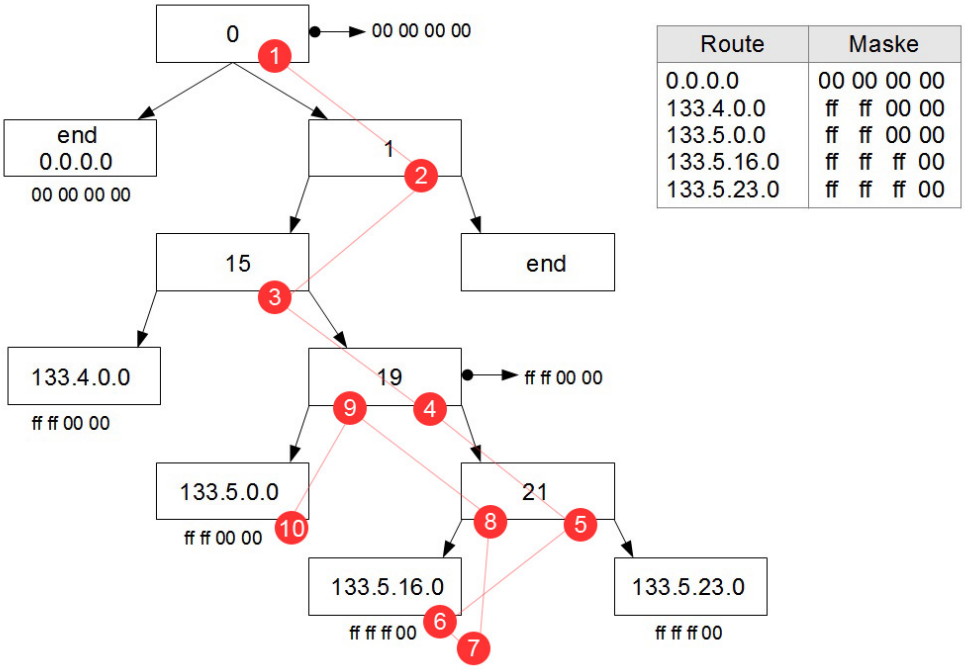


Abbildung 3, Radix Suchtreffer “network match“ [05]

Beispiel: 133.5.80.9 soll die nächste IP-Zieladresse sein. Die bitweise Überprüfung ergibt für Bit 0 ein gesetztes Bit, Bit 1 ist ungesetzt, Bit 15 gesetzt, Bit 19 gesetzt und Bit 21 ungesetzt. Somit landet die Suche beim gleichen Blatt wie im vorherigen Beispiel, aber ein Vergleich der Zieladresse mit 133.5.16.0 ergibt keinen Treffer.

Daher wird die Zieladresse mit der Netzwerkmaske ff ff ff 00 AND-verknüpft, aber auch das Ergebnis 133.5.80.0 führt zu keinem Treffer beim Vergleich mit der Zieladresse.

Als nächsten Schritt wird die Rückwärtssuche eingeleitet, da die bisherige Suche bis in ein Blatt erfolgte und zu keinem Treffer bei den Vergleichen der Adressen geführt hatte.

Die Rückwärtssuche springt schrittweise zu den Knoten Richtung Wurzel zurück, bis ein Knoten mit zusätzlich abgespeicherter Netzwerkmaske gefunden wird. Dies trifft hier auf den Knoten 19 zu. Dieser wird zum neuen lokalen Wurzelknoten.

Die Zieladresse wird mit der hinterlegten Netzwerkmaske ff ff 00 00 AND-verknüpft. Hieraus entsteht die neue Zieladresse 133.5.0.0. Dadurch ist die Suche vereinfacht, da nicht mehr mit einer exakten IP-Adresse der Vergleich durchgeführt wird, sondern mit dem Netzwerk-Teil der Zieladresse.

Die Vorwärtssuche wird daraufhin wieder eingeleitet und führt aufgrund des ungesetzten Bit 19 zum Blatt 133.5.0.0. Ein Vergleich der neuen Zieladresse führt dann zu einem Treffer. Dies wird (unter der Berücksichtigung der Korrektur der Zieladresse) "network match" genannt.

Wird auch bei der AND-Verknüpfung kein Treffer geliefert, wird die Rückwärtssuche (backtrack) aktiviert. Dabei wird so weit rückwärts gesprungen, bis ein Knoten mit hinterlegter Maske gefunden wird. Dieser wird als neue Wurzel des (Teil)baum gesetzt und die Zieladresse wird mit der Maske AND-verknüpft und ein neues Blatt zum Vergleich gesucht.

Dieser Prozess wird solange wiederholt, bis ein Treffer vorliegt. [05]

Suchtreffer "root match"

Als letztes Beispiel dient die Zieladresse 169.11.16.4. Hier wird das Szenario gezeigt, dass im Baum kein passender Knoten auch nach (mehreren) Rückwärts-Suchvorgängen gefunden wird. Letztendlich wird dann im Wurzelknoten die Default-Route als Ergebnis geliefert:

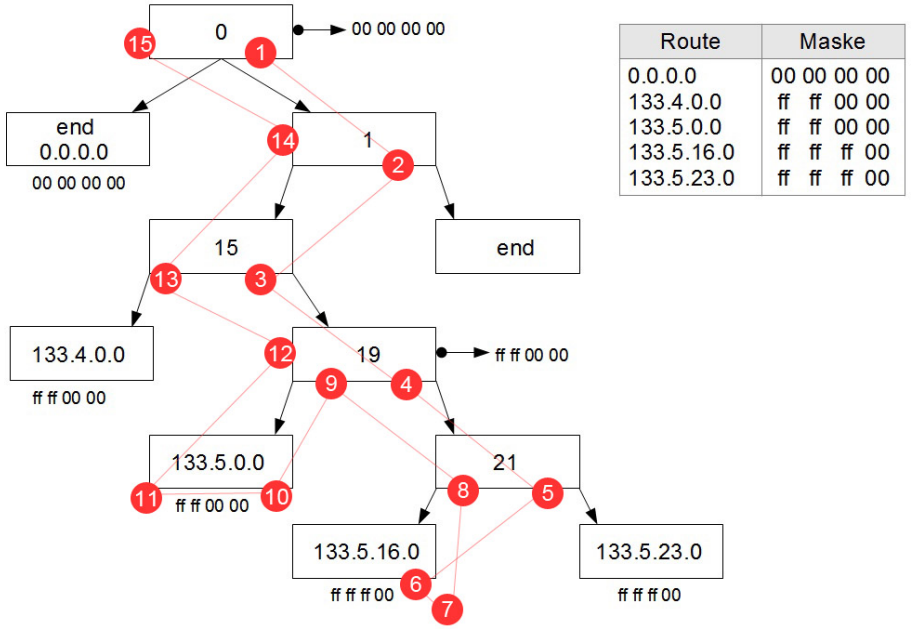


Abbildung 4, Radix Suchtreffer “root match“ [05]

Die Vorwärtssuche führt über Bit 0 (gesetzt), Bit 1 (ungesetzt), Bit 15 (gesetzt), Bit 19 (gesetzt), Bit 21 (ungesetzt) zum Blatt mit 133.5.16.0.

Der direkte Vergleich mit der Zieladresse und die folgende AND-Verknüpfung mit der Netzwerkmaske ff ff ff 00 führt zu keinem Treffer.

Bei der nun startenden Rückwärtssuche wird der Knoten 19 vorübergehend zum neuen lokalen Wurzelknoten, die Zieladresse wird durch die Verknüpfung mit der Netzwerkmaske ff ff 00 00 zu 169.11.0.0.

Die Vorwärtssuche endet bei Blatt 133.5.0.0. Aber auch hier erfolgt kein Treffer, eine erneute Rückwärtssuche wird notwendig. Der nächste Knoten mit hinterlegter Netzwerkmaske ist erst der ursprüngliche Wurzelknoten, der Knoten 0.

Die AND-Verknüpfung mit der Netzwerkmaske 00 00 00 00 ergibt die neue Zieladresse 0.0.0.0 und führt im Blatt 0.0.0.0 zum Treffer. [05]

4.2 Radish

Zur Vereinfachung des Radix-Algorithmus wurde der Radish-Algorithmus entwickelt. [05]

Als wesentlicher Vorteil zur simpleren Erstellung und Suche in Routing-Tabellen wird die Unterstützung für unterschiedliche und nicht zusammenhängende Adressbereiche entfernt. Diese waren im Radix implementiert, um Problematik der Einteilung von verschiedenen Netzadress-Bereichen zu umgehen.

Als grundsätzliche Ziele bei der Planung von Radish dienten drei Punkte:

- Einfachheit
- Nur Unterstützung von zusammenhängenden Adressbereichen
- Schnellerer Suchvorgang (lookup) als beim Radix

4.2.1 Spezifizierung des "longest prefix match"

Bei der Suche wird ein Vergleich mit einem initialen Substring durchgeführt. Dies bedeutet, dass nur ein beginnender Teil eines Strings als Substring zugelassen wird, aber kein mittlerer oder endender Substring.

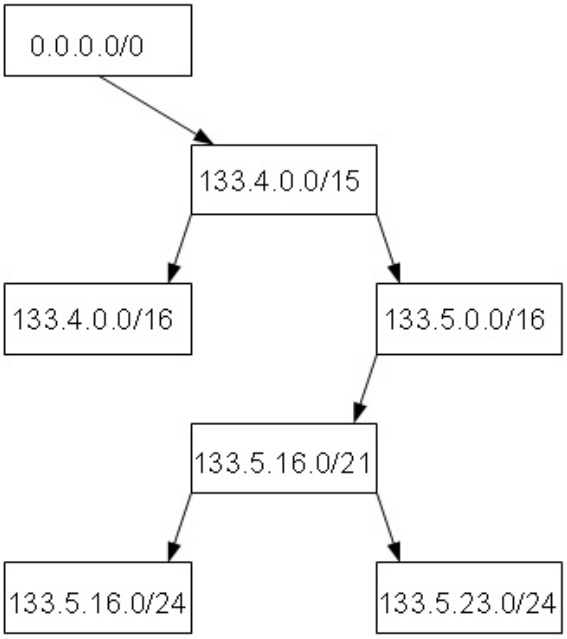
Als einfaches Beispiel zur Erläuterung: beim String „routing“ wäre der Substring „rou“ zulässig, der Substring „ting“ aber nicht. Mit Bezug auf IP-Adressen wären bei einer vorgegebenen Zieladresse von 133.5.16.2 die IP-Adressen 133.0.0.0 und 133.5.16.0 zulässige initiale Substrings.

Je länger der initiale Substring ist, desto besser ist das Trefferergebnis.

Der Aufbau der Radish Routing Tabelle ist ein binärer Baum, der Suchvorgang liefert als Ergebnis den längsten initialen Substring. Zur Vereinfachung und Beschleunigung

der Suche werden beim Baum die Knoten entfernt, bei denen keine Verbindung zu einer reellen Route besteht (virtueller Knoten) und nicht zwei „Kinder“ vorhanden sind. Jeder Knoten besitzt als Informationen eine Route und eine Maske.

Beispiel:



Route	Maske
0.0.0.0	00 00 00 00
133.4.0.0	ff ff 00 00
133.5.0.0	ff ff 00 00
133.5.16.0	ff ff ff 00
133.5.23.0	ff ff ff 00

Abbildung 5, Radish Beispiel [05]

Aus der rechts aufgeführten Tabelle entsteht dieser binäre Radish-Tree.

Ein Suchvorgang würde folgendermaßen ablaufen:

Der Start erfolgt an der Wurzel, und eine temporäre Markierung wird an der Wurzel gesetzt. Solange es Unterknoten gibt, wird die Suche fortgeführt.

Mittels der Markierung kann bei den verschiedenen Suchverfahren ein Knoten temporär hervorgehoben werden, um bei eventuellen Rückwärts-Suchvorgängen Pfadlängen zu reduzieren. Die Markierung dient dadurch als temporäre Speicherung des momentan besten Treffers bei der Suche.

An jedem Knoten werden zwei verschiedene Tests durchgeführt: der erste testet auf die korrekte Route anhand des Substrings, der zweite Test entscheidet, ob der nächste Punkt der Suche links- oder rechtsseitig vom Knoten erfolgt. [05]

Suchverfahren “Testen auf korrekte Route“

Dabei wird die Zieladresse mit der im Knoten hinterlegten Adresse verglichen. Ist die Zieladresse, AND-verknüpft mit der Maske, identisch mit der hinterlegten, ist die Route korrekt und eine temporäre Markierung wird auf diesen Knoten gesetzt.

