

Technische Universität München

Fakultät für Informatik

**Efficient IP Prefix Lookup Algorithms and Datastructures:
A Framework for Performance Evaluation**

Bachelor-Arbeit

Lehrstuhl für Netzwerkarchitektur
Prof. Anja Feldmann

von

Dennis Knorr

1. Prüfer: Prof. Anja Feldmann
2. Prüfer:
Betreuer: Fabian Schneider, Jörg Wallerich
Abgabedatum: 14.09.2006

Eidesstattliche Erklärung

Hiermit versichere ich eidesstattlich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, wurden als solche kenntlich gemacht.

Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

München, 14. September 2006

Unterschrift

Abstract

Heutzutage wird der Vorgang des IP-Präfix-Lookups zu einem ernstem Problem beim Forwarding der Pakete im Internet. Diese Problematik verschärft sich noch durch den zunehmenden Einsatz von IPv6. Deshalb wurde ein Framework entwickelt, um die Leistung und Effizienz von verschiedenen IP-Präfix-Lookup-Algorithmen und Datenstrukturen – für Longest-Prefix-Matching – zu vergleichen. Dieses Framework kann auch als C-Bibliothek zum Speichern und Abrufen von Information, die IP-Präfixen zugeordnet sind, benutzt werden. Drei Ansätze wurden verglichen: Brute Force (lineare Suche), Patricia Trees/Tries und der Elevator-Stairs-Algorithmus.

Inhaltsverzeichnis

1. Einleitung.....	8
1.1 Aufgabenstellung.....	8
1.2 Gliederung.....	9
2. Einführung in die Thematik.....	10
2.1 Einführendes Beispiel.....	10
2.2 Forwarding.....	10
2.3 Routing.....	11
2.4 Adressierung von Netzwerkbereichen.....	12
2.4.1 Klassenbasiertes Adressierungsschema.....	12
2.4.2 Classless Interdomain Routing.....	12
2.5 IPv4 und IPv6.....	14
3. Zielsetzung.....	15
3.1 Quelldaten.....	15
3.2 Relevante Datenstrukturen und Methoden.....	16
3.3 Mögliche Algorithmen.....	16
3.4 Aktuell eingesetzte Algorithmen und Datenstrukturen.....	17
4. Verwendete Algorithmen.....	18
4.1 Der Brute-force-Ansatz.....	18
4.2 Patricia Trees.....	18
4.3 Der Elevator-Stairs-Algorithmus.....	22
4.4 Theoretische Betrachtung.....	24
5. Das Framework.....	25
5.1 Konfiguration.....	25
5.2 Bitvektoroperationen.....	27
5.3 Präfixoperationen.....	30
6. Evaluation.....	32
6.1 Beschreibung des Experiments.....	32
6.1.1 Beschreibung der Zeitmessung	32
6.1.2 Messungen des Speicherverbrauchs der Datenstrukturen.....	33
6.2 Ergebnisse.....	33
6.2.1 Präfix-Operationen pro Sekunde.....	33
6.2.2 Anzahl der aufgerufenen IP-Funktionen.....	34

6.2.3 Speicherverbrauch der Datenstrukturen.....	34
6.3 Fazit der Evaluation.....	35
7. Zusammenfassung und Ausblick.....	36

Tabellen:

Tabelle 1: Beispiel einer Forwarding-Tabelle.....	11
Tabelle 2: Beispiel eines LPM-Problems in einer Forwarding-Tabelle.....	13
Tabelle 3: Forwarding-Tabelle für den Patricia Tree Aufbau.....	19
Tabelle 4: Komplexitätstheoretische Grenzen der Präfixoperationen.....	24
Tabelle 5: Operationen pro Sekunde für die einzelnen Prozeduren.....	33
Tabelle 6: gemittelte Anzahl der Aufrufe der wichtigen IP Funktionen.....	34
Tabelle 7: Speicherverbrauch der Datenstrukturen (in Kilobytes).....	34

Abbildungen:

Abbildung 1: Beispiel eines Patricia Trees.....	19
Abbildung 2: Kollaps einer Kante in einem Patricia Tree.....	19
Abbildung 3: Die Elevator-Stairs-Datenstruktur.....	22

Glossar

AVL-Baum	Datenstruktur aus der Algorithmik[7]
Bitvektor-API	Programmierschnittstelle, um mit generischen Bitvektoren umgehen zu können
DFN	Deutsches Forschungsnetz, Hochleistungsnetz für Wissenschaft und Forschung
Forwarding	Weiterleiten eines IP-Pakets aufgrund der Forwarding-Tabelle von einem Netzwerkknoten zum nächsten
IP	Internet Protocol, Vermittlungsprotokoll, Adressenname
IPv4	4. Version des IP Protokolls
IPv6	6. Version des IP Protokolls
LRZ	Leibniz-Rechen-Zentrum
N	Anzahl von Datensätzen in einer Datenstruktur
Netzwerksegment	Ein Netzwerkbereich, der über sein Präfix identifiziert wird
NextHop	Der Router auf einem Pfad durchs Netzwerk
OSI-Modell	Open Systems Interconnection, Abstrahiertes Schichtenmodell für Kommunikationsnetze
Port	Schnittstelle einer Netzwerkkomponente zum Verbinden von Endgeräten
Präfix	Vorderer Teil eines Bitvektors
Routing	Mechanismus der die Wegewahl im Internet bestimmt
TCP	Transport Control Protocol, verlässliches Übertragungsprotokoll im Internet
W	W beschreibt die Bitlänge einer IP-Adresse
Worst-Case	Schlimmstmögliche Zeit- oder Speicherkomplexität, abhängig von der Eingabegröße

1. Einleitung

Das Internet wird immer mehr zum Bestandteil des alltäglichen Lebens der Menschen. Ob von unterwegs oder von zu Hause, immer mehr Nutzer und Anbieter verwenden es und nehmen es oftmals nicht wahr. Durch dieses Wachstum wird das Netz, welches einerseits von Anfang an auf Effizienz ausgelegt, aber andererseits nie für diese generelle Verwendung gedacht war, bis an die Grenzen seiner Leistungsfähigkeit getrieben. Auch Anwendungen, wie zum Beispiel solche aus Medizin und Militär, setzen immer mehr auf das Internet. Um dieser ständig steigenden Belastung gewachsen zu sein, müssen neuere Technologien für die Kernkomponenten des Internets entwickelt werden.

Diese Kernkomponenten haben die primäre Aufgabe, Pakete möglichst schnell von einem Host zu einem anderem Host zu schicken, auch wenn dieser ein völlig anderes Gerät ist, und am anderen Ende der Welt liegt. Die Mechanismen, die Pakete durch das Internet zu schicken, nennen sich Routing und Forwarding. Auf welche Weise diese Delegation durch den Netzwerkkern stattfindet, hängt von mehreren Einzelheiten ab; unter anderem auch davon, wie schnell im Router die Zieladresse oder das Netzwerksegment für ein bestimmtes Paket nachgeschlagen werden kann. Dieser Vorgang, im weiteren "Adresslookup" genannt, entwickelte sich durch das explosive Wachstum des Netzes und die Verbesserung aller anderen Komponenten der Router beim Forwarding der Pakete im Netz zum Flaschenhals. Aus diesem Grund wurden Verfahren entwickelt, die dieser Belastung durch bessere Skalierbarkeit gewachsen sein sollen.