Bei einem reduzierten Radish-Baum kann die Überprüfung noch vereinfacht werden, indem einzig die unterschiedlichen Bits zwischen parent und child verglichen werden. [05]

Suchverfahren “Testen auf links- oder rechtsseitig“

Bitweise wird getestet, ob die Suche links oder rechts fortgesetzt wird. Ist das Bit gesetzt (“1“), wird die Suche rechtsseitig fortgeführt. Bei einem ungesetzten Bit (“0“) folglich linksseitig. [05]

4.2.2 Lookup Beispiele

Basierend auf dem vorherigen Beispiel, dass beim Thema Radix erläutert wurde, soll hier der Lookup-Algorithmus mit mehreren Beispielen beschrieben werden:

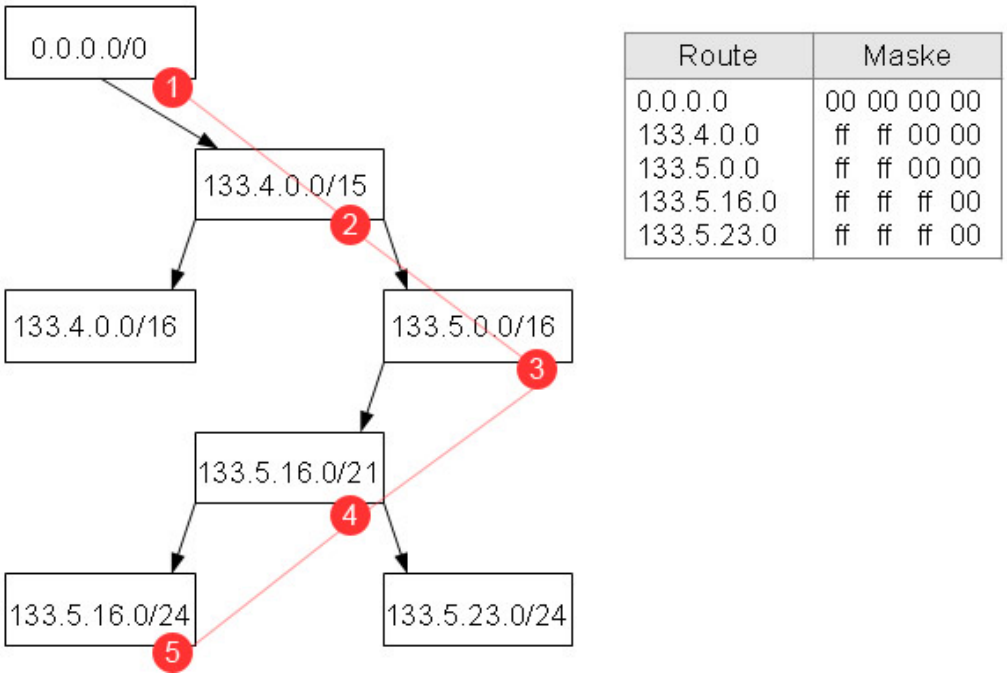


Abbildung 6, Radish Lookup [05]

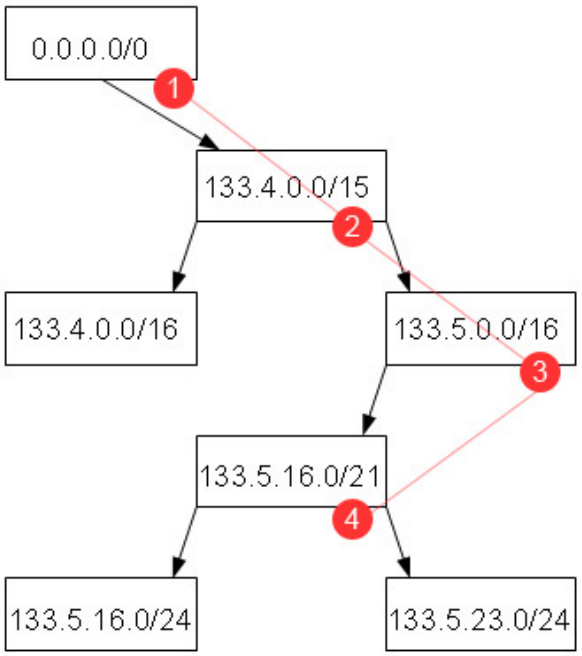
Bei einer Suche nach der Zieladresse 133.5.16.2 beginnt die Suche an der Wurzel. Die Zieladresse wird AND-verknüpft mit 00 00 00 00 und ergibt den Wert 0.0.0.0. Der zweite Teil des Testes liefert an dieser Stelle kein gesetztes Bit (bzw. es existiert nur ein Unterknoten).

Die nächste Überprüfung erfolgt bei 133.4.0.0/15. Der erste Test liefert einen Treffer der identischen Zieladresse (verknüpft mit der Maske) und hinterlegte Adresse, daher wird eine Markierung auf diesen Knoten gesetzt. Die bitweisen Überprüfung ergibt ein gesetztes Bit 15 und führt zu dem Knoten 133.5.0.0/16.

Hier wird nach dem ersten Test die Markierung gesetzt, und der zweite Test lässt linksseitig weitersuchen.

Die Adresse 133.5.16.0 ist immer noch Teil der passenden Route, die Markierung wird darauf neu gesetzt. Das ungesetzte Bit 21 leitet dann zu dem Blatt 133.5.16.0/24. Nach dem ersten Test wird die Markierung auf das Blatt gesetzt. Da hier keine Unterknoten (child) mehr existieren, ist die Suche beendet und liefert die Informationen vom Blatt.

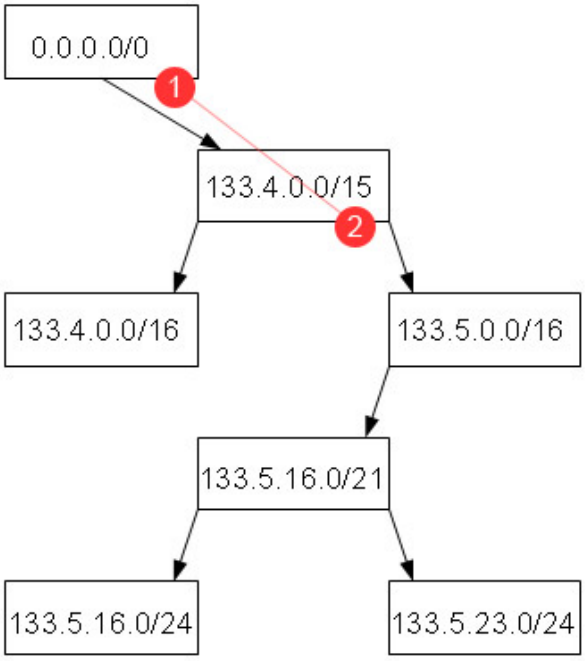
Eine Suchanfrage nach 133.5.80.9 führt zum Knoten 133.5.16.0/21. Der erste Test der AND-verknüpften Zieladresse (133.5.80.0) und der hinterlegten Adresse (133.5.16.0) ist nicht erfolgreich, daher wird die Suche unterbrochen und die Daten des Knotens übermittelt, bei dem die Markierung gesetzt war. In diesem Vorgang bei 133.5.0.0/16.



Route	Maske
0.0.0.0	00 00 00 00
133.4.0.0	ff ff 00 00
133.5.0.0	ff ff 00 00
133.5.16.0	ff ff ff 00
133.5.23.0	ff ff ff 00

Abbildung 7, Radish Lookup 2 [05]

Bei dem Test auf 169.11.16.4 ist die Suche schon beim zweiten Knoten beendet. Der Test mit 133.4.0.0 liefert keinen Erfolg, daher werden die Daten der Wurzel übermittelt. [05]



Route	Maske
0.0.0.0	00 00 00 00
133.4.0.0	ff ff 00 00
133.5.0.0	ff ff 00 00
133.5.16.0	ff ff ff 00
133.5.23.0	ff ff ff 00

Abbildung 8, Radish Lookup 3 [05]

4.2.3 Konstruktion des Radish – Tree

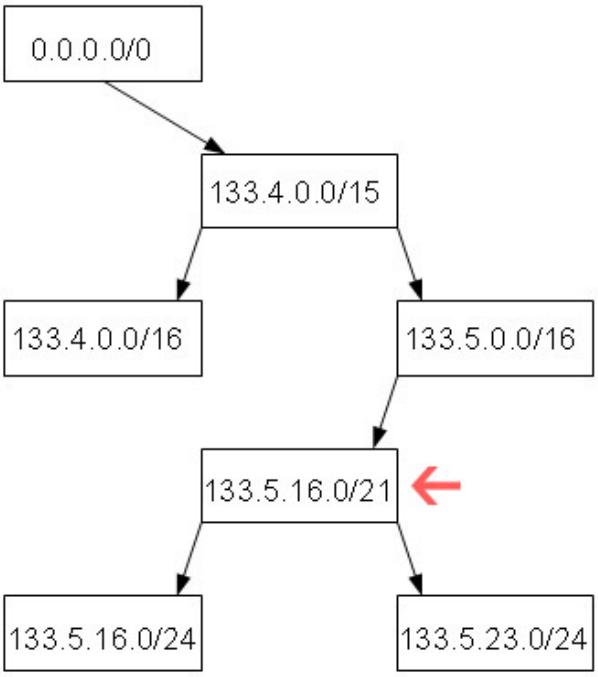
Vereinfacht gesagt gibt es drei verschiedene Möglichkeiten, in einen bestehenden Tree weitere Knoten bzw. Blätter einzufügen.

Dazu wird die Adresse des neuen Eintrags für einen Suchvorgang verwendet, der die passende Stelle innerhalb des Baumes ermittelt.

Die drei verschiedenen Muster werden anhand von Beispielen erklärt:

Variante 1 – Eintrag ist identisch

Es soll ein Knoten mit 133.5.16.0/21 in den Baum eingefügt werden. Dazu wird bei der Suche die passende Stelle ermittelt, die in diesem Fall identisch mit dem neuen Eintrag ist. Daher wird der bisherige Eintrag als neuer Eintrag markiert bzw. keine Änderung in der Struktur des Baumes vorgenommen. [05]



Route	Maske
0.0.0.0	00 00 00 00
133.4.0.0	ff ff 00 00
133.5.0.0	ff ff 00 00
133.5.16.0	ff ff ff 00
133.5.23.0	ff ff ff 00

Abbildung 9, Radish Konstruktion, identischer Eintrag [05]

Variante 2 – neuer Knoten

Ein neuer Eintrag mit der Adresse 133.5.19.0/24 wird in den vorhandenen Baum eingesetzt. Dazu wird zwischen die Knoten mit der Adresse 133.5.16.0/21 und 133.5.16.0/24 ein Knoten mit 133.5.16.0/22 eingebaut, der die neue Adresse dann als Blatt bzw. Unterknoten einsetzt. Durch die Anpassung der Maske ist die Vollständigkeit des Baumes gegeben[05].

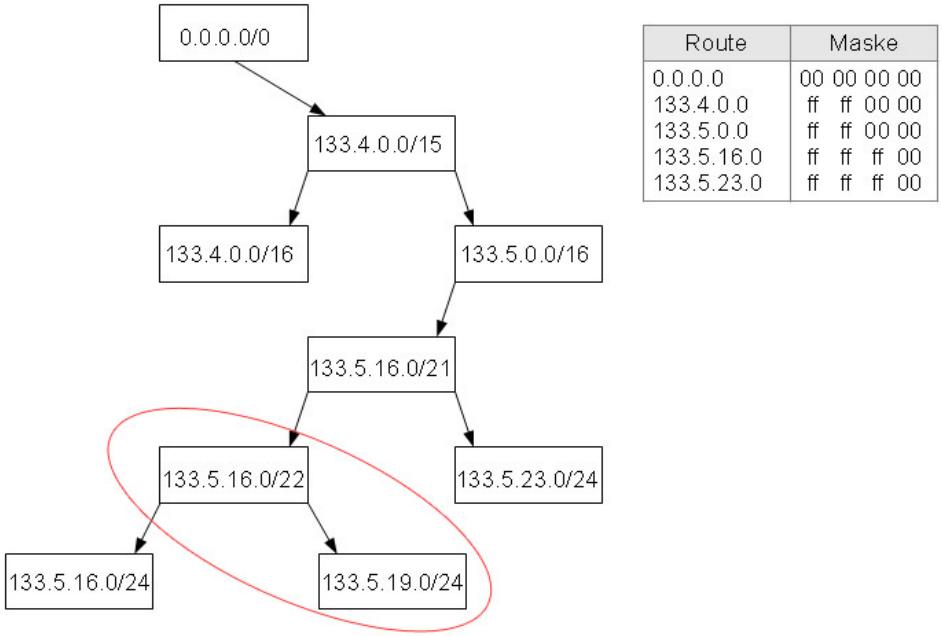


Abbildung 10, Radish Konstruktion, neuer Knoten [05]

Variante 3 – neues Blatt

Alternativ kann an ein vorhandenes Blatt noch ein weiteres Blatt angefügt werden. Beispielsweise die Adresse 133.4.1.0/24 als neues Blatt unterhalb des früheren Blattes 133.4.0.0/16.

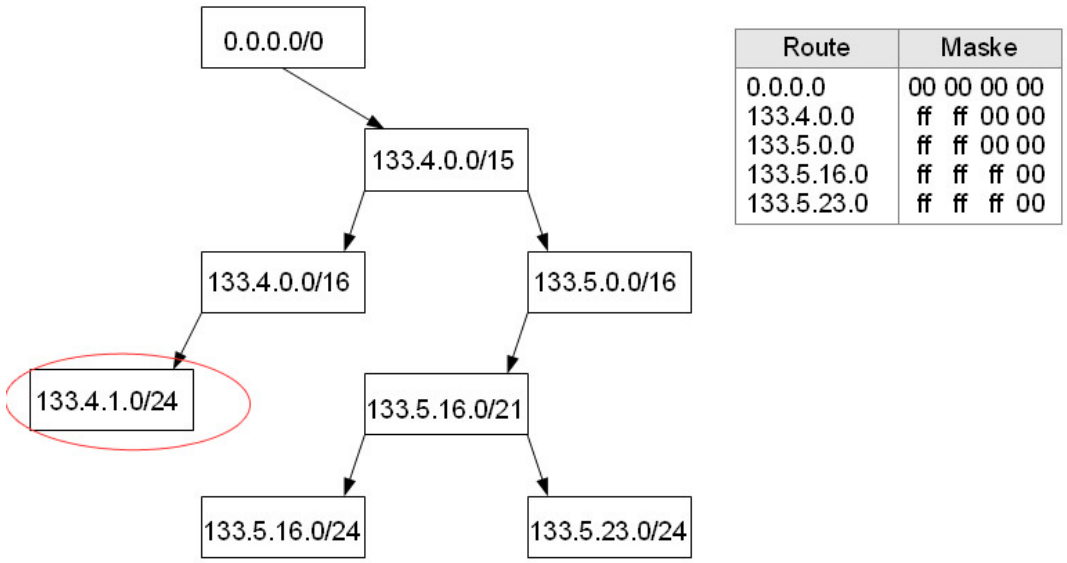


Abbildung 11, Radish Konstruktion, neues Blatt [05]

5. Full expansion / compression

Ein experimenteller Ansatz führte zur Methodik full expansion / compression, bei dem ein Multibit-Baum mit Adressen aus einer Forwarding-Tabelle generiert wird. Zur besseren Verarbeitung bzw. schnellen lookup-Abfrage wird dieser Multibit-Baum in mehrteiligen Verfahren komprimiert und in eine Daten-Struktur überführt, die aus einem Multibit-Baum und Tabellen besteht.[16]

Die Vorteile der Methode sind hauptsächlich [16]:

- Anwendbar auf (fast) alle Router
  Getestet wurde die Anwendung mit verschiedensten Router, unter anderem Backbone-Router mit einer hohen Anzahl von Einträgen als auch lokal benutzte Router, die deutlich weniger Prefix-Einträge enthalten
- Anwendbar auf längere Zeit
  Die Anwendung der Methode erfolgte über mehrere Monate unter Benutzung realer Daten und wies keine signifikanten Unterschiede in der Effizienz aus
- Geringe und garantierte Anzahl von Speicherzugriffe
  Jede Überprüfung auf einen Prefix benötigt genau drei Speicherzugriffe. Mittels der Adressierung von drei benötigten Tabellen garantiert die Umsetzung für den lookup ein schnelles Ergebnis.
- Umsetzbar in Hardware
  Der lookup Algorithmus ist vergleichsweise simpel und damit mit wenig Aufwand als Hardware zu realisieren

5.1 Multibit-Bäume

Ein Multibit-Baum erlaubt im Gegensatz zu einem binären Baum die gleichzeitige Überprüfung bzw. Speicherung auf mehrere Bits. Bei einem binären Baum und einer 32-bit Adresse können beispielsweise im ungünstigsten Fall 32 Speicherzugriffe

entstehen, falls die Adresse komplett als Pfad abgebildet ist, da pro Ebene nur auf ein Bit getestet werden kann [17]. In einem Multibit-Baum hingegen gibt es pro Knoten eine bestimmte Anzahl von Kindern, bezeichnet mit 2^k. (k steht für die Anzahl der gleichzeitig getesteten Bits). Ist z.B. k=8, reduziert diese die Adresssuche auf 4 Speicherzugriffe im ungünstigsten Fall, da der zugehörige Multibit-Baum eine Tiefe von 4 Ebenen besitzt.