1.1 Aufgabenstellung

Um die Leistungsfähigkeit verschiedener Verfahren vergleichen zu können, ist im Rahmen dieser Bachelor Thesis ein Framework konzipiert worden, welches das Testen von solchen Adresslookup-Algorithmen zum Nachschlagen von Routing-Informationen abstrahiert und erleichtert.

Hierfür wird auch eine generalisierte Bitvektor-API bereitgestellt, welche die Nutzung von IPv4- oder IPv6-Adressen innerhalb der im Framework getesteten Algorithmen vereinheitlicht.

Es werden drei Algorithmen erläutert, ihre Implementierung vorgestellt und deren Effizienz untersucht. Diese sind der Brute-force-Ansatz, die Patricia Trees[4], sowie der Elevator-Stairs-Algorithmus[4].

1.2 Gliederung

Die Arbeit ist folgendermaßen aufgebaut: In Kapitel 2 erfolgt eine kurze Einführung in die Thematik. Anschließend werden in Kapitel 3 die Anforderungen an das zu entwickelnde Framework und die dabei auftretenden Probleme dargestellt. Die für die Evaluation benutzten Algorithmen werden in Kapitel 4 vorgestellt. Die Implementation wird in Kapitel 5 näher beleuchtet und im sechsten Kapitel werden schließlich die Experimente beschrieben. Dort werden auch die Ergebnisse der Leistungsmessungen aufgeführt. Im siebten und letzten Kapitel wird schließlich eine Zusammenfassung und ein Ausblick gegeben.

2. Einführung in die Thematik

Bevor die Algorithmen näher erläutert werden, müssen die Grundlagen für Routing und Forwarding dargelegt werden, und das Adressierungskonzept im Internet erklärt werden, um deutlich zu machen, wofür die Algorithmen verwendet werden.

2.1 Einführendes Beispiel

Wenn man beispielsweise von einem Computer in einem Privathaushalt eine Verbindung zu einem Webserver aufbaut, wird leicht übersehen, wie viele Komponenten hierbei zusammenspielen. Dabei müssen zwei Dinge bekannt sein: Die Zieladresse, welche den Rechner, der den Dienst bereitstellt identifiziert, sowie der bereitgestellte Dienst. Die Verbindungsschicht zwischen dem Sender und Empfänger der Anfrage ist hierbei IP. Wenn der Privatrechner an den Webserver eine Nachricht schicken will, so verpackt er diese in TCP-Pakete, die wiederum in IP-Pakete verpackt werden. Hierbei ist für die Anwender irrelevant, ob das in dem IPv4- oder IPv6-Format geschieht. Dies ist aufgrund des OSI-Modells für den Benutzer des Rechners nicht sichtbar.

Die Nachricht wird dann in mehreren Teilen vom Sender zum Empfänger über mehrere Router geschickt. Ein Router ist ein zentraler Knoten- und Verkehrspunkt des Netzes. Diese Router haben mitunter mit mehreren Schwierigkeiten zu kämpfen. Zum einen müssen sie oft starke Netzlast bewältigen. Das bedeutet, dass in kurzer Zeit sehr viele Pakete ankommen, die sehr schnell über die beste Route weitergeleitet werden sollen. Wenn dies nicht schnell genug geschieht, kann es sein, dass der Paketpuffer des Routers überläuft und Pakete verloren gehen. Falls zu viele Pakete verloren gehen, bricht die Verbindung vom Rechner zum Webserver ab.

Weiterhin haben Pakete verschiedenartige Priorisierungen, wodurch sie unterschiedlich schnell weitergeleitet werden. Durch das exponentielle Anwachsen des Netzes wird dieses Problem noch erheblich verschärft.

2.2 Forwarding

Der Begriff "Forwarding" bezeichnet das Weiterleiten eines Pakets von einem Router zu einem anderen Router anhand der Forwarding-Tabelle. Diese ist eine Tabelle aus Netzwerkbereichen und zugehörigen NextHops, aus der der Router herauslesen kann, über welchen Port und an wen er ein Paket weiterleiten muss, wenn er es zu einem bestimmten Netzwerk schicken will. In Tabelle 1 ist ein Beispiel für eine Forwarding-Tabelle gegeben.

Netzwerksegment	Port	NextHop
Default	A	192.168.1.1
10.128/10	B	192.168.2.2
10.192/10	C	192.168.3.3
10.129/16	D	192.168.4.4
192.168.1/24	A	/
192.168.2/24	B	/
192.168.3/24	C	/
192.168.4/24	D	/

Tabelle 1: Beispiel einer Forwarding-Tabelle

Wie diese Forwarding-Tabellen intern verwaltet werden, ist für den Administrator nicht ersichtlich. Das Erstellen solcher Forwarding-Tabellen bezeichnet man als Routing.

2.3 Routing

Der Routing-Prozess oder ein spezifisches Routingprotokoll beschreibt zunächst, wie die Forwarding-Tabelle für den einzelnen Router erstellt wird. Zur Definition eines Routingprotokolls gehört zwar auch die Kommunikation zwischen den Routern zum Austausch von Routing- beziehungsweise Forwarding-Informationen, vor allem bestimmt das Protokoll aber den Mechanismus zur Erzeugung einer Forwarding-Tabelle. Hier unterscheidet man Link-State-, Distance-Vector- sowie Pathvector-Algorithmen.

Bei Link-State-Algorithmen werden Informationen über alle Links, die die an den Routern angeschlossen sind, über das Netz an alle anderen Router geflutet. Dabei erhält jeder Router Informationen über die komplette Topologie des Netzes, wodurch er eine optimale Forwarding-Tabelle errechnen kann.

Im Gegensatz dazu schickt ein Router bei einem Distance-Vector-Algorithmus nur seinen direkten Nachbarn die erlernten Erreichbarkeitsinformationen samt Entfernung zum Ziel. Diese Informationen schickt er all seinen Nachbarn mit Ausnahme desjenigen, von dem er sie erhalten hat.

Ein Pathvector-Algorithmus ist eine spezielle Version eines Distance-Vector-Algorithmus und wird vom BGP, dem „Border-Gateway-Protocol“, verwendet.

Jeder Router berechnet die beste Route zu allen Netzwerksegmenten. Wenn nun ein Paket ankommt, wird es über den Port an den NextHop geschickt, der für das Netzwerksegment

beziehungsweise Präfix eingetragen ist, zu dem die Ziel-IP des Pakets am besten passt. Dies geschieht nach dem Prinzip des Longest-Prefix-Matching.

2.4 Adressierung von Netzwerkbereichen

Als man die Überlegungen zum Wegwahlverfahren im Internet machte, wusste man nicht, wie stark sich das Internet entwickeln würde. Aus diesem Grund gibt es zwei Generationen von Adressierungsschemata, wobei das zweite entwickelt wurde, als man merkte, dass das erste auf Dauer keine gute Aufteilung des Netzes ermöglichte.

2.4.1 Klassenbasiertes Adressierungsschema

Bei IPv4 sind die Adressen 32 Bit lang, wobei die Adresse in 4 Einheiten von je einem Byte aufgeteilt wird. Ein Beispiel für eine IP-Adresse ist "131.159.74.65".