Nachteilig in einem Multibit-Baum ist die Umsetzung der Speicherung von verschiedenen Bit-Längen (Prefix-Längen), die nicht ein die Länge k besitzen bzw. ein Vielfaches der Länge k sind. Zum vorangegangenen Beispiel mit k=8 muss für eine vorgegebene Prefix-Länge mit 7 Bit noch eine Erweiterung auf das 8. Bit erfolgen, um den Prefix in dem Multibit-Baum abbilden zu können. [17]

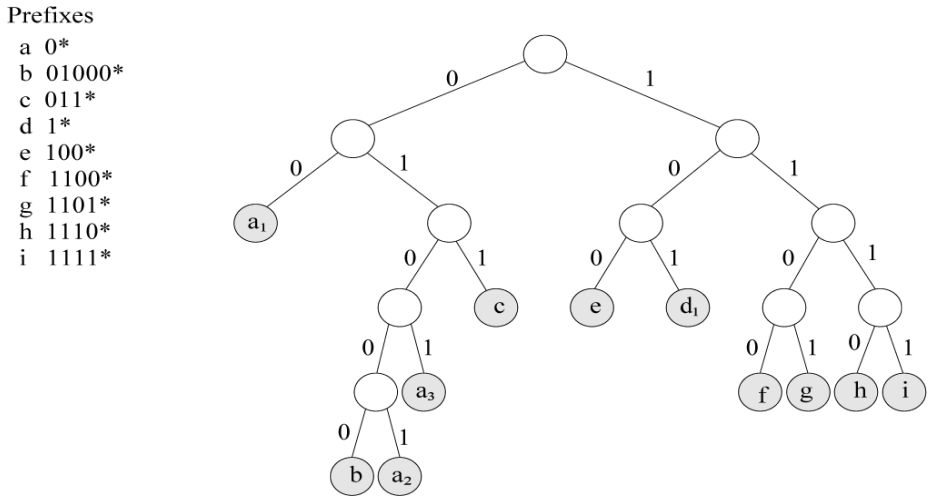


Abbildung 12, Binärer Baum [17]

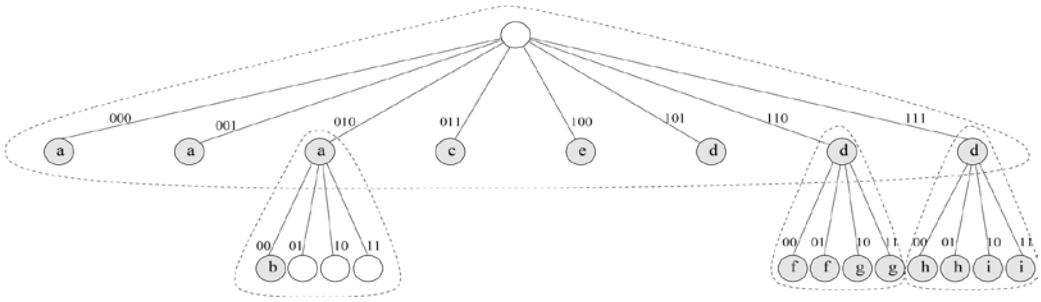


Abbildung 13, Multibit-Baum [17]

In den oben aufgeführten Abbildungen wird der Unterschied zwischen einem binären und einem Multibit-Baum dargestellt:

Die Darstellung in einem Multibit-Baum (Abbildung 13) ist umfangreicher bei der Anzahl der Knoten bzw. Blättern durch die Ausbalancierung. Hingegen ist die jeweilige Pfadlänge im binären Baum (Abbildung 12) sehr unterschiedlich. Dies kann nachteilig in Suchverfahren sein, da im schlechtesten Fall eine Tiefe des Baumes erreicht wird, die identisch mit der Länge der maximal verfügbaren Adresse ist.

5.2 Funktionsweise full expansion / compression

Mit der Methode full expansion / compression wird bei einer vorgegebenen forwarding Tabelle im ersten Schritt die Abbildung auf einen Multibit-Baum vorgenommen und fehlende Pfade bzw. Kinder ergänzt (expansion). Im zweiten Schritt wird dieser dann ab einer definierten Bit-Länge in einen Multibit-Baum mit 2 Level unterteilt, bei dem in der 2. Ebene identische Teil-Strecken zusammengelegt werden können. (compression)

Die dadurch entstehende Struktur ist ein Multibit-Baum, der auf der 2. Ebene tabellarisch die Einträge verwaltet.

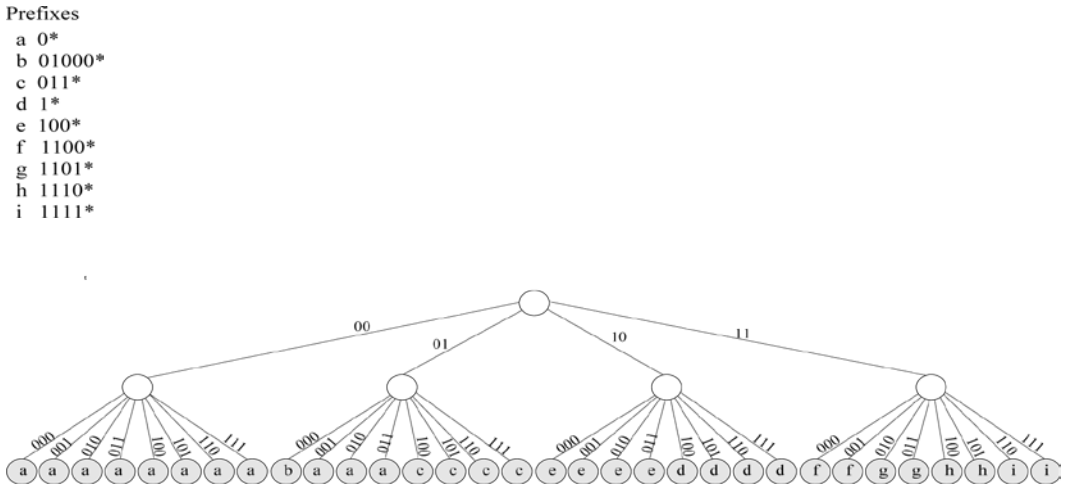


Abbildung 14, kompletter Multibit-Baum [17]

Abbildung 14 zeigt einen komplett erweiterten Multibit-Baum, dessen erster Adressteil mit 2 Bit unterschieden werden. Diese Erweiterung entspricht der expansion-Phase.

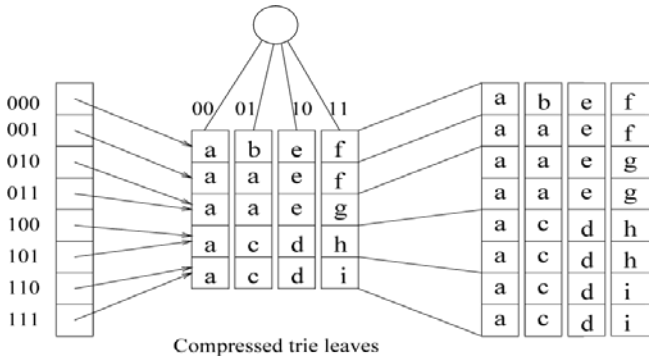


Abbildung 15, Beispiel expansion / compression – Struktur [17]

Als Ergebnis der anschließenden compression-Phase wird eine Struktur wie beispielsweise in Abbildung 15 generiert. Dabei ist der obere Teil identisch mit der Multibit-Baum-Darstellung aus Abbildung 14, die 2 Bit der Adresse darstellen. Der zweite Adress-Teil wird in einer Liste (linksseitig dargestellt) gespeichert, die zusammen auf eine Zelle in der mittleren Tabelle verweisen, indem die passenden Interfaces notiert sind. Rechtseitig ist die komplette Tabelle der Interfaces vorhanden, die aber aufgrund identischer Verweise (siehe 010 und 011, Verweis auf Tabellenzeile aaeg) komprimiert in der mittleren Tabelle gespeichert werden können.

Formale Definition

Zu einem binären Alphabet $\sum = \{0, 1\}$ gibt es eine Adresse mit m Bit-Stellen. Dann ist die Menge aller binären Bitfolgen mit der Länge k definiert durch $\sum_k$ mit $\sum \leq$ Vereinigung von $\sum_k$ mit k=0 bis m.

Zwei Bitfolgen α und β der Menge $\sum_k$ besitzen die Länge $k_\alpha = |\alpha|$ und $k_\beta = |\beta|$. Dann wird α als Prefix von β bezeichnet, gdw. die ersten $k_\alpha \leq k_\beta$ Bits von β identisch mit α sind.

Beispiel: α = 001 ist Prefix von β = 0011

Die Verknüpfung von α und β ist definiert durch α * β.

Beispiel: mit α = 001, β = 0011 ergibt α * β = 001 0011

Eine Bitfolge α und eine Teilmenge S von $\sum \leq$ m ist a * S definiert mit
a * S = {x | x = α * β mit β ∈ von S}.

Eine Forwarding Tabelle T für Adressen der Länge m besteht aus einer Menge von Paaren (p, h), wobei p eine Bit-Folge mit der Länge ≤ m ist (Prefix) und h die Interface-Angabe. Die Gesamtanzahl der Einträge wird mit |T| angegeben. Die Anzahl ist mindestens eins aufgrund des Standard-Eintrages (ε, h_ε), der den default next-hop beschreibt.[16]

5.3 Expansions-Phase

In der ersten Phase wird eine vorhandene Forwarding-Tabelle sortiert und soweit ergänzt, dass sich ein kompletter Multibit-Baum abbilden lässt. Dies bedeutet, dass alle Einträge in der Tabelle, deren Länge kleiner als die Adress-Länge m ist, auf die notwendige Länge m erweitert werden unter Berücksichtigung des angegebenen Interfaces.

Route	Interface
ε	C
10	B
001	B
01	C
0010	C
00	A

Abbildung 16, Bsp. Forwarding-Tabelle 01 [16]

Route	Interface
10	B
01	C
0010	C
001	B
00	A
ε	C

Abbildung 17, Bsp. Forwarding-Tabelle 02 [16]

Als Beispiel dient eine Forwarding-Tabelle, die in Abbildung 16 zu sehen ist. Die absteigende Sortierung liefert die Tabelle aus Abbildung 17.

Die sortierte Liste wird anschließend um eine Spalte erweitert, die eine Bezeichnung zur Gruppierung einführt. Diese wird mit T_1, T_2, …, $T_{|T|}$ nummeriert.

Route	Interface	$T_{(x)}$
10	B	T_1
01	C	T_2
0010	C	T_3
001	B	T_4
00	A	T_5
ε	C	T_6

Abbildung 18, Bsp. Forwarding-Tabelle 03 [16]

Die Funktion EXP (Ti) = (pi * Em-|pi|) x {hi} für $1 \leq i \leq |T|$ erzeugt die Menge der Erweiterungen der jeweiligen Prefix-Einträge in der Tabelle, die mit T‘ bezeichnet werden.

Die Teilmengen T’i sind definiert als: T‘1 = EXP(T1) und T’i= EXP(Ti) @ U1≤j≤i T’i.

Der Zusatz @ vermeidet redundante Einträge, indem vorhandene Prefix-Erweiterungen nicht nochmals in nachfolgenden Teilmengen erzeugt werden. Ist z.B. in T3 ein Prefix 0010 vorhanden, wird dieser in T4 nicht nochmals eingetragen.[16]

Route	Interface	$T'_{(x)}$
1000	B	T'_1
1001	B	
1010	B	
1011	B	
0100	C	T'_2
0101	C	
0110	C	
0111	C	
0010	C	T'_3
0011	B	T'_4
0000	A	T'_5
0001	A	
1100	C	T'_6
1101	C	
1110	C	
1111	C	

Abbildung 19, Bsp. Forwarding-Tabelle 04 [16]

Als Ergebnis wird die oben aufgeführte Tabelle erzeugt, die nun alle notwendigen Erweiterungen beinhaltet. Als Resultat dieser Phase lässt sich festhalten:

Wenn $(p_x, h_x) \in$ von T das Ergebnis des IP lookup-Prozesses für alle m-bit langen Adressen x ist, dann ist $(x, h_x) \in$ von T‘. Anders formuliert: wird bei einer Forwarding-Tabelle (p_x, h_x) als Ergebnis einer Adress-Abfrage geliefert, muss der Eintrag (x, h_x) in der erweiterten Tabelle vorhanden sein.[16]

Im nächsten Schritt erfolgt die Komprimierung von Teilmengen der Tabellen.

5.4 Compression - Phase

Als ersten Schritt in der Komprimierungsphase werden abhängig von einem bestimmten Wert k (mit $1 \leq k \leq m$) die Adressen aus der erweiterten Tabelle in zwei Teile zerlegt, einen α- und einen β-Teil.

Insgesamt entstehen somit drei Tabellen in der Phase, bestehend aus den Tabellen mit den α- und β-Einträgen und der dazugehörigen (Ergebnis-)Tabelle mit den Interfaces. Die Splittung der Adresse erlaubt eine effektivere Suche, bei der zuerst in den Teiltabellen die Adresse überprüft wird und dann auf das passende Interface verweist.

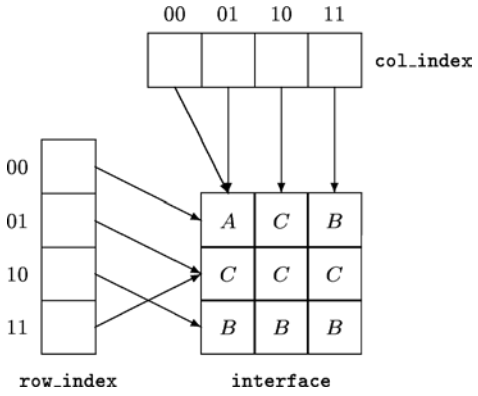


Abbildung 20, compression-Struktur [16]

Vergleichbar mit der Abbildung 15 ist in Abbildung 20 das Ergebnis der compression-Phase dargestellt: In der Liste col_index sind die Bitfolgen vorhanden, die den ersten Teil der vollständigen Adresse ergeben. Bei dem Beispiel ist k = 2 gewählt, daher ist die Liste row_index (2. Hälfte der Adressen) identisch mit col_index. Die Suche in den zwei Listen verweist dann auf das passende Interface, die in der Tabelle interface gespeichert sind. Dabei ist zu beachten, dass die die Größe der Tabelle durch die Anzahl von α und β bestimmt wird.[16]

Definition:

Bei einer Tabelle T mit Adressen der Länge m, ist die Menge k-size von T mit k $\leq$ m definiert mit

$\alpha_k = \left|\{ T'_{(x)} \mid x \in \sum_k \}\right|$

Die dabei entstehende Menge ist eine erste, einfache Form der Komprimierung. Die Anzahl der in der Menge enthaltenen Elemente ist kleiner bzw. gleich groß wie die Menge der Prefixe mit der Länge k, die anfangs in der Tabelle T enthalten sind. Sind also in der Tabelle T beispielsweise n Prefixe vorhanden, deren Länge $\leq$ k ist, sind in der später entstehenden Tabelle α_k niemals mehr Elemente als n enthalten.[16]

Formal beschrieben:

Bei einer gegebenen Tabelle T sei r_k die Anzahl der Interfaces, die in Einträgen mit der maximalen Länge k vorkommen. N_k ist die Anzahl der Routing-Einträge, die länger als k sind, mit $1 \leq k \leq m$. Dann gilt: $\alpha_k \leq r_k + n_k \leq |T|$

Beim folgenden Schritt wird ein Schema angewandt, was mit run length encoding (RLE) bezeichnet wird. Dies komprimiert die Anzahl der Einträge in den jeweiligen Teilmengen $T'_{(x)}$ in zwei Anwendungsschritten:

1. Sortiere $T'_{(x)}$ in aufsteigender Reihenfolge und nummeriere die Paare basierend nach dem jeweiligen Rang. $T'_{(x)} = \{(y_i, h_i)\}\ 1 < i < 2^{m-k}$

2. Ändere $T'_{(x)}$ in $s_{(x)}$, indem bei jedem maximalem Durchlauf (y_i, h_i), (y_{i+1}, h_{i+1}), …, (y_j, h_{i+1}) eine Ersetzung erfolgt bei $h_i = h_{i+1} = \ldots = h_{i+1}$ mit dem Paar $[h_i, {}_{l+1}]$. L+1 wird als run length von h_i bezeichnet.

Das Verfahren wandelt eine Menge von Einträgen, die in $T'_{(x)}$ enthalten sind, in eine einzige Sequenz. α_k entspricht damit der Anzahl der entstandenen Sequenzen $s_{(x)}$.[16] Beispielsweise auf die vorherige Tabelle bezogen:

Route	Interface	T‘(x)
1000	B	T‘$_1$
1001	B	
1010	B	
1011	B	
0100	C	T‘$_2$
0101	C	
0110	C	
0111	C	
0010	C	T‘3
0011	B	T‘4
0000	A	T‘5
0001	A	
1100	C	T‘6
1101	C	
1110	C	
1111	C	

Abbildung 21, Bsp. Forwarding-Tabelle 05 [16]

α	Route	Interface	T‘(x)
00	**00**00	A	T‘$_5$
	0001	A	T‘$_5$
	0010	C	T‘$_3$
	0011	B	T‘$_4$

Abbildung 22, *α*-Tabelle [16]

$T_{(00)}$ = {(00, A), (01, A), (10, C), (11, B)}

ergibt dann die Sequenz

[$S_1 = \langle A, 2\rangle \langle C, 1\rangle \langle B, 1\rangle$]

Die Einträge aus der Tabelle T‘ (Abbildung 21) mit dem Prefix α werden aufsteigend sortiert und dann als einzelne Sequenz ausgegeben. In diesem Beispiel ist nur α = 00 (Abbildung 22) angegeben. Die Sortierung und Sequenz-Erstellung für die anderen α – Werte erfolgt analog.

Die Sequenzen müssen anschließend eine einheitliche Länge aufweisen. Daher wird noch ein weiterer Bearbeitungsschritt angewendet, der die verschieden langen Sequenzen anpasst, bezeichnet mit s‘$_{(x)}$. Dies erfolgt durch eine Funktion φ (s, t), definiert mit

$s = \langle a, f\rangle * s_1$

$t = \langle b, g\rangle * t_1$

und

$$\varphi(s,t) = \begin{cases} t & \text{if } s_1 = t_1 = \varepsilon, \\ \langle b, f\rangle * \varphi(s_1, t_1) & \text{if } f = g \text{ and } s_1, t_1 \neq \varepsilon, \\ \langle b, f\rangle * \varphi(s_1, \langle b, g - f\rangle * t_1 & \text{if } f < g \text{ and } s_1, t_1 \neq \varepsilon, \\ \langle b, f\rangle * \varphi(\langle a, f - g\rangle * s_1, t_1) & \text{if } f > g \text{ and } s_1, t_1 \neq \varepsilon. \end{cases}$$

Die Funktionsweise besteht in einer Aufteilung von einem Paar $\langle b, f \rangle$ in eine Menge von Paaren $\langle b, f_1\rangle$, …, $\langle b, f_n\rangle$, um bei zwei RLE Sequenzen mit einer gleichen Länge zu erhalten [16]. Da diese Funktion sich nur auf 2 Sequenzen bezieht, muss mit einer rekursiven Funktion Φ die Vereinheitlichung aller Sequenzen noch durchgeführt werden:

$\Phi = (s_1, \ldots, s_q)$ mit den Sequenzen $s_1, s_2, \ldots, s_q$

Als Ergebnis werden vereinheitlichte Sequenzen $s'_1, s'_2, \ldots, s'_q$ geliefert durch folgenden rekursiven Aufbau [16]:

$$s'_q = \varphi(\varphi(\ldots \varphi(\varphi(\varphi(s_1, s_2), s_3), s_4)\ldots, s_{q-1}), s_q)$$

Dies führt zu folgender Definition:

> Gegeben sei eine Forwarding-Tabelle T mit Adressen der Länge m. Dann entspricht der Wert β_k von T mit $1 \leq k \leq m$ der Länge der vereinheitlichten RLE-Sequenzen durch die Funktion $\Phi = (s_1, \ldots, s_q)$, $\beta_{k=|}s'_{\alpha k}|$.[16]

Der Ablauf zusammengefasst:

1. Forwarding-Tabelle sortieren (absteigend)
2. Zusätzliche Spalte T'_i zur Gruppierung ergänzen
3. Einträge in der Tabelle erweitern (zu 2^m)
4. Logische Teilung der Adressen in 2 Teile durch gewähltes k
5. αk ermitteln
6. Sequenzen bestimmen und vereinheitlichen
7. β_k ermitteln
8. neue Struktur aufbauen

Die Auswahl eines passenden k bestimmt maßgeblich die Effizienz der Suche, indem der Wert die Struktur bzw. Größe der Tabellen vorgibt, die durch die Teilung der Adresse entstehen. Der Algorithmus wurde basierend auf realen Daten aus den Jahren

1998 / 1999 (Abbildung 23) entwickelt und lässt darauf schließen, dass für k ein Vielfaches von 8 gewählt werden sollte, da die meisten Prefixe ein Länge von 16 oder 24 Bit besitzen.

Router	Length 16	Length 24	Next
MaeEast	13%	56%	8% (23 bits)
MaeWest	14%	53%	7% (23 bits)
Aads	25%	54%	8% (23 bits)
PacBell	13%	56%	7% (23 bits)
Paix	12%	53%	8% (23 bits)

Abbildung 23, Prefix-Verteilung Router [16]

5.5 Effizienz-Vergleich

Anhand realer Daten aus 5 verschiedenen Routern (Abbildung 24) sind folgend Ergebnisse für die Anwendung des Schemas aufgezeigt. Dabei wurde k = 16 Bit ausgewählt, um eine Balance der 2 Tabellen zu erhalten, die die jeweiligen Adress-Teile speichern.

Die Router sind exemplarisch so ausgewählt, dass auch ein Backbone-Router mit hoher Prefix-Anzahl (“MaeWest“) und ein lokaler Router mit relativ wenigen Prefix-Einträgen (“Paix“) bei der Analyse mit einbezogen wurden.[16]

Router	Jul. 7, 1998	Jan. 11, 1999	Min/Max	*H*
MaeEast	41231	43524	37134/44024	62
MaeWest	18995	23411	17906/23489	62
Aads	23755	24050	18354/24952	34
PacBell	22416	22849	21074/23273	2
Paix	3106	5935	1519/5935	21

Abbildung 24, Prefix-Anzahl Router [16]

Die zwei Spalten mit Datum geben die zum jeweiligen Zeitpunkt vorhandenen Einträge in der Forwarding-Tabelle an. H steht für die Anzahl von Interfaces.

Die Anwendung des Schemas ergibt folgende Werte für die Parameter α und β:

Router	Jul. 7, 1998 (α_k/β_k)	Jan. 11, 1999 (α_k/β_k)	Maximum (α_k/β_k)
MaeEast	2577/277	2745/299	2821/299
MaeWest	2017/263	2335/268	2335/285
Aads	1903/259	2100/269	2140/273
PacBell	1399/256	1500/256	1500/260
Paix	722/256	984/261	989/261

Abbildung 25, α – β – Verteilung Router [16]

Die Schlussfolgerung aus der Vergleichstabelle der Abbildung 25 ergibt, dass sowohl die α- als auch die β-Daten mit maximal 2 Bytes speicherbar sind. Für die Anzahl der Interfaces genügt ein Byte als Speicheraufwand.[16]

Der reale Speicheraufwand lässt sich in folgender Vergleichstabelle ermitteln (Angaben in Bytes):

Router	Jul. 7, 1998 (M_1/M_2)	Jan. 11, 1999 (M_1/M_2)	Maximum (M_1/M_2)
MaeEast	975973/1107045	1082899/1213971	1082899/1213971
MaeWest	792615/923687	887924/1018996	887924/1018996
Aads	755021/886093	827044/958116	837804/968876
PacBell	620288/751360	646144/777216	646424/777496
Paix	446976/578048	518968/650040	518968/650040

Abbildung 26, Speicherbedarf Router [16]

Der Router MaeEast überschreitet zeitweise die Grenze von 1 Megabyte zur Speicherung der Struktur, was bei der Analyse nachteilig sich auswirkt, da der damalige L2-Cache des Prozessors eine Größe von einem Megabyte aufwies.[16]

Eine effiziente Suche wird also möglich, falls die Datenstruktur zur Suche komplett in einem Cache vorgehalten werden kann.

6. Analyse des Algorithmus LPFST

Das nun folgende Suchverfahren **LPFST** ist auch baumbasiert, besitzt darüber hinaus aber einige Mechanismen zur Verringerung der Knoten-Anzahl innerhalb des Baumes. Grundlegend verschieden zu den zuvor aufgeführten Algorithmen ist die Sortierung des Baumes: die längsten Prefixe befinden sich direkt unterhalb der Wurzel, nicht in den Blättern.

LPFST steht für “Longest Prefix First Search Tree”. Der Algorithmus soll gegenüber anderen Baumstrukturen eine geringere Anzahl von Knoten aufweisen bei gleicher Informationsdichte. [04].

Durch LPFST wird die Routing-Tabelle so aufgebaut, dass bei einer Suche nach einem Eintrag der längst möglichste Prefix beim IP Lookup als Treffer geliefert werden soll.

Der dabei konstruierte Baum ist binär aufgebaut, pro Knoten gibt es 2 Möglichkeiten zum nächsten Unterknoten zu gelangen. Je länger ein Prefix ist, desto höher ist die Rangordnung im Baum bzw. die Nähe zur Wurzel. Eine weitere Besonderheit ist die Möglichkeit, in einem Knoten mehrere IP-Einträge abzuspeichern und dadurch die Größe zu minimieren.

6.1 Knoten-Struktur

Bei einem Baum als Speicherstruktur sind die Informationen über die Entscheidung, welche Seite von einem Knoten ausgehend gewählt wird, normalerweise direkt im Knoten hinterlegt. Bei LPFST existieren zwei verschiedene Knotentypen (**type0** und **type1**).

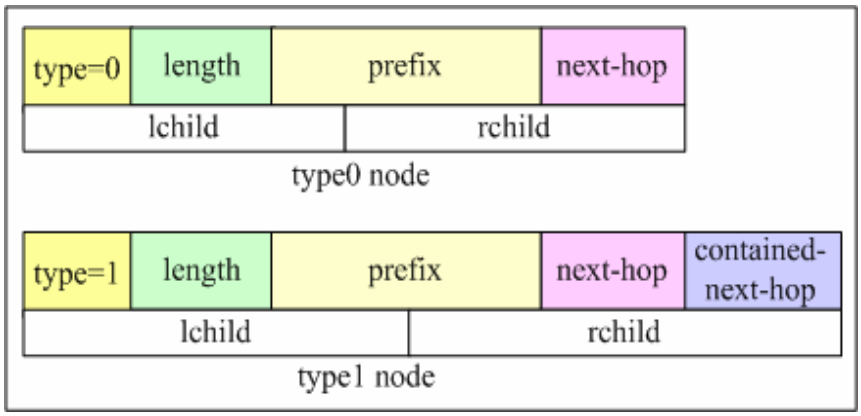


Abbildung 27, Aufbau eines LPFST-Knoten [04]

Erklärung Struktur:

- type: 0 oder 1
  Der **Typ 0 – Knoten** enthält einen Speicherplatz für nur genau eine next-hop-Adresse. Dies trifft auf die meisten Knoten zu.
  Der **Typ 1 – Knoten** hingegen besitzt zwei Speicherplätze für next-hop-Adressen (“contained-next-hop“)

- length: Länge des Prefix

- prefix: Prefix eines Eintrages in der Routing-Tabelle

- next-hop: next-hop des Prefix

- lchild / rchild: Verweis auf die linkes oder rechtes Unterzweig

Die Verwendung des type1-Knoten ergibt sich durch die folgende Definition 3. Dabei können 2 aufeinanderfolgende Knoten bzw. deren next-hop-Adressen in einem Knoten abgespeichert werden, wenn die next-hop-Adresse des unteren Knoten in dem oberen Knoten vollständig erhalten ist.