IP-Adressen, im Folgenden „IP“ genannt, bestehen aus einem Netzwerkanteil und einem Hostanteil. Man unterteilt hier den Adressraum in 4 Subnetze unterschiedlicher Größe, die A, B, C und D genannt werden. Ein Subnetz aus der Adressklasse A hat 16 Millionen Hosts, eins aus der Klasse B hat 64536 Hosts, und eins aus Klasse C hat 256 Hosts. IPs aus der Klasse D sind Multicast-Adressen. Der Vorteil dieser klassenbasierten Adressierung war die feste Länge des Netzwerkanteils der Adresse, und dass die Entscheidung für die betreffende Netzklasse aufgrund der ersten drei Bits getroffen werden konnte. Dann konnten die Router ein "Exact Matching Scheme" mit 3 Tabellen verwenden, um den Ausgangs-Port für das Paket zu ermitteln. Dadurch konnte der Lookup sehr schnell erfolgen. Diese Art der Adressierung hatte allerdings auch zwei große Nachteile. Zum einen kostete das Verfahren wesentlich mehr Speicher, da immer für alle möglichen Netzbereiche Platz vorhanden sein musste. Zum anderen war diese Lösung bei der Adressvergabe nicht flexibel genug. Deswegen stieg man schließlich auf CIDR um.

2.4.2 Classless Interdomain Routing

Im Gegensatz zur klassenbasierten Adressierung ist die Länge des Netzwerkanteils bei dem CIDR-Verfahren variabel. Dadurch setzt sich der Netzwerkschlüssel aus zwei Komponenten zusammen, dem Netzwerkpräfix und der Länge dieses Bitstrings. Ein Beispiel für ein privates Netzwerk ist das Präfix 192.168.1/24.

Wenn der Router entscheiden muss, ob ein Paket in solch ein Netzwerk geschickt werden soll, verknüpft er die IP und das Präfix mit dem binären UND-Operator. Wenn dabei in der

Binärrepräsentation wieder das Präfix herauskommt, hat der Router das richtige Netzwerksegment und das dazugehörige Ausgangsinterface gefunden. Durch die variable Bitlänge kann allerdings kein Exact-Matching-Scheme mehr angewendet werden. Hierbei kommt das aus der Algorithmik bekannte Longest-Prefix-Matching, kurz LPM, zur Anwendung.

Präfix	Präfixlänge	Port	NextHop
*	0	A	192.168.1.1
0000101010	10	B	192.168.2.2
0000101011*	10	C	192.168.3.3
0000101010000001	16	D	192.168.4.4
110000001010100000000001	24	A	/
1100000010101000000000010	24	B	/
1100000010101000000000011	24	C	/
1100000010101000000000100	24	D	/

Tabelle 2: Beispiel eines LPM-Problems in einer Forwarding-Tabelle

Wenn die Zieladresse eines Pakets 00001010100000010000000100000001 ist, stellt sich die Frage, zu welchem Ausgang das Paket herausgeschickt werden soll, wenn man obige Forwarding-Tabelle zur Orientierung heranzieht. Wie leicht zu erkennen ist, passt die IP auf das zweite und vierte Präfix, da die ersten zehn Bits von links gleich den ersten zehn Bits der IP sind. Allerdings stimmen beim dritten Präfix die nächsten sechs vorhandenen Bits auch mit denen in der Zieladresse überein.

Folglich kommen hier zwei Möglichkeiten in Betracht: Entweder steht das physikalische Netzwerk, welches durch das dritte Präfix adressiert wird, außerhalb des Netzwerkes des ersten Präfix, oder IPs aus dem Subnetzwerk sollen aus irgendeinem Grund schneller erreicht werden als IPs aus dem übergeordnetem Netzwerksegment.

Folglich scheint es egal zu sein, ob das Paket bei A oder C abgeschickt wird. Allerdings kann man annehmen, dass das Paket schneller beim Ziel ankommt, wenn man es bei C abschickt. Schließlich wurde dieses Präfix ja wahrscheinlich eingefügt, weil das Ziel netzwerktopologisch näher am Ausgang C als an Ausgang A liegt, da C ein Teil des Netzwerkes ist, welches an A angeschlossen worden ist.

2.5 IPv4 und IPv6

1991 begann das IETF sich mit der Weiterentwicklung des IP-Protokolls zu beschäftigen. Im Laufe der Zeit wurde dann IPng entwickelt, woraus sich dann schließlich IPv6 entwickelte. IPv6 enthielt viele Neuerungen, da die Netzentwickler mehrere Probleme auf einmal mit dem Protokoll lösen wollten. Unter anderem wurde auf folgende Merkmale Wert gelegt:

- Unterstützung von Sicherheitsmerkmalen
- Automatische Konfiguration
- Erweiterte Routing-Funktionalität

Die wichtigste Neuerung war allerdings die massive Vergrößerung des Adressraums von 32 Bit auf 128 Bit. IPv6 besitzt auch keine Klassen mehr. Allerdings sind wie in IPv4 einige Präfixe reserviert. Durch die Erhöhung der Adresslänge wurden dann auch neue IP-Lookup-Algorithmen benötigt, da die Einführung von CIDR zwar die Forwarding-Tabellen bezüglich der Anzahl der Subnetze verkleinerte, jedoch die Vergrößerung der Präfixe die Suchzeit in der Forwarding-Tabelle erheblich verlängerte.

3. Zielsetzung

Wenn man neue IP-Lookup-Algorithmen entwickeln will, gibt es verschiedene Ansätze, um das LPM-Problem zu bewältigen. In [1] werden mehrere Ansätze wie Wörterbücher - auch Hashtables genannt - und Listen, Bäume sowie deren Kombinationen miteinander verglichen und deren Effizienz überprüft.

Bei der Konstruktion dieser Algorithmen muss auch bis zu einem gewissen Grad auf Skalierbarkeit bezüglich der IP-Länge geachtet werden, da durch die Umstellung von IPv4 auf IPv6 die Anzahl der Adressen und Präfixe um ein vielfaches wachsen. Andererseits ist nicht zu erwarten, dass die Adresslänge in absehbarer Zeit noch einmal vergrößert wird, da so viele IPv6-Adressen existieren, um jeden Quadratzentimeter der Erde mit mehreren IP-Adressen abzudecken.

Folglich gibt es bei der Konstruktion von Algorithmen komplexitätstheoretisch zwei Größen zu beachten: Das sind die maximale Länge der Adresse, welche im Folgenden mit W bezeichnet wird, und die Anzahl der zu verwaltenden Präfixe, N genannt.

3.1 Quelldaten

Um die implementierten Algorithmen gut testen zu können, braucht es zwei verschiedene Arten von Daten: Zum einen die Präfixdaten und zum anderen die zu suchenden IPs. Die Präfixdaten können von www.ripe.net oder www.routeviews.org bezogen werden. Dort sind Archive der Präfixe gespeichert, die durch Mitschneiden der BGP-Kommunikation zwischen den Routern ermittelt wurden. In einem aktuellen Archiv sind knapp 180000 verschiedene Präfixe gesammelt. Die Präfixdaten wurden mittels `bgpdump`[2] extrahiert. Diese Ausgabe war so umfangreich, dass die relevanten Daten zusätzlich aus der restlichen Ausgabe herausgefiltert werden mussten. Nach der Trennung dieser Ausgaben wurden in den BGP-Daten auftretende Präfix-Duplikate entfernt. Die sich daraus ergebende Präfixdatendatei enthält in der ersten Zeile die Anzahl der Präfixe, die in der Datei vorhanden sind. Eine Zeile mit einem solchen Präfixdatum hat dann das Format „*Eintrag: <Präfix>/<Präfixlänge> <Routeninformation>*“

Die IPs, welche die Suchdaten darstellen, wurden mittels `tcpdump` an dem Uplink des LRZs zum DFN ermittelt. `Ipsumdump` schreibt die IPs aus den gewonnenen Daten in eine Datei. Durch die Loslösung der IPs aus den erfassten Paketen wurden diese Daten anonymisiert. Das Format der erzeugten IP-Daten-Datei ähnelt dem Format der Präfixdatendatei. In der ersten Zeile steht die

Anzahl der IPs und nachfolgend in jeder Zeile eine IP. Für den Verwendungszweck der Algorithmen spielt es keine Rolle, dass die IPs anonymisiert werden, da sie bloß als Suchdaten gebraucht werden.

3.2 Relevante Datenstrukturen und Methoden

Bei Algorithmen für IP-Lookup gibt es fünf bis sechs potentielle Methoden, die auf der Datenstruktur arbeiten. Zu allererst die Funktionen, die die Datenstruktur für den Lookup vorbereiten oder zerstören; diese werden oft `init` und `destroy` genannt. Dazu kommen die Funktionen, welche die Präfixdaten in die Datenstruktur einfügen und löschen. Diese haben die Bezeichner `insert` und `delete`. Die Suchfunktion `lookup`, welche nach passenden NextHops in den Datensätzen sucht, ist natürlich das Herzstück jedes Verfahrens. Unter Umständen kommt dann noch eine Updatefunktion `update` hinzu, die die Datenstruktur nach jedem schreibenden Zugriff auf Konsistenz und Effizienz testet.

3.3 Mögliche Algorithmen

Wie oben schon kurz erläutert wurde, sind verschiedene Datenstrukturen und die darauf operierenden Algorithmen bezüglich des LPM in einer anderen Arbeit[1] untersucht worden. Im Folgenden werden diese Datenstrukturen und Algorithmen noch einmal kurz angerissen, bevor die relevanten und implementierten Algorithmen eingehend erörtert werden. Eine ausführliche Erläuterung solcher Algorithmen findet sich auch in [7]

Zunächst kann man einfach Felder, beziehungsweise Tabellen oder Listen zum Speichern von Präfixen verwenden. Bei geeigneter Ablage im Speicher können diese dann auch durch einfaches Anhängen leicht eingefügt werden.

Der Nachteil bei tabellenbasierten Datenstrukturen liegt bei der Suche, da die Präfixe nicht nach dem LPM-Prinzip durchsucht werden, sondern ein Exact-Matching-Scheme durchgeführt wird, wobei sich der Suchalgorithmus das längste Präfix zusätzlich merken muss. Dadurch werden zu viele, fast beliebig redundante Bitvektoren verglichen, was vor allem bei IPv6 sehr viel Zeit kosten kann. Außerdem muss bei einem Update die gesamte Liste sortiert werden, da sonst bei jedem Lookup die komplette Forwarding-Tabelle nach dem längsten existierendem Präfix durchsucht werden muss.

Eine weitere Möglichkeit sind Wörterbücher. Diese haben den Vorteil, dass die Suche nach einem bestimmten Präfix aufgrund der Wörterbucheigenschaft nur konstante Zeit braucht. Auch hier besteht wiederum das Problem, dass nicht nur das Präfix selbst wichtig ist, sondern auch seine Länge. Dies kann die Suche erheblich erschweren, wenn man nach dem längsten passenden Präfix fahndet. Weiterhin sind die variabel langen Präfixe in einem Wörterbuch mit gleich langen Schlüsseln schlecht verwaltbar. Dafür ist die Updatefunktion sehr effizient, da hier keine Operation vonnöten ist.

Die dritte Möglichkeit sind Bäume, die sich aufgrund ihrer Struktur bei Schlüsseln hervorragend für LPM eignen. Hierbei stellt ein Weg durch den Baum den Schlüssel für den gespeicherten Wert dar; bei IP-Lookup-Algorithmen ist das der NextHop.

3.4 Aktuell eingesetzte Algorithmen und Datenstrukturen

Zurzeit wird in den aktuellen High-End-Routern eine hardware-basierte Speichertechnologie namens „Ternary Content Addressable Memory“, kurz TCAM, eingesetzt. Mit TCAMs ist es möglich schnelle Adresslookups in großen Tabellen durchzuführen[7]. Ansonsten werden Patricia Trees oder Hash-Verfahren eingesetzt, wie zum Beispiel auch in dem unixoiden Betriebssystem FreeBSD. Dort wird schon seit längerer Zeit eine BSD Trie Implementation verwendet. Die hier untersuchten Algorithmen stellen daher Alternativen zu den nicht-CAM-basierten Verfahren dar.

4. Verwendete Algorithmen

In den nächsten Abschnitten werden die drei Algorithmen vorgestellt, die implementiert worden sind. Dies ist ein Bruteforce-Ansatz, der den Worst-Case darstellen soll. Dann wird das Konzept der Patricia Trees eingeführt, und schließlich der Elevator-Stairs-Algorithmus vorgestellt.

4.1 Der Bruteforce-Ansatz

Das Feld für die Präfixdaten wird dynamisch allokiert und bei jeder Einfügeoperation werden die neuen Daten angehängt. Die Lookup-Operation durchsucht das komplette Feld nach dem längsten Präfix, wobei es sich merkt, wie lang die Bitlänge jedes gefundenen Präfixes ist. Die Delete-Operation löscht einfach den Eintrag im Feld, behält jedoch den allokierten und auf 0 gesetzten Speicher. Update hat hier keinerlei wirkliche Funktionalität. Zu betonen ist, dass dabei weder eine Komprimierungs- noch eine Sortieroperation verwendet werden.

4.2 Patricia Trees

Der zweite Algorithmus wurde nach dem Konzept der Patricia Trees aufgebaut. Oft wird diese Datenstruktur aufgrund ihrer Geschichte auch als Patricia Trie bezeichnet, aber dieser Name wird hier aus Gründen der Relevanz und Übersichtlichkeit nicht verwendet. Die Ursprünge der Patricia Tries können in [5] nachgelesen werden. Ein Patricia Tree ist ein Binärbaum, wobei die Kanten Bitsequenzen eines Schlüssels darstellen. Es wäre auch möglich, einen Patricia Tree auf Feldern zu implementieren. Dies wird in [5] dargestellt, hier wird allerdings eine wirkliche Baumstruktur verwendet. Das Besondere an dieser Datenstruktur im Gegensatz zu öfters genutzten Bäumen wie AVL-Bäumen ist, dass die Werte nicht nur in den Blättern, sondern auch mitten im Baum sitzen können. Der prinzipielle, nicht optimierte Aufbau eines Patricia Trees ist einfach: Das linke Kind eines Knotens hat mindestens eine binäre Null als Anfang seiner Bitsequenz, und das rechte Kind eine binäre Eins. Falls eine Bitsequenz für ein Kind mehr als ein Element hat, wird die Bitsequenz von links nach rechts auf Gleichheit überprüft. Je nach Art der Schlüssel und deren Wert kann der Baum sehr leicht entarten.