Ein Beispiel dazu: gegeben sei ein Baum mit mehreren Typ 0 – Knoten, der schon nach der Länge der jeweiligen Prefixe sortiert ist. Je länger also der Prefix in einem Knoten ist, desto näher befindet er sich im Verhältnis zu den anderen Knoten zur Wurzel.

Beispielsweise ist der Knoten mit der next-hop-Adresse 010 aufgrund der Prefix-Länge oberhalb des Knoten mit der next-hop-Adresse 01. Dann kann zur Reduzierung der Anzahl der Knoten im Baum aus den beiden einzelnen Knoten des Typ 0 ein Knoten mit zwei abgespeicherten next-hop-Adressen konstruiert werden, der Typ 1 – Knoten.

6.2 Definitionen

Zur Konstruktion des Baumes existieren 3 grundlegende Definitionen: Mittels der **ersten Definition** wird ein bestimmter Teil einer Adresse extrahiert. Dieser Vorgang wird für die **zweite Definition** benötigt, um 2 Adressen auf einen gleichen Prefix zu überprüfen. Wird ein zusätzlicher Prefix aus einem Eintrag der Routingtabelle in den Baum eingefügt, kann aus einem Typ-0-Knoten ein Typ-1-Knoten werden, falls dieser Definition 3 erfüllt.

Definition 1: Die Funktion Get(X, a, b) liefert den Wert vom a-ten Bit zum b-ten Bit der Variable X. Das 0-te Bit ist das linkeste mögliche der Variable X. Beispiel: Get(‘01101‘,1, 4) liefert ‘1101‘

Definition 2: Ein Prefix (P1/L1) wird übereinstimmend mit einem anderen Prefix (P2/L2) unter der Voraussetzung L1>L2 genannt, wenn (und genau nur dann) die Funktion Get(P1, 0, L2-1) identisch mit P2 ist.

Definition 3: Angenommen, es existieren zwei Einträge in der Routing-Tabelle (P1/L1; o1) und (P2/L2; o2). “o“ bezeichnet den Ausgangsport. Es gelte L1 > L2. Dann kann der Knoten als Typ 1 zwei Einträge enthalten (und nur genau dann) wenn der bisherige Knoten im Baum-Level von L2 existiert und P1 mit P2 laut Def. 2 übereinstimmt. Der Knoten vom Type 1 wird folgendermaßen aufgebaut: (1, L1, P1, o1, o2) [04]

6.3 Konstruktion des LPFST

Grundsätzlich werden bei der Erzeugung des Baumes linear die Eintragungen in der Routingtabelle ausgelesen und daraus die Knoten des Baumes erzeugt. Je nach Länge des Eintrages in der Tabelle entscheidet dies über die Position in dem Baum.

Grundschema:

```
{
lese den ersten Eintrag in der Routing-Tabelle und erzeuge im Baum den
ersten Eintrag;
while (mehr Einträge in der Routing Tabelle existieren)
        {
          lese den Eintrag in der Routing-Tabelle (Px / Lx; ox);
          erzeuge einen Typ-0-Knoten mit x=(0, Lx, Px, ox);
          insert (x, root, 0);
        }

}
```

Algorithmus “insert“:

```
{
 if Lx>Ly then swap (x, y); //tausche length, prefix,
                            // next-hop von x und y
 if Lx=level then {ändere y zu Typ-1-Knoten; y = (1,
  Ly, Py, oy, ox); return;}
 if Get(Px, level, 1) = 0 then
        {
    if lchild(y)≡NIL then lchild(y)=x;
    else insert(x, lchild(y), level+1);
         return;
        }
 else
        {
    if rchild(y)≡NIL then rchild(y)=x;
         else insert(x, rchild(y), level+1);
         return;
        }
```

6.4 Beispiel LPFST

Anhand des folgenden Beispiels wird die Funktionsweise des LPFST-Algorithmus erläutert. Linksseitig ist eine vorhandene Routing-Tabelle, analog sieht man rechts dazu den Aufbau des Baumes durch den Algorithmus.

Den aktuellen Fortschritt verdeutlicht der Pfeil neben der Routing-Tabelle. Zur Vereinfachung sind die ersten Schritte ausführlich dargestellt, danach das Endergebnis. [04]

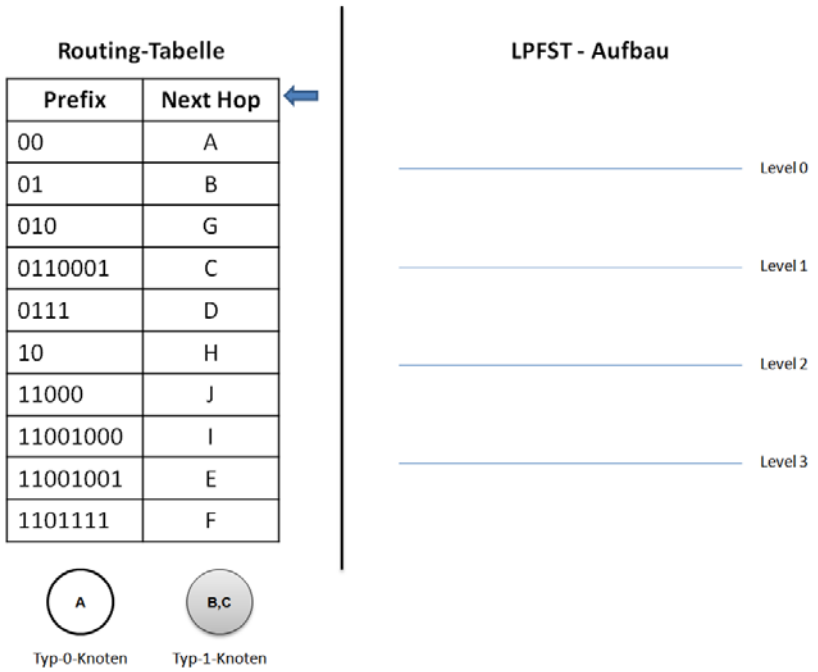


Abbildung 28, Beispiel LPFST – Baum [04]

Ein Grundsatz gilt dabei:

Ein Prefix wird dann in einem Knoten abgespeichert, wenn das Baum-Level niedriger oder gleich dem Prefix-length entspricht. [04]

Einzeln aufgeteilt ergibt sich folgende Konstruktion des Baumes:

Erster Eintrag in der Routing-Tabelle führt zum root-Element des Baumes.

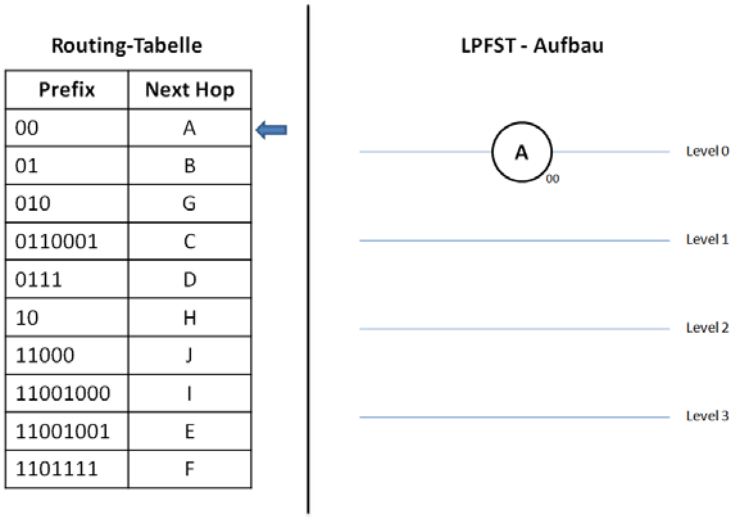


Abbildung 29, LPFST – Schritt 01 [04]

Das zweite Element hat keinen längeren Prefix als Eintrag 1, daher wird es als lchild eingerichtet.

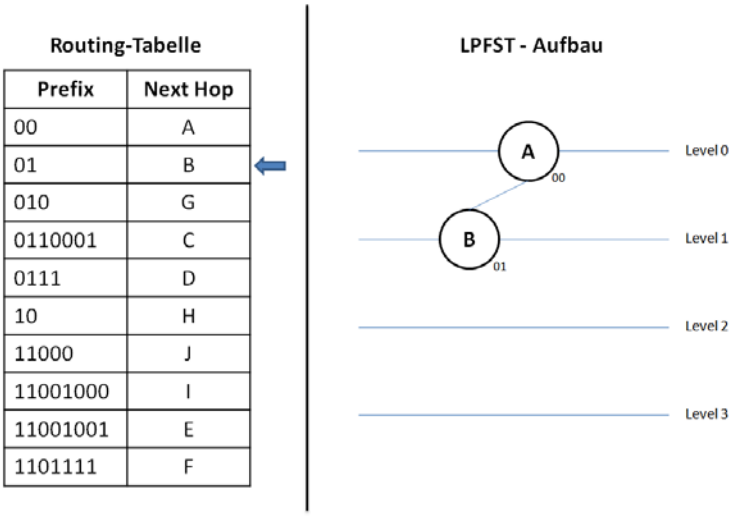


Abbildung 30, LPFST – Schritt 02 [04]

Beim dritten Tabelleneintrag ergibt sich ein längerer Prefix als der bisher vorhandene, daher wird dieser als neuer höchster Knoten eingerichtet und im Tausch damit der bisherige unterhalb der Baumstruktur gespeichert.

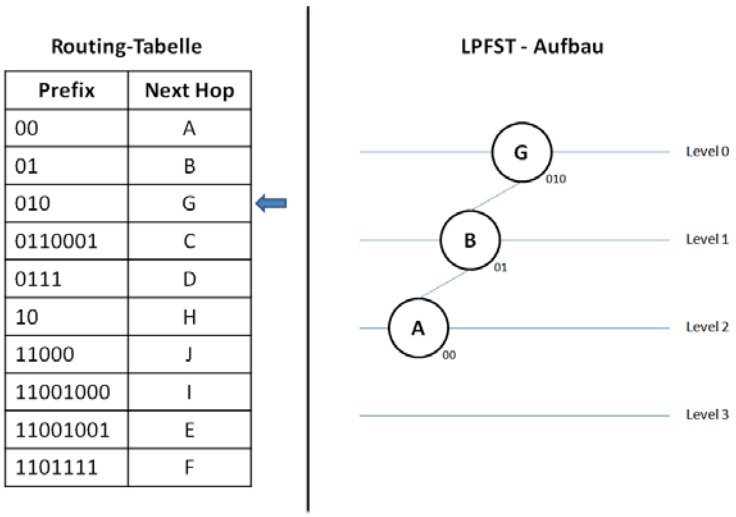


Abbildung 31, LPFST – Schritt 03 [04]

Der vierte Eintrag besitzt einen längeren Prefix, wird daher oben gespeichert. Der vorher oberste wird um ein Level nach unten gesetzt (Prefix größer als der vorherige), der Knoten mit “B“ als rchild gespeichert.

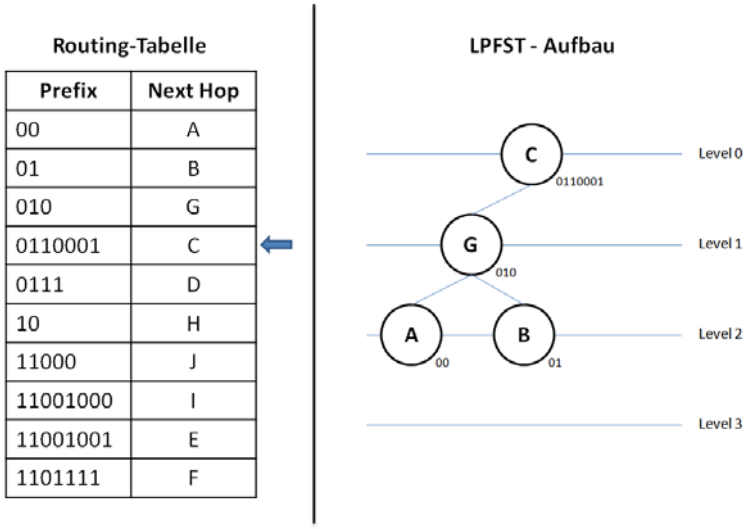


Abbildung 32, LPFST – Schritt 04 [04]

Der Eintrag mit “D“ wird an Level 1 gespeichert, der vorherige wird nun auf Level 2 zu einem Typ-1-Knoten und beinhaltet zusätzlich den dort vorher gespeicherten Prefix.

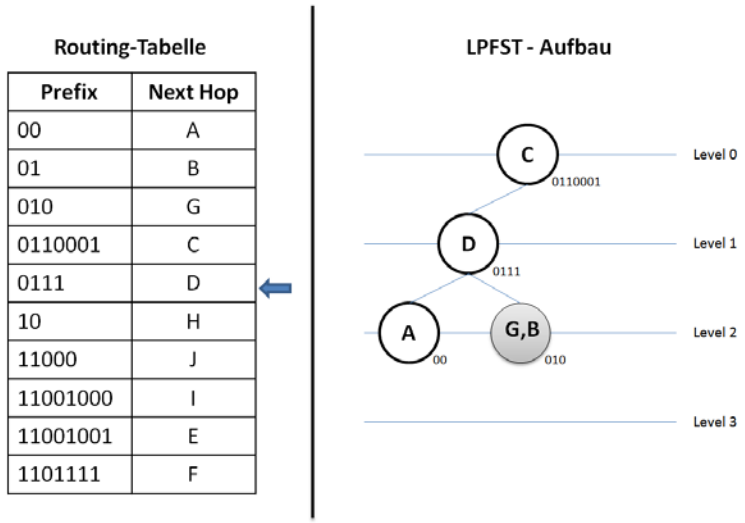


Abbildung 33, LPFST – Schritt 05 [04]

Vollständige Auflistung nach Abarbeitung aller vorhandenen Routing-Tabellen-Einträge.
Im vorliegenden Baum lässt sich gut erkennen, dass beim Verfahren nicht zwingend die Prefix-Länge verkürzt ist, wenn man alle Elemente einer Ebene vergleicht. (siehe “G“ zu “H“.

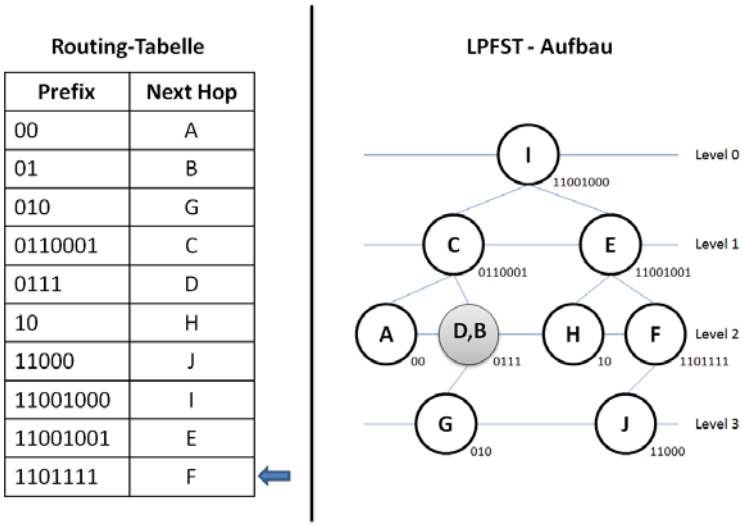


Abbildung 34, LPFST – Schritt 06 [04]

6.5 Algorithmus für IP Lookup in LPFST

Mit dem folgenden Algorithmus wird bei eingehenden Datenpaketen die Zieladresse ausgelesen und dann die Suche nach einem passenden Eintrag im Baum gesucht. Dabei startet die Suche immer in der Wurzel und vergleicht schrittweise die Einträge mit der Zieladresse.

```
Function Lookup(DA, y, level)
     {
       next-hop = default route;
       while y ≠ NIL do
         {
          if Get(DA, 0, Ly – 1) ≡ Py then return oy;
          if y is a type 1 node then next-hop  =
           contained-next hop of y;
          if Get(DA, level, level) ≡ 0 then y =
           lchild(y);
          else y = rchild(y);
            level++;
         }
          return next-hop;

}
```

Bei einem ankommenden Paket wird die Zieladresse (destination address, DA) extrahiert und der Algorithmus Lookup mit den Parametern (DA, root, 0) ausgeführt.

Bezogen auf das vorherige Beispiel ergibt sich folgender Ablauf, wenn ein Paket mit der Zieladresse “01100010“ verarbeitet wird:

Die Adresse wird mit der Baumwurzel verglichen, ohne Treffer. Der nächste Abgleich erfolgt mit dem linken Unterzweig (lchild) und stimmt überein (ersten 7 Stellen). Dann wird der Algorithmus unterbrochen und der next-hop des Pakets auf “C“ gesetzt.

Bei einem Paket mit der Zieladresse “01101111“ erfolgt wiederum der Abgleich mit den ersten Elementen im Baum ohne Treffer. Beim Vergleich mit dem Typ-1-Knoten erfolgt zuerst noch kein Treffer (0111*), aber beim zusätzlich enthaltenen Prefix für “B“ (01*). Da dieser Knoten nur einen linken Unterzweig hat und keinen rechten (rchild), wird der next-hop des Paketes auf “B“ gesetzt. [04]

6.6 LPFST Aktualisierung

Sobald sich Änderungen in der Netzwerk-Struktur ergeben, z.B. durch einen neuen Router und dadurch neuen Pfad, müssen die neuen Informationen auch im Baum hinterlegt werden bzw. korrigiert werden, falls ein bisheriger Eintrag nicht mehr gültig ist. Als Beispiel dafür ist die Funktion **Remove** folgend aufgeführt:

```
Function Remove(Px, y, level)
 {
   if Px ≡ Py then {
     case1: y is a leaf and a type-0-node
       {Delete node y; return;}
     case2: y is a leaf and a type-1-node
       { oy =  contained-next-hop of y;
         Py = Get(Py, 0, level – 1);
         Ly = level;
         change y to type-0-node and y = (0,Ly,Py,oy);
         return;
       }
     case3: y is an internal node
       { if y's left child has no longer prefix than
           its right child then {replace (Ly, Py, oy)
           with the length, prefix, next-hop of y's left
           child;
         Remove(Py, lchild(y), level + 1);}
         else {replace (Ly, Py, oy) with the length,
           prefix, next-hop of y's right child;
         Remove(Py, rchild(y), level + 1);}
         return;
       }
   if Lx ≡ level then {change y to type-0-node;
     return;}
```

```
   if Get(Px, level, level) ≡ 0
      then Remove(Px, lchild(y), level + 1);
      else Remove(Px, rchild(y), level + 1);
}
```

Beim folgenden Beispiel wird das Entfernen eines Knotens erläutert. Als Grundlage dazu dient der bisher schon verwendete Tree, bei dem der Eintrag “0111*“ gelöscht werden soll. Dieser Prefix befindet sich im Beispiel (Abbildung 34, Schritt 06) im Typ 1 – Knoten “D,B“.

Dies führt zu mehreren Bearbeitungsschritten, die sich auch direkt auf den folgenden Knoten “G“ auswirken:

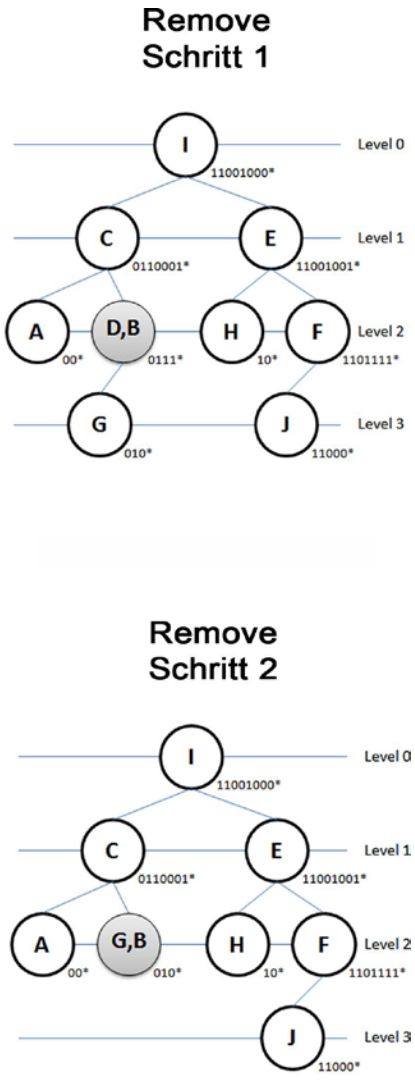


Abbildung 35, Beispiel Entfernung eines LPFST-Knoten [04]

Erläuterung:

Der Eintrag "0111*" soll entfernt werden. Der Wurzeleintrag "I" und Knoten "C" stimmen nicht überein. Beim Abgleich mit dem type-1-Knoten "D, B" wird ein Treffer geliefert.

Der Eintrag D mit der Adresse "0111*" wird gelöscht. Da es ein Knoten mit einem Nachfolger-Knoten ist, werden die Parameter prefix, prefix length und next-hop von dem Knoten "G" in den aktuellen kopiert.

Da der Knoten ein Typ 1 – Knoten war, ergibt sich im Schritt 2 wiederum ein Typ 1 – Knoten, der die Adresse "010" von "G" einträgt und in dem "01", die contained-next-hop-Adresse von "B", enthalten ist. [04]

6.7 LPFST im Vergleich mit anderen Strukturen

Im Vergleich zu anderen Baumstrukturen wie Trie, Patricia und einem Prefix Tree lassen sich insgesamt durch die Typ-1-Technik einige Knoten sparen, wie sich im folgenden Beispiel erkennen lässt:

Die Werte sind von einem Router im AS4637, vom 23.06.2004 mit 138286 Einträgen. Durch Anwendung der Algorithmen ergibt sich die Tabelle:

	Trie	PATRICIA	Prefix Tree	LPFST
Knoten-Anzahl	387722	258684	138286	132050
Leere Knoten	249436	120398	0	0
Durchschnittlicher lookup	23,36	20,89	21,04	18,90
Tiefe	32	24	24	24
Speicherbedarf	2271	2778	1350	1424

Abbildung 36, LPFST Knotenvergleich [04]

Durchschnittlich wird durch LPFST ein kompakterer Baum hergestellt, der weniger Knoten enthält und eine geringere Tiefe aufweist. Dadurch wird im Mittel eine geringere Anzahl von Suchschritten erreicht.

Auch der Speicherbedarf und die durchschnittliche Anzahl von lookup-Schritten weisen auf eine effiziente Speicherstruktur hin. Die Tiefe des Baumes ist nicht geringer als bei anderen Verfahren, aber durch das Fehlen von leeren Knoten ist der gesamte Aufbau des Baumes vergleichsweise kompakt.

6. Hash-basiertes Suchverfahren

In der wissenschaftlichen Arbeit von Waldvogel, Varghese und Turner [02] wird ein Algorithmus beschrieben, der basierend auf einer Hash-Tabelle eine binäre Suche ermöglicht, die eine Hierarchierung des längsten Prefix berücksichtigt.

Eine **Hash-Tabelle** besteht aus einem Index, der auf abgespeicherte Daten verweist. Dieser Index bzw. Schlüssel wird mit einer mathematischen Funktion berechnet und ermöglicht so, einen gesuchten Datensatz direkt zu finden, ohne aufwendige Schlüsselvergleiche durchführen zu müssen.

Dieser hat für die IPv4 – Adressen im Schnitt 5 hash lookups, vorausblickend auf IPv6 wären es 7 hash lookups. [02]
Als konkretes Beispiel wird das Hash-basierte Suchverfahren in diesem Kapitel genauer beschrieben.

6.1 Aufbau des Schemas zur Hash-Suche

Damit die Suche mittels Hash-Werten funktioniert, sind drei Punkte als grundlegende Voraussetzung zu betrachten: [02]

- Die Suche erfolgt mit einem prefix einer Adresse, nicht mit einem exakten Wert

- Die binäre Suche reduziert den Aufwand zu einem logarithmischen Ergebnis und erfolgt nicht linear

- Optimierungen sind sinnvoll, um die Rückwärtssuche einzuschränken

6.1.1 Lineare Suche in Hash-Tabellen

Zur Grundlage der binären Suche wird zuerst eine einfache lineare Suche mittels Hash-Tabellen eingeführt.

Dabei werden in einem Array alle möglichen Prefix-Längen mit einem Hash-Wert erfasst. Innerhalb eines einzelnen Array-Felds befinden sich Zeiger (Pointer) auf die jeweiligen Hash-Tabellen, in denen die verschiedenen Einträge mit der gleichen Prefix-Länge aufgeführt sind.

Beispielsweise existiert eine Routing-Tabelle mit 4 Einträgen, die in eine Hash-Suche umgesetzt werden soll. Die 4 Werte sind (01010, 0101011, 0110110, 011011010101).

Es existieren 3 verschiedene Prefix-Längen von 5, 7 und 12 Bit-Stellen. In einem Array mit 3 Feldern werden dann die verschiedenen Bit-Längen mit einem Hash-Wert abgespeichert.

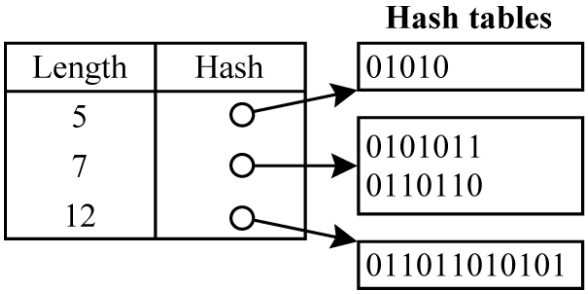


Abbildung 37, Hashtabelle Beispiel [02]

Im Feld, das den Hash-Wert für die Prefix-Länge 7 beinhaltet, zeigt ein Pointer auf eine weitere Hash-Tabelle, in der alle Einträge mit der Länge 7 enthalten sind. Dies sind in diesem Fall zwei Einträge.

Die Suche startet mit dem längsten Prefix-Eintrag der Länge 12. Von der vorgegebenen Adresse werden die ersten 12 Bits extrahiert und die Suche innerhalb der Hash-Tabelle mit allen Prefix-Längen 12 gestartet. Wird in dieser Tabelle kein passender Eintrag gefunden, erfolgt die Suche in der nächst kleineren Prefix-Längen-Tabelle, hier also mit der Länge 7. Dies erfolgt bis zur kleinstmöglichen Prefix-Länge. [02]

```
Function LinearSearch(D)

Initialize BMP to the empty string;
i:=Highest index array L;
While(BMP=nil) and (i>=0) do
  Extract the first L[i].length bits of D into D';
  BMP := Search(D', L[i].hash);
  i:= i - 1;
Endwhile
```

BMP = Best Matching Prefix

D = Adresse

L = Array mit verschiedenen Prefix-Längen

L[i].length = Länge des Prefix an Position i

L[i].hash = Pointer zur Hash-Tabelle mit Prefix-Länge L[i].length

6.1.2 Binäre Suche in Hash-Tabellen

Bei einer linearen Suche kann es im schlechtesten Fall bei N Einträgen zu N Suchvergleichen kommen. Um diesen Aufwand zu reduzieren, ist die binäre Suche geeignet.

Mittels Markierungen (Markern) in der Hash-Tabelle wird ein binärer Baum abgebildet, der den Aufwand auf **$O(\log_2 N)$** verringert. [02] Diese Marker verweisen auf Prefixe mit einer längeren Bit-Anzahl.

Als Beispiel dienen die drei Prefixe P1 = 0, P2 = 00, P3 = 111. In dem Array L verweist der erste Eintrag auf P1, der zweite auf P2, der dritte auf P3.