Präfix	NextHop
01	A
00	B
010	C
11	D
1	E

Tabelle 3: Forwarding-Tabelle für den Patricia Tree Aufbau

Die Präfixtabelle aus Tabelle 3 würde in folgenden Patricia Tree in Abbildung 1 umgesetzt werden.

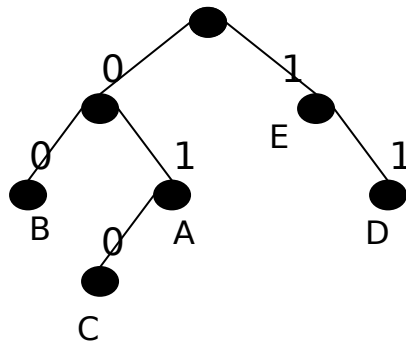


Abbildung 1: Beispiel eines Patricia Trees

Doch selbst bei Patricia Trees würde die Entartung zuviel Laufzeit kosten, wenn zum Testen des längsten Schlüssels sehr viele Kanten bis zum Ende vom Algorithmus gegangen werden müssen. Aus diesem Grund werden entartete Bäume ohne Verzweigung kollabiert, wie in Abbildung 2.

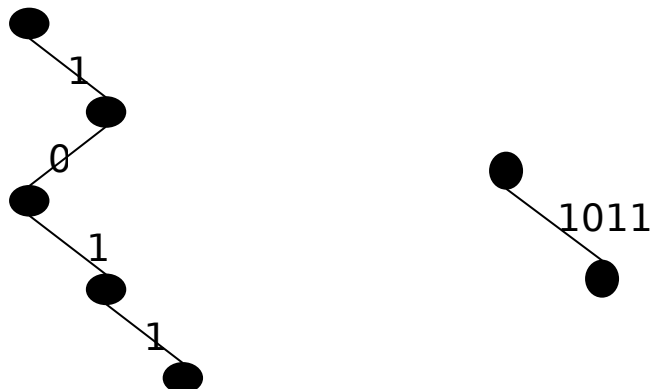


Abbildung 2: Kollaps einer Kante in einem Patricia Tree

Wenn mehrere Kanten ohne Verzweigung oder Information hintereinander existieren, können dessen ein- oder auch mehrelementige Bitsequenzen auf eine mehrelementige Bitsequenz

zusammengezogen werden. Dabei ist dieser Kindknoten ein direkter Nachfahre des Vaters. Dies hat den Vorteil, dass beim Suchdurchlauf von einer Kante, sofern sie vollständig im Suchstring enthalten ist, direkt zum nächsten Knoten weitergeschaltet werden kann. Sollte nur ein Teil des Bitvektors mit dem Bitstring übereinstimmen, muss trotzdem abgebrochen werden. Wenn man einen neuen Knoten einfügen will, ist in der Praxis die Fallunterscheidung nötig, ob die aktuelle Bitsequenz passt, oder – falls nicht – wie weit sie übereinstimmt. Dann muss man unter Umständen eine kollabierte Bitsequenz genau vor dem Bit teilen, welches nicht mehr auf den Bitstring des einzufügenden Knotens passt.

Wenn ein Knoten aus einem Patricia Tree gelöscht wird, muss der Löschalgorithmus beachten, ob der Baum auch nach dem Löschvorgang immer noch maximal kollabiert ist. Dabei ist zu unterscheiden, ob der Wert für einen Schlüssel in einem Blatt steckt, oder sich mitten im Baum befindet. Wenn sich der Wert mitten im Baum befindet, also Kinder hat, wird die Information gelöscht. Falls dabei eine Kante entsteht, weil der Knoten, der ehemals die Information hatte, nur ein Kind hat, werden diese zwei Kanten kollabiert. Falls der Knoten mit dem Schlüssel ein Blatt ist, so wird der Knoten samt Information gelöscht, und eventuell das Nachbarkind mit dem Vater kollabiert, sofern der Vater keine Information enthält.

Die Eigenschaft der maximalen Kollabierung wird in [3] auch als “Path-Compressed Trie” bezeichnet.

Ein weitergehender Ansatz ist, dass man nicht nur zwei Kinder hat, sondern pro Kante eine feste Bitlänge mit mindestens zwei Bit nimmt. Dadurch hat man 2^x Kinder, wobei x die Länge der Bitsequenz ist. Das Problem bei diesen MultiStride-Bäumen ist, dass ein Knoten, abhängig von der Kantenbitlänge, möglicherweise mehrere Informationen speichern muss, da in einer Kante mehrere Schlüssel enden können. Die Wahrscheinlichkeit dafür steigt proportional mit der Anzahl dieser festen Bits pro Kante.

Folglich sind die Operationen auf Patricia Trees etwas umfangreicher als beim Tabellenansatz. Zwar weiß man, dass per Definition die Bitsequenz des linken Kindes mit Null beginnt, und korrespondierend die des rechten Kindes mit Eins, allerdings weiß man noch nichts über die Länge der Bitsequenzen.

Die Initialisierungs-Funktion besteht aus dem Erzeugen eines Wurzelknotens, der eine Bitstringlänge von 0 hat. Die Suchoperation geht, abhängig von den Bitsequenzen der Kinder weiter, bis sie den letzten Knoten oder die letzte Information gefunden hat. Es müssen hierbei mehrere Fälle unterschieden werden:

Zum einen muss beim Übergang vom Knoten zu seinem Kind überprüft werden, ob die Bitsequenz des Kindes am richtigen Offset vollständig im zu suchenden Bitvektor enthalten ist. Ist dies nicht der Fall, bricht man die Suche ab. Aus diesem Grund holt man sich aus dem scheinbar passenden Knoten die vorhandene Information und erneuert sie, falls man später eine andere Information für einen spezifischeren Schlüssel findet.

Die Einfügeoperation geht ähnlich wie die Suchoperation vor. Sie wandert erst solange durch den Baum, wie schon ein Präfix des neuen Schlüssels vorhanden ist. Wenn dieser Schlüssel zu Ende ist, wird der neue Schlüssel einfach angehängt. Sollte aber die Bitsequenz einer Kante nicht vollständig einen Teil des Bitvektors enthalten, so muss diese Bitsequenz in zwei Knoten aufgeteilt werden.

An dem entsprechend der Bitsequenz noch passenden Knoten wird dann das Restpräfix mit der Information angehängt.

Beim Anhängen des Restpräfixes kann man auf zwei Arten vorgehen. Entweder man hängt gleich ein vollständig kollabiertes Präfix an, oder man erzeugt pro einzufügendem Bit einen neuen Knoten. Das kann effektiver sein, wenn man gerade sehr viele Präfixe einfügt, und dann alles zusammen kollabiert. Allerdings ist es in allen anderen Fällen, wenn der Baum schon aufgebaut ist, ineffizient.

Man kann hier eine wirkliche Updatefunktion hinzufügen, die allerdings bezüglich der Laufzeit sehr kostenintensiv ist. Wenn man nicht optimal einfügt oder löscht und eventuell kollabierbare Kanten innerhalb des Baumes entstehen, kann die Updatefunktion diese beseitigen, indem sie durch den Baum iteriert. Die Updatefunktion ist allerdings sehr kostspielig und, falls sie nach nur einem Löschvorgang aufgerufen wird, nicht sehr effizient. Dies resultiert auf der Eigenschaft der Update-Operation, auf jeden Fall komplett durch den Baum zu iterieren, auch wenn nur eine Kante kollabiert werden muss.