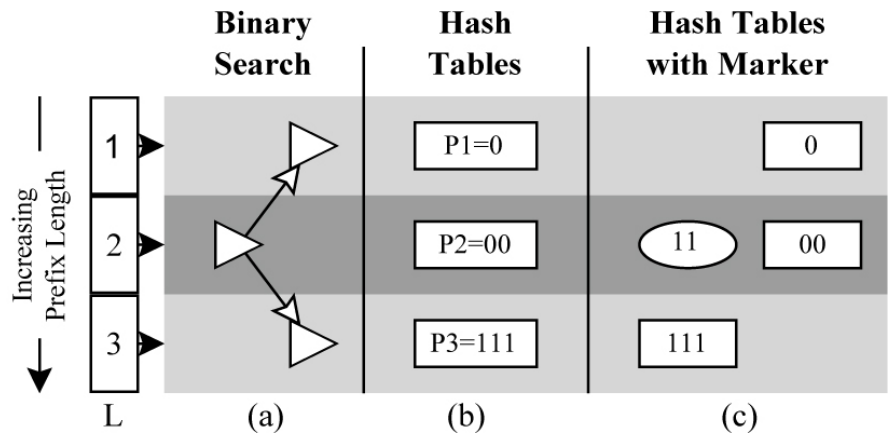


Abbildung 38, Hashtabelle Beispiel binär [02]

Angenommen, es wird nach der Adresse 111 gesucht. Der erste Suchvorgang findet in der mittleren Hash-Tabelle statt, P2 = 00. Diese Suche liefert keinen Treffer. Problematisch ist, dass es keinen Anhaltspunkt gibt, die Suche in der Hash-Tabelle fortzuführen, die einen längeren Prefix besitzt (hier P3 = 111).

Daher wird ein Marker mit 11 in der der Hash-Tabelle P2 gesetzt. Bei einer erneuten Suche nach 111 wird in der Tabelle der Marker gefunden, der auf die Tabellen mit längerem Prefix verweisen kann.

Verallgemeinert lassen sich die Hash-Tabellen mit den Markern zu einem Baum abbilden, der wie folgt aussehen kann:

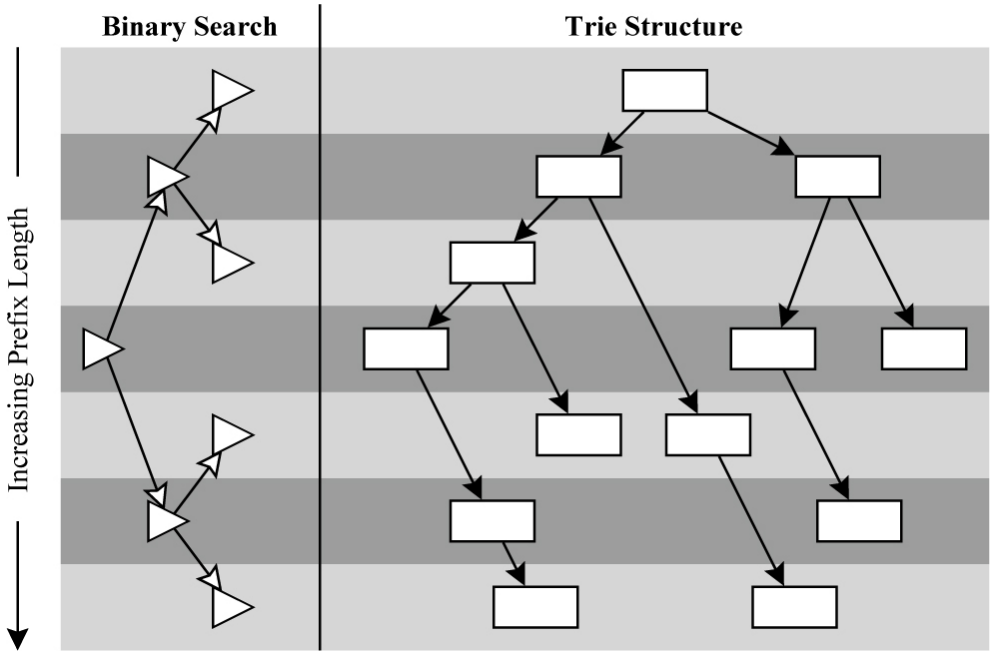


Abbildung 39, Hashtabelle mit Markern [02]

Horizontal sind die einzelnen Hash-Tabellen mit verschiedener Prefix-Länge. Die Prefix-Längen nehmen vertikal zu.

Im binären Schema ist die Wurzel linksseitig und entspricht der mittleren Prefix-Länge. Beim Baum-Schema ist die Wurzel die kürzeste Prefix-Länge.
Auf den Baum bezogen ist damit die obere Hälfte aller Prefix-Längen kleiner als die durchschnittliche Prefix-Länge. Somit ist die untere Hälfte der Prefixe länger das die mittlere Prefix-Länge.

```
Function NaiveBinarySearch(D)

Initialize search range R to cover the whole array
L;
While R is not a single entry do
  Let i correspond to the middle level in range R;
  Extract the first L[i].length bits of D into D';
```

```
  Search(D', L[i].hash);
  If found then set R := lower half of R
    Else set R := upper half of R;
  Endif
Endwhile
```

Mit diesem Algorithmus lässt sich eine simple binäre Suche in Hash-Tabellen abbilden, die auch Marker einsetzt. Die Suche erfolgt initial bei der mittleren Prefix-Länge, und wird beim Auffinden einer Markierung in der unteren Hälfte fortgesetzt, also Prefixe mit größerer Bit-Anzahl, ansonsten läuft die Suche in der oberen Hälfte weiter. [02]

6.1.3 Rückwärtssuche

Problematisch beim zuvor aufgeführten Algorithmus ist die Tatsache, dass ein Marker auch einen fehlerhaften Weg in der Suche verursachen kann, der keinen endgültigen Treffer liefert.

Dies würde dann eigentlich eine Rückwärtssuche notwendig machen, um auch die anderen Bereiche der Tabelle auf das gesuchte Prefix zu vergleichen. Dieses kann somit einen linearen Aufwand verursachen, wenn in diesen Algorithmus eine Rückwärtssuche implementiert wird.

Dies wird an folgendem Beispiel erläutert:

Die Prefixe in der Tabelle seien P1 = 1, P2 = 00, P3 = 111. Im vorherigen Beispiel ist der mittleren Hash-Tabelle der Prefix-Wert 00 und der Marker 11 hinterlegt, der auf P3 verweist.

Bei einer Suche nach 110 würde der erste Vergleich in der mittleren Tabelle erfolgen und aufgrund des Markers 11 einen Treffer liefern. Darauf würde der nächste Suchvorgang in P3 gestartet, aber keinen gültigen Treffer liefern. Auf alle hinterlegten Prefixe bezogen müsste aber der Prefix aus P1 als Ergebnis der Suchabfrage geliefert werden. Der Marker in P2 führt somit

in die falsche Richtung, die Suche hätte in der oberen Hälfte weiterlaufen müssen.

Bei einer erfolglosen Suche müsste anstatt des markierten Knotens die erneute Suche von der ursprünglichen Wurzel an erfolgen, welches aber den Aufwand vergrößern würde. Der Aufwand entspricht O(W). Anhand des folgenden worst-case-Beispiel wird dies deutlich:

Bei einer Adresse mit W Bits sei der Prefix P_i vorhanden mit der Länge i, und es gilt $1 \leq i < W$. Alle i Stellen sind mit 0 gesetzt. Zusätzlich gibt es ein Prefix Q mit W – 1 Stellen, die mit 0 gesetzt sind, außer der letzten (1).

Bei einer Suche nach der Adresse W würde jedes Prefix-Level durchlaufen werden, bis der Suchvorgang im längsten Prefix Q zum Treffer führt. [02]

6.1.4 Verhindern der Rückwärtssuche

Der Aufwand lässt sich dagegen reduzieren, indem eine Rückwärtssuche verhindert wird. Ein Ansatz dazu berücksichtigt nur noch Entscheidungssprünge in die untere Hälfte der Tabelle, wenn ein Marker gefunden wird. Das Ergebnis der weiterführenden Suche der oberen Hälfte hingegen wird schon vorberechnet und als Wert im Marker hinterlegt.

```
Function BinarySearch(D)

Initialize search range R to cover the whole array
L;
Initialize BMP found so far to null string;
While R is not empty do
  Let i correspondend to the middle level in Range
R;
  Extract the first L[i].length bits of D into D';
  M := Search(D', L[i].hash);
  If M is nil Then set R := upper half of R;
  Elseif M is a prefix and not a marker
```

```
    Then BMP := M.bmp; break;
  Else
    BMP := M.bmp;
    R := lower half of R;
  Endif
Endwhile
```

Erklärung:

Am Anfang wird die komplette Tabelle mit allen verschiedenen Prefix-Längen berücksichtigt. Davon wird die mittlere Prefix-Länge bestimmt und die Suche dort gestartet. Dazu wird von der vorliegenden Adresse die benötigte Bit-Anzahl extrahiert und in der entsprechenden Hash-Tabelle nach dem Eintrag gesucht.

Wird kein Treffer gefunden (kein Prefix oder Marker), wird die weitere Suche auf die obere Hälfte beschränkt, indem die Reichweite R auf die obere Tabelle eingeschränkt wird.

Wurde vorher ein Prefix gefunden, so wird dieses als Ergebnis geliefert. Wurde ein Marker gefunden, wird der im Marker hinterlegte Wert als vorläufiges Ergebnis abgespeichert und die Reichweite R auf die untere Tabellenhälfte beschränkt. [02]

6.2 Weitere Optimierungsmöglichkeiten

Ausgehend von einer Routingtabelle werden einige Optimierungen vorgestellt. In der folgenden Grafik sieht man die Verteilung von verschiedenen Prefix-Längen in einer Routingtabelle:

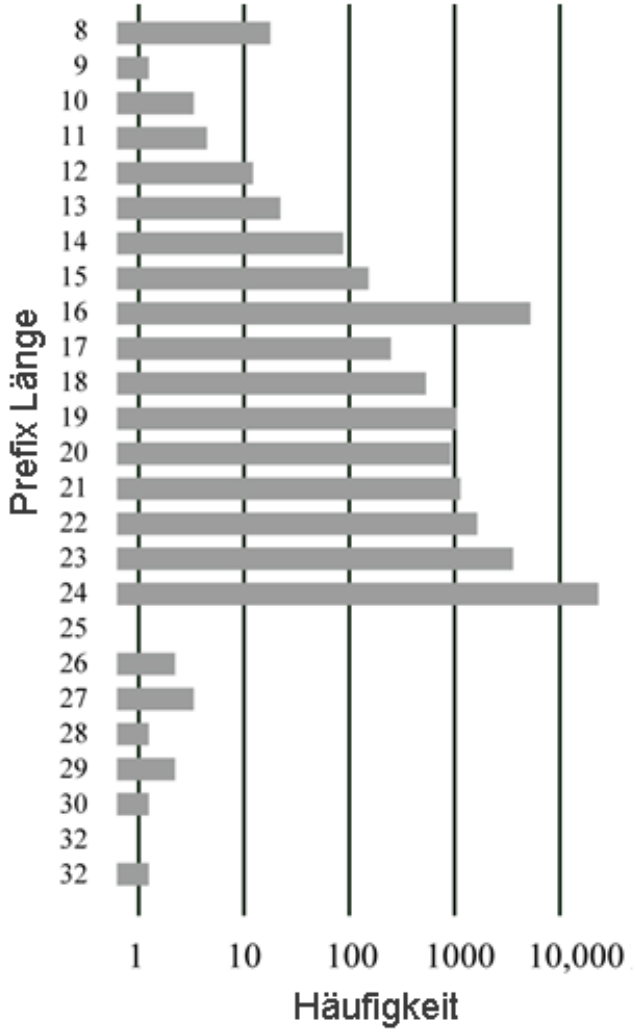


Abbildung 40, Beispiel prefix-Verteilung im Router [02]

Reduzierung der Prefix-Möglichkeiten

Eine einfache Optimierung besteht in der gezielten Auswahl von relevanten Prefix-Längen, da oft nicht alle Varianten existieren. Damit reduziert sich beispielsweise der Aufwand von $O(\log_2(32))=5$ zu $O(\log_2(23))=4{,}5$. [02]

Ändern der Baumstruktur

Ändert man die Struktur so, dass im Baum häufig gesuchte Prefix-Längen als erstes durchsucht werden, lässt sich die durchschnittliche Trefferzeit verringern. Dafür kann aber der Worst Case größer ausfallen, falls der Baum asymmetrisch aufgebaut ist.

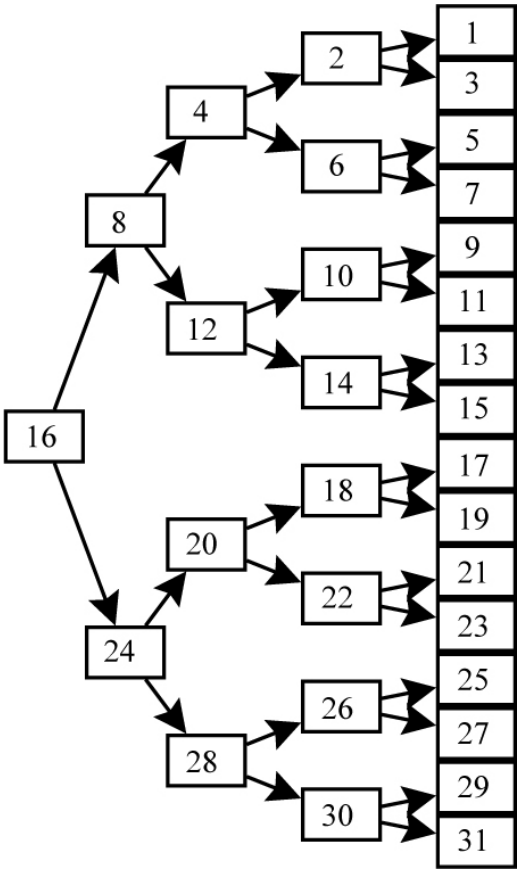


Abbildung 41, symmetrischer Prefix-Baum [02]

Die Neuordnung eines Baumes ist je nach Vorgaben ein kompliziertes Verfahren, und führt zu unterschiedlichen Baumstrukturen.

Bei den aufgeführten Abbildungen 42 bis 44 sieht man verschiedene Strukturarten, wie der Baum aufgebaut sein kann.

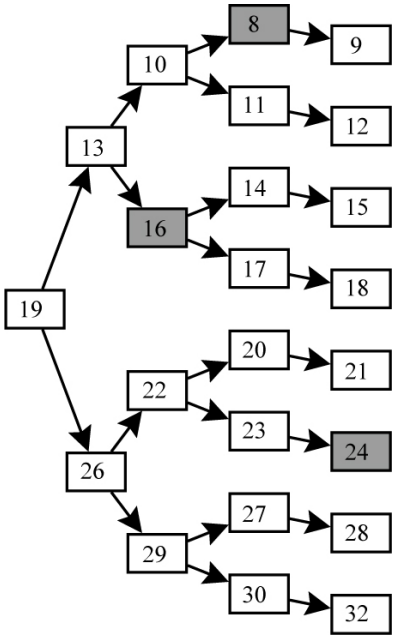


Abbildung 42, symmetrischer Prefix-Baum mit Balance [02]

Abbildung 42 ist symmetrisch aufgebaut, ausbalanciert und die einzelnen Knoten bzw. Blätter sind in einer Hierarchie angeordnet. Die 19 ist von den aufgelisteten Prefix-Längen die mittlere Zahl und bildet damit die Wurzel. Rechtsseitig sind alle Knoten kleiner, linksseitig somit größer als die Wurzel.

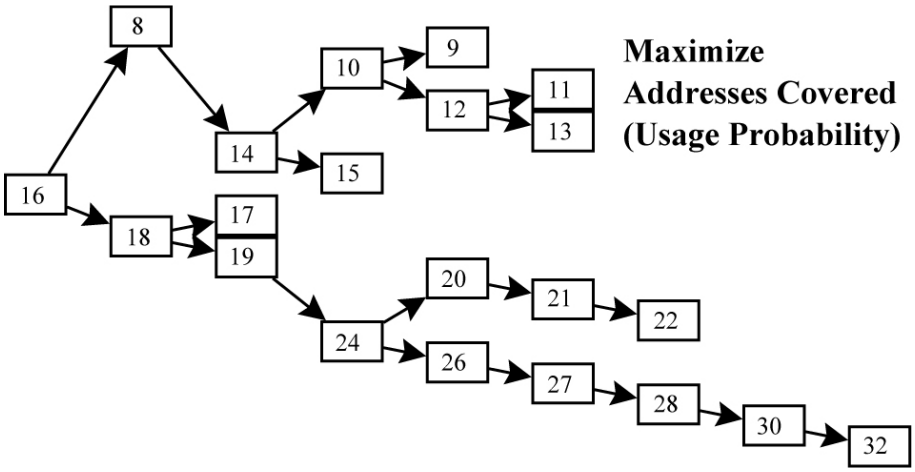


Abbildung 43, Prefix-Baum mit Häufigkeitsverteilung [02]

In der **Abbildung 43** ist der Baum so strukturiert, dass alle häufig abgefragten Adresslängen nahe der Wurzel sind, alle selten abgefragten Adressen umso weiter entfernt. Diese Struktur bietet eine gute durchschnittliche Abfragegeschwindigkeit, andererseits kann der Worst-Case einer selten abgefragten Adresse zu einer langwierigen Suche im Baum führen.

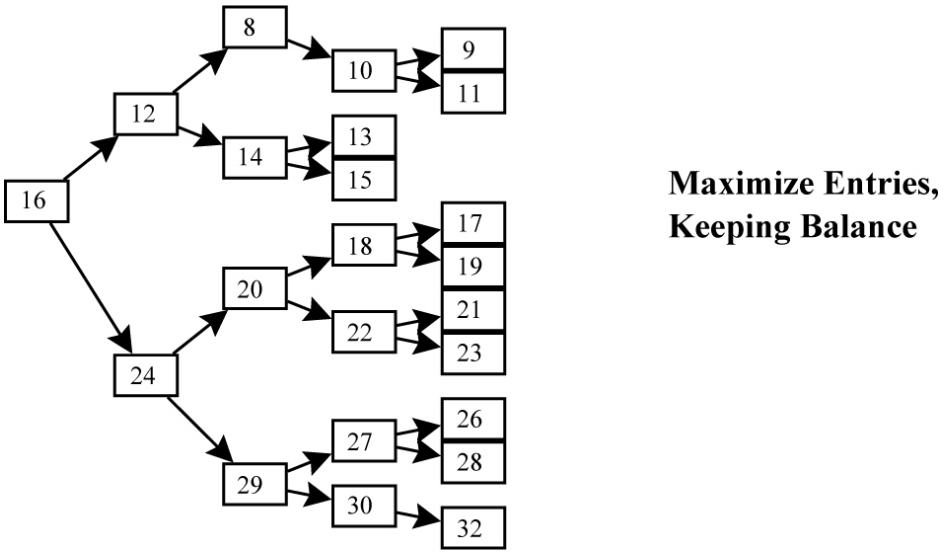


Abbildung 44, Prefix-Baum mit Balance und Häufigkeitsverteilung [02]

Als Kombination beider Strukturen ist die **Abbildung 44** aufgeführt: dort sind häufig angeforderte Adresslängen in wenigen Abfragen zu erreichen, der Baum ist aber ausbalanciert, fast alle Pfade haben die gleiche Länge.

7. Fazit

Bei den hier genannten Verfahren zum Aufbau einer Routing-Tabelle zeigt sich, dass keines generell als optimal zu bezeichnen wäre.

Vielmehr bietet jedes Verfahren gewisse Vor- und Nachteile, die sich letztendlich aber erst in einem realen Netzwerk fair vergleichen lassen. Als Grundlage für diese Studienarbeit dienten vor allem wissenschaftliche Arbeiten, die das jeweilige Verfahren als effizient beschreiben, aber keine vergleichbaren Messwerte der verschiedenen Algorithmen ausweisen.

So bietet zum Beispiel die Hash-Tabelle eine schnelle Möglichkeit zum auffinden von Adressbereichen, und auch eine Neustrukturierung durch IPv6 würde diesen Geschwindigkeitsvorteil beibehalten.

Zur Konstruktion sind aber einige Optimierungen und Anpassungen notwendig, was eine reale Umsetzung auf einen Router verkomplizieren könnte.

Schema	Worst case lookup	Update	Speicherbedarf
(lineare) Liste	O(N)	O(1)	O(N)
Binärer Baum	O(W)	O(W)	O(N)
Pfad-komprimierter Baum	O(W)	O(W)	O(N)
Radix	O(W)	O(W)	O(NW)
Multibit-Baum mit k-Bit-Block	O(W/k)	$O(W/k + 2^k)$	$O(2^k N W/K)$
Full expansion/compression	3	-	$O(2^k + N^2)$
Hashbasierte Lookups	O(log (2N))	O(N log N)	O(N)

Abbildung 45, Aufwandsvergleich Algorithmen [02], [17]

In der oben aufgeführten Tabelle in der **Abbildung 45** sind die verschiedenen Aufwandswerte gelistet. Dabei ist z.B. die hashbasierte binäre Suche dem Trie und Radix Trie überlegen.

Besonders bei der Realisierung mit der Hashtabelle müssen aber einige Optimierungen vorgenommen werden. Ohne diese würde oftmals eine Rückwärtssuche notwendig werden, mit der Marker-Setzung lässt sich dies aber minimieren.

Ein weiterer Vorteil der Hashtabelle ist die effektive Umsetzung auf IPv6. Dort ist der Aufwand durch die logarithmische Rechnung im Vergleich zu anderen Verfahren günstiger, die 128-bit Adressen benötigen im Schnitt zwei Suchschritte mehr als bei den 32-bit Adressen. [02]

Eine konkrete Untersuchung wäre aber notwendig, ob sich diese Ansätze auch in einem Netzwerk realisieren lassen. Besonders die Größe und Organisation eines Netzwerkes kann zu unterschiedlichen Ergebnissen führen.

In einem Netzwerk mit wenigen Einträgen in den Tabellen kann eine einfach zu erstellende lineare Suche eventuell auf Dauer effizienter sein als eine aufwendigere zu aktualisierende binäre Suche, die als Baumstruktur abgebildet wird.

Zu berücksichtigende Faktoren im Vergleich der Algorithmen sind z.B. [08]:

- Durchschnittlicher Zeitaufwand für Lookup
- Zeitaufwand für Aktualisierungen der Routing-Tabelle
- Speicherbedarf der Routingtabelle

Vor allem die Gewichtung des IP-Lookup wird vorrangig über die Effizienz eines Schemas entscheiden, da dort ein geringer Aufwand (gleichbedeutend mit schneller Suche) im realen Einsatz meist wichtiger sein wird als der Speicherbedarf. Ein Beispiel für konkrete Lookup-Zeiten (basierend auf einem Backbone-Router von 1999) ist in der folgenden Abbildung 46 aufgeführt:

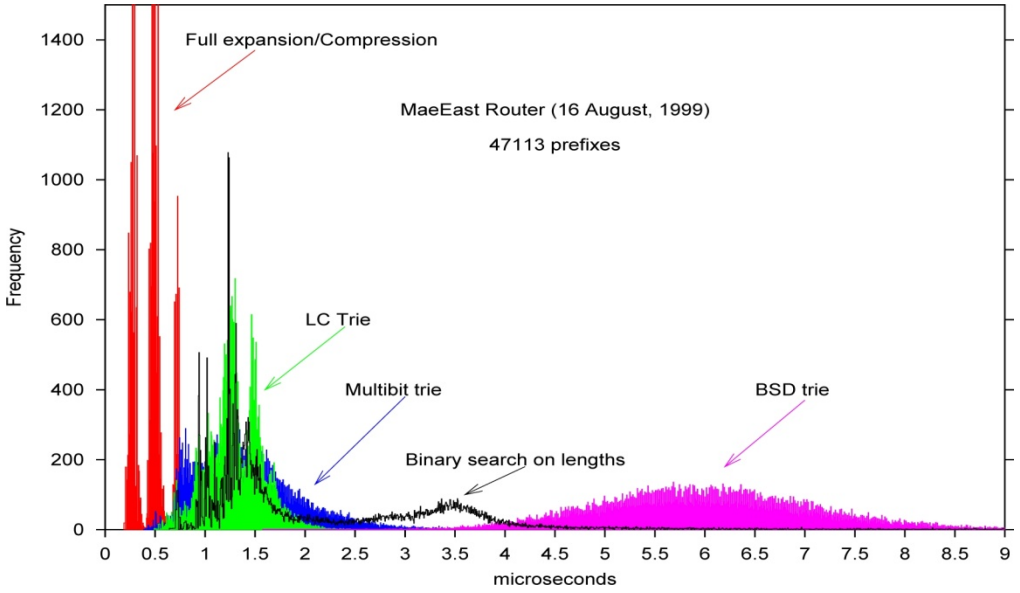


Abbildung 46, Lookup-Geschwindigkeit [17]

Dabei fällt vor allem die geringe Dauer des Lookups bei Full expansion / compression auf. Die anderen Schemata besitzen eine deutliche breitere Verteilung der Lookup-Zeiten. Trotz der vielversprechenden Werte ist eine Etablierung dieses Schemas bis heute eher nicht zu vermerken. Nachteilig ist der Speicherbedarf: übertrifft diese z.B. einen zur Verfügung stehenden Speicherplatz des L2-Caches eines Prozessors, müssen im schlechtesten Falle Teile der Struktur nachgeladen werden, welches die Zeiten dann deutlich verlangsamen würde.[17]

Allgemein kann man feststellen, dass durch die verschiedenen Optimierungen sowohl eine Baum-basierte Struktur als auch Hashtabellen in einem größeren Netzwerkbereich effizient einsetzbar sind.

Literaturverzeichnis

[01] IPv6: Internet-Protokoll kurz vor der Einführung. Autor: Hans Bär. http://www.pc-magazin.de/ratgeber/ip-adressen-242618.html. Veröffentlichung: 26.11.2009. Stand: 19.03.2012.

[02] Waldvogel / Varghese / Turner u.a. Scalable High Speed IP Routing Lookups. IEEE ACM SIGCOMM ’97 conference on Applications, Technologies, Architecture and Protocols for Computer Communication. 1997, S. 25-36.

[03] Tanenbaum, Andrew S. Computernetzwerke. Pearson Studium Verlag.
3. Aufl. 2003.

[04] Wuu / Chen / Liu. A Longest Prefix First Search Tree for IP Lookup. IEEE ACM 0-7803-8938-7/05. 2005, S. 989-993.

[05] Yamamoto / Kato / Watanabe. Radish – A Simple Routing Table Structure for CIDR. Technical memo of WIDE project. 1995.

[06] Güting, Ralf Hartmut / Dieker, Stefan. Datenstrukturen und Algorithmen.
Viewig+Teubner Verlag. 2. Aufl. 2003

[07] Degermark / Brodnik / Carlsson u.a. Small Forwarding Tables for Fast Routing Lookups. IEEE ACM 0-89791-905-X/97/0009. 1997.

[08] Sangireddy / Futamura / Aluru u.a. Scalable, Memory Efficient, High-Speed IP Lookup Algorithms. IEEE/ACM Transactions on Networking, Vol. 13, No.4. 2005, S. 802-812.

[09] Akhbarizadeh / Nourani. Hardware-Based IP Routing Using Partitioned Lookup Table. IEEE/ACM Transactions on Networking, Vol. 13, No.4. 2005, S. 769-781.

[10] Peterson, Larry L. / Davie, Bruce S. Computernetze. dpunkt-Verlag. 4. Aufl. 2008.

[11] Saake, Gunter. / Sattler, Kai-Uwe. Algorithmen und Datenstrukturen. dpunkt Verlag. 4. Aufl. 2010.

[12] Morrison, Donald R. PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. Journal of the Association for Computing Machinery. Vol. 15, No. 4. 1968, S. 514-534.

[13] Malkin, G. RIP Version 2. Technical Report RFC-2453. 1998.

[14] Moy, J. The open shortest path first (OSPF) specification. Technical Report RFC-1131. 1989.

[15] Rekhter, Y. / Li, T. / Hares, S. A Border Gateway Protocol 4 (BGP-4). Technical Report RFC-4271. 2006.

[16] Crescenzi, Pierluigi / Dardini, Leandro / Grossi, Roberto. IP Address Lookup Made Fast and Simple. Technical Report : TR-99-01. Universitá di Pisa, Dipartimento di Informatica. 1999.

[17] Ruiz-Sánchez, Miguel Á. / Biersack, Ernst W. / Dabbous, Walid. Survey and Taxonomy of IP Address Lookup Algorithms. 2001.