Die Löschfunktion geht davon aus, dass der Patricia Tree maximal kollabiert ist. Wenn dies der Fall ist, muss ein Knoten mit Information, welcher nur ein Blatt ist, einfach gelöscht werden. Unter Umständen muss das Nachbarkind danach mit dem Vater kollabiert werden. Innerhalb des Baumes wird die Information einfach gelöscht.

4.3 Der Elevator-Stairs-Algorithmus

Der Elevator-Stairs-Algorithmus wird in [4] von Sangireddy, Futamura und Aluru beschrieben. Dieser Algorithmus soll den Vorteil von Wörterbüchern mit den oben vorgestellten Patricia Trees kombinieren.

Die Datenstruktur verwendet einen Baum aus Wörterbüchern, sowie einen parallelen oder viele kleine angehängte Patricia Trees, je nach Art der Konstruktion.

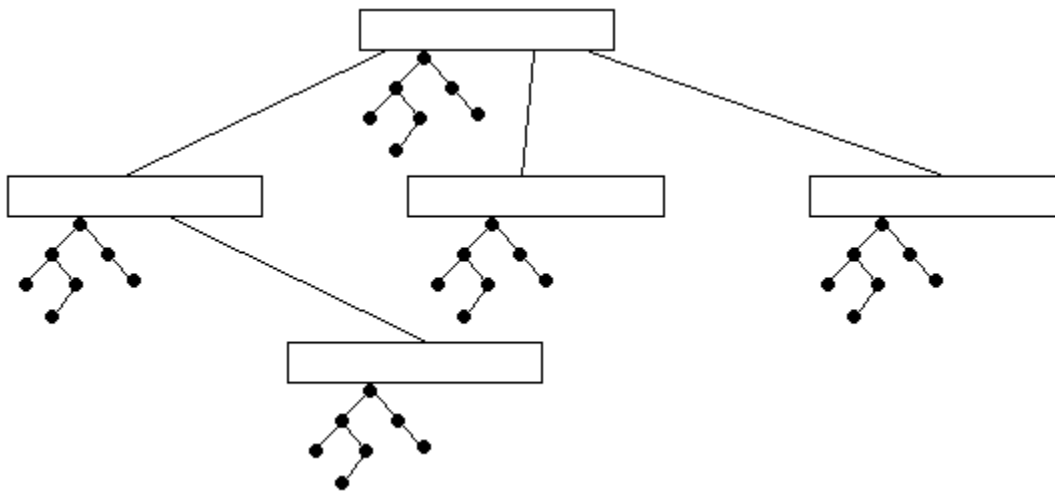


Abbildung 3: Die Elevator-Stairs-Datenstruktur

In Abbildung 3 sieht man einen Baum aus Wörterbüchern, wobei an jeden Knoten des Baumes ein kleiner Patricia Tree angehängt wird.

Wenn man Wörterbücher benutzen will, braucht man für eine effiziente Anwendung allerdings erst konstante Stringlängen, um diese einfügen zu können. Deshalb setzt man eine Bitvektorlänge k fest, die eine signifikante Teillänge der Adresslänge sein sollte.

Dadurch kann man immer in den Wörterbüchern einen Teil der Adresse festlegen, die dann als Ziel den weiteren Weg durch den Wörterbuchbaum bestimmt.

Es gibt jetzt zwei mögliche Arten die Gesamtdatenstruktur für den Elevator-Stairs-Algorithmus festzulegen. Zum einen kann man erst einen vollständigen Patricia Tree aufbauen, der durch eine zweite Datenstruktur, einen Wörterbuchbaum, auch Hashtable-Tree genannt, ergänzt wird. Dieser Wörterbuchbaum hat in jedem Knoten ein Wörterbuch, wobei für jedes Präfix ein Eintrag im Wörterbuch existiert, falls ein solches Präfix vorhanden ist.

Der Vorteil ist, dass man in konstanter Zeit k Schritte weiterspringen kann. Dadurch spart man sich die Abfrage, welche im Patricia Tree für jede Kante vollzogen werden muss. Weiterhin kann man je nach gewähltem k viel größere Sprünge in der Präfixabfrage vollziehen. In jedem Knoten des Wörterbuchbaums wäre dann ein Eintrag für den vom Bitvektor entsprechenden Knoten im Patricia Tree.

Das hätte für den Patricia Tree beim Kollabieren in der Updatefunktion zusätzlich die Prämisse, dass Knoten, die dem k -ten Level im Wörterbuchbaum entsprechen, nicht mit den anderen kollabiert werden dürfen, falls dieser Vorgang den Level für die Bitsequenzkante zerstören würde.

Der Nachteil dieser Bauart ist allerdings, dass der Verwaltungsaufwand, die beiden Datenstrukturen synchron zu halten, kompliziert werden kann. Denn unter Umständen müssen die Wörterbücher des Hashtable-Tree vergrößert werden, falls der Platz für neue Einträge zu knapp wird. Weiterhin muss man bei schreibendem Zugriff aufpassen, dass die Datenstrukturen optimal und synchron bleiben.

Die zweite Möglichkeit diese Datenstruktur aufzubauen, ist etwas leichter als die oben beschriebene.

Dabei wird zuerst der oben beschriebene Wörterbuchbaum aufgebaut. Wie bereits erläutert, kann dieser ja nur mit fixen Präfixlängen umgehen, die ein Vielfaches von k sind. Deshalb hängt man an jeden Knoten des Wörterbuchbaums noch einen Patricia Tree an. Das sind zwar viel mehr Teilbäume. Diese sind allerdings auch viel leichter zu verwalten, da sie maximal die Bitlänge $k-1$ besitzen. Darüber hinaus läuft die Suche in solch kleineren Bäumen schneller. Zusätzlich hat man den Vorteil, dass weniger Teilbäume kollabiert oder auseinander gerissen werden müssen. Dies hängt natürlich immer stark davon ab, wie man den Faktor k wählt.

Je größer k ist, desto schneller kann man zwar im Baum springen, aber desto größer werden auch die Patricia Trees für den einzelnen Knoten, und desto größer die Wahrscheinlichkeit, dass einzelne Kanten auseinander gerissen werden müssen.

Die Korrektheit dieser zweiten Datenstruktur ist leicht zu zeigen. Wenn ein Präfix zum Beispiel eine Bitlänge von $3*k+(k-1)$ hat, wird es auf der dritten Ebene des Wörterbuchbaumes sein, und das Restpräfix $k-1$ wird im Patricia Tree angehängt. Dieser wird definitiv nicht das Level $4*k$ überschreiten, denn wenn dies passiert, dürfte es nicht in den Patricia Tree eingetragen werden, sondern in den passenden Knoten des Wörterbuchbaums der nächsten Ebene.

Bei der Überlegung, wie man geschickt den Wert k wählt, kann man zwei Punkte in Augenschein nehmen. Zum einen kann man k so wählen, dass die Zahl eine Potenz von 2 ist, da der Computer sehr gut mit diesen Zahlen umgehen kann. Andererseits könnte man k in Abhängigkeit der Adresslänge wählen. Weiterhin darf die Tatsache nicht vernachlässigt werden, dass die Wahl des Wörterbuch-Verfahrens die Geschwindigkeit des Algorithmus stark beeinflussen kann. Für die aktuelle Implementierung wurde ein sehr simples Wörterbuch geschrieben, das die Werte direkt abbildet.

4.4 Theoretische Betrachtung

Die theoretische Laufzeit der Algorithmen ist für die Felderimplementierung leicht zu ermitteln, da die betreffenden Algorithmen sehr leicht sind. Die übrigen Komplexitätsbetrachtungen stammen aus [4].

Algorithmen	IP Lookup	Insert/Delete	Speicherverbrauch
Feld	$O(NW)$	$O(1)$ bzw. $O(N)$	$O(NW)$
Patricia Tree	$O(W)$	$O(W)$	$O(NW)$
Elevator-Stairs	$O((W/k)+k)$	$O((W/k)+k)$	$O(NW)$

Tabelle 4: Komplexitätstheoretische Grenzen der Präfixoperationen

Die Komplexität für das Einfügen oder Löschen von Präfixen ist gleich, da nur die Routing-Information gelöscht werden müsste. Zusätzlich müssen unter Umständen noch Speicheroperationen durchgeführt werden. Diese fallen allerdings nicht ins Gewicht, da diese Operationen nicht die Datenstrukturen betreffen. Die tatsächlichen Laufzeiten werden in Kapitel 5 geprüft. In der tatsächlichen Implementierung ist der Speicherverbrauch für Patricia Tree und die Elevator-Stairs-Datenstruktur größer, weil in den einzelnen Knoten viel Verwaltungsinformation mitgeführt werden muss. Im Feld hingegen passiert dies nicht. Trotzdem bleiben alle Komplexitätsgrößen gültig.

5. Das Framework

Um die aktuell analysierten Algorithmen, wie auch zukünftige gut testen zu können, sollte ein Framework entwickelt werden. Dies besteht aus einer generalisierten Bitvektor-API, drei Algorithmen, den Testfällen, sowie dem Framework selbst. Die Software wurde in C, Version C99, auf FreeBSD 6.1 programmiert und zusätzlich auf Debian Linux 3.1 getestet. Bei der Entwicklung wurde besonders auf Plattformunabhängigkeit und dem Einhalten von Standards Wert gelegt. Zur Analyse der Algorithmen und der Programme wurde `valgrind`, `ddd` und `gprof` herangezogen. In der aktuellen Implementierung sind die Algorithmen und Funktionen auf IPv4 ausgerichtet worden. Die spezifischen Änderungen für IPv6 werden im Weiteren erläutert.

5.1 Konfiguration

Das Framework verwaltet die Testdaten, die Bitvektorfunktionen, die Algorithmen und Datenstrukturen, die man testen will. Es generalisiert den Zugriff auf die Funktionen, und ermöglicht es auch, Statistikdaten zu erheben.

Das Framework, welches in `dsframework.h` und `dsframework.c` ist, besteht hauptsächlich aus einem großen Objekt, in welches Daten und Funktionen eingehängt werden. Zum einen wird die IP-API eingehängt, wodurch die Arbeit mit den Bitvektoren für Präfixe und IPs generalisiert wird. Dadurch wird der Austausch für IPv4 und IPv6 leichter und transparenter in der Leistungsmessung. Des Weiteren können die Eingabedaten aus einer Datei eingelesen und in die Datenstruktur eingefügt werden, damit diese gleich im Speicher für die Operationen verfügbar sind. Dies ist realistischer, da Zugriffe auf Dateien aufgrund der Systemaufrufe sehr viel Zeit kosten. Die Einlese-Operationen werden auch in das Framework eingehängt.

Schließlich werden auch die direkten Funktionen eines Algorithmus eingehängt, damit generalisierte Tests für einen Algorithmus möglich sind.

```
void create(FWBLOB *Objekt);  
void destroy(FWBLOB *Objekt);
```

`create` erstellt und initialisiert ein Framework-Objekt, und mittels `destroy` kann es wieder gelöscht werden.

Über diverse Hooking-Funktionen werden die verschiedenen Prozeduren für I/O, Algorithmen und IP-API eingehängt. Aus diesem Grund sind auch die Prototypen für die Algorithmen soweit wie möglich generalisiert und besitzen keinen speziellen Typ.

```
void HookAlgorithms(FWBLOB *a,
    void* (*algo_init)(FWBLOB *Objekt, void *information, unsigned long param),
    void (*algo_destroy)(FWBLOB *Objekt, void *Datenstruktur),
    void (*algo_insert)(FWBLOB *Objekt, void *Datenstruktur, void *PräfixFeld,
        unsigned long Präfixanzahl),
    void (*algo_update)(FWBLOB *Objekt, void *Datenstruktur, void *generic),
    void* (*algo_lookup)(FWBLOB *Objekt, void *Datenstruktur, IP *adresse),
    void (*algo_delete)(FWBLOB *Objekt, void *Datenstruktur, IP *Präfix)
);
```

Die Funktion `algo_init` erzeugt die Datenstruktur, auf denen die IP-Lookup-Algorithmen arbeiten. Falls hierbei beliebige Parameter benötigt werden, können diese über `information` und `param` im Testfall definiert werden.

Analog zerstört `algo_destroy` die Datenstruktur wieder. Über `algo_insert` können mehrere Präfixe auf einmal eingefügt werden. Typischerweise ist das jedes Präfix vom Typ *komponente*, welches das Präfix und die Information für das Präfix beinhaltet. Die Information wird über deren Adresse in einem void-Zeiger gespeichert. `algo_update` ist nicht für alle Typen von Algorithmen wichtig. Trotzdem wird hier diese Funktion definiert, damit erforderliche Optimierungsoperationen auf den Datenstrukturen durchgeführt werden können.

`algo_lookup` ist die Prozedur, die die Suche nach der Forwarding-Information für eine bestimmte IP durchführt. Sie bekommt die Datenstruktur und die IP als Parameter. `algo_delete` löscht die Information für ein bestimmtes Präfix. Die eingehängten Funktionen können über die in 5.3 beschriebenen Operationen aufgerufen werden.

Die Hooking-Funktion, welche die IP-Funktionen einhängt, hat die gleiche Parameter-Struktur, wie die IP-Funktionen selbst. Die Funktionen, welche die Daten ins Framework hängen und damit arbeiten, werden über folgende Funktion selbst eingehängt:

```
void HookIOFunction(FWBLOB*,
    void (*Process)(FWBLOB *Objekt),
    void (*GRD)(FWBLOB *Objekt, FILE *Präfixdaten),
    void (*GSD)(FWBLOB *Objekt, FILE *Suchdaten),
    void (*DO)(FWBLOB *Objekt, FILE *ErgebnisDaten)
);
```

`Process` startet die Funktion, welche die Algorithmen testet. Sie ist der so genannte Testfall. Mittels `GRD` werden die Präfixdaten aus der Datei ausgelesen und im Objekt verzeichnet. Dasselbe geschieht analog mit den Suchdaten über die Funktion `GSD`. Falls in `Process` Daten bezüglich der Laufzeit, der Aufrufe oder der Speicherkomplexität erhoben werden, so können diese ermittelten Werte in die Ergebnisdaten-Struktur geschrieben werden

Die Funktionen zum Einlesen der Präfixdaten und IPs wurden komplett vor die eigentliche Testfunktion gestellt, so dass sämtliche Daten schon im Speicher existieren und die Präfixoperationen diese nicht von der Festplatte holen müssen. Dies hätte die Tests sehr verfälscht, da die Systemaufrufe für Lese-Operationen von der Festplatte im Gegensatz zu den Algorithmen sehr viel Laufzeit kosten. Während des Testlaufs wurden Zähler mitgeführt, die verschiedene Daten erhoben. Diese Daten wurden nach Beendigung der Tests für den Mittelwert und die Standardabweichung normiert und in eine Ausgabedatei geschrieben.

5.2 Bitvektoroperationen

Das Framework besteht aus zwei Teilen. Zum einen wurde eine Bitvektor-API erstellt, die den Zugriff auf IP-Objekte über Funktionen abstrahieren soll. Hierfür wurde ein IP-Objekt geschaffen, das folgenden Inhalt hat:

```
typedef struct sip {
    unsigned int bytes;
    unsigned int effectivbits;
    char typ;
    unsigned char IP_Array[bytes];
} IP;
```

Dieses Objekt kann beliebige IPs oder Präfixe speichern. In der aktuellen Implementierung wurde es auf IPv4 ausgerichtet, so dass `IP_Array` mit `IPv4_BYTELEN`, welches eine Konstante von 4 ist, initialisiert wird. Das ist auch die Zahl, die in der Komponente `bytes` gespeichert ist. Für IPv6 müsste dort dann maximal 16 stehen. Bei IPv4 wurden aufgrund der Einfachheit und der Anzahl der Bytes immer konstant 4 Byte allokiert. Für IPv6 könnte diese Byteanzahl in `IP_Array` variabel sein, um einen zu großen Overhead zu vermeiden.

Die Komponente `effectivbits` gibt die Anzahl der tatsächlich genutzten Bits im `IP_Array` an, da das Objekt ja auch ein Präfix sein kann, welches eine ungerade Bitlänge besitzt. `Typ` beschreibt schließlich die Art des Objekts: Möglich ist hier `IP`, `Präfix`, oder ein einfacher Bitstring.

Für diese IP-Objekte gibt es auch eine API, welche es den Algorithmen ermöglicht mit diesen generalisierten Bitvektortypen zu arbeiten. Für eigene Implementierungen, wie zum Beispiel für IPv6 kann man diese Implementierung in `ip.c` anpassen; die Prototypen sollten aber aufgrund der Framework-API, welche später detailliert dargestellt wird, möglichst nicht verändert werden.

```
int Ipv4ToStruc(char *text, unsigned int textlaenge, unsigned int efbits, char typ , IP
*objekt);
```

`IPv4ToStruc` ist eine Funktion, die aus der ASCII-Darstellung eines Präfixes oder IP ein entsprechendes Objekt an der Speicherstelle `objekt` schafft. Dies hängt von den Werten ab, die in der Komponente `efbits` und in der Komponente `typ` übergeben werden. Im Fehlerfall wird ein Wert ungleich 0 zurückgegeben. Diese Funktion ist als Prototyp auch für IPv6 deklariert, aber nicht implementiert.

```
int createIP(unsigned char *Feld, unsigned int len, unsigned int bitlen, char typ, IP
*objekt);
```

`createIP` holt aus dem `Feld`, welches `len` bytes lang ist, die Bytes und konvertiert diese gegebenenfalls passend für das `objekt` vom Typ `IP`. Außerdem kann noch die real existierende Bitlänge mittels `bitlen` und der Typ des Bitstrings an das `objekt` übergeben werden. Für `objekt` muss dabei schon Speicher reserviert sein.

Im Erfolgsfall gibt `createIP` 0 zurück, ansonsten eine Fehlernummer ungleich 0. Dies ist aktuell jedoch nicht implementiert. Alle nicht belegten Bits sind auf 0 gesetzt.

```
int even(IP *Obj_A, IP *Obj_B);
```

`even` vergleicht zwei Objekte, ob die Bitstrings `Obj_A` und `Obj_B` komplett gleich sind.

```
int PrefixTest(IP *Obj_A, IP *Obj_B, unsigned int offset, unsigned int len);
```

`PrefixTest` vergleicht bei zwei Bitvektoren ab dem `offset` für die Bitlänge `len`, ob diese gleich sind. Es wird dabei zurückgegeben, wie viele Bits denn dann wirklich übereinstimmen. Wenn der verglichene Bitstring vollständig gleich ist, so ist der Rückgabewert gleich `len`.

```
int BitCut(IP *Obj_A, unsigned int offset, unsigned int len, IP *Obj_B);
```

`BitCut` schneidet aus `Obj_A` Bits vom `offset` ab inklusive `len` Bits aus und kopiert sie nach `Obj_B`. Alle nicht benutzten Bits im `IP_Array` sind wiederum auf 0 gesetzt. Im Erfolgsfall wird 0 zurückgegeben, sonst ein Wert kleiner 0.

```
int BitJoin(IP *Obj_A, IP *Obj_B);
```

`BitJoin` hängt an das `Obj_A` das Objekt `Obj_B` an, sofern die Bitlängen beider Bitvektoren nicht die maximale Adresslänge übersteigt.

```
int BitRemove(IP *Objekt, unsigned int offset);
```

`BitRemove` löscht im Objekt ab dem Bit `offset` bis zum Ende des Bitstrings alles. Bei Erfolg wird 0 zurückgegeben, im Fehlerfall ein Wert kleiner 0.

```
int destroyIP(IP *Objekt);
```

`destroyIP` löscht den Bitvektor und gibt im Erfolgsfall eine 0 zurück.

```
int AtOffset(IP *objekt, unsigned int offset);
```

`AtOffset` zeigt an, ob im Objekt an dieser Stelle eine Null oder eine Eins steht.

Falls auf IPv6 umgestellt wird, könnte unter Umständen eine Funktion vonnöten sein, die die Länge des Objektes zurückgibt. Dabei müssten die Bytes des Objektes und zusätzlich die Bytes des Bitvektors zurückgegeben werden. Bisher ist aber kein solcher Anwendungsfall aufgetaucht.

5.3 Präfixoperationen

Wenn man jetzt innerhalb des Testfalls auf die Funktionen des eingehängten Algorithmus zugreifen will, verwendet man die Funktionen aus der Framework-API.

```
static inline void* df_ds(FWBLOB *a);
```

Mittels des Makros `df_ds` ist es möglich direkt auf die verwendete Datenstruktur zuzugreifen.

```
static inline void* df_init(FWBLOB *a, void *b, unsigned long c);
```

`df_init` ruft die für die Datenstruktur korrekte Initialisierungsfunktion auf, und gibt den Zeiger darauf zurück. Zusätzlich ist es möglich, bei der Initialisierung zusätzliche Parameter anzugeben.

```
static inline void df_destroy(FWBLOB *a, void *b);
```

`df_destroy` löscht die aktuell verwendete Datenstruktur. Diese wird über den void-Zeiger übergeben.

```
static inline void df_insert(FWBLOB *a, void *b, void *c, unsigned long d);
```

Mittels `df_insert` ist es möglich, der Datenstruktur `b` eine Reihe von Präfixen zu übergeben. Die Anzahl der Präfixe muss in `d` stehen, wobei die Präfixe selbst als Feld in `c` stehen.

```
static inline void df_update(FWBLOB *a, void *b, void *c);
```

`df_update` kann die Datenstruktur aktualisieren. Diese Funktion ist nicht immer notwendigerweise implementiert. Ihr wird die Datenstruktur `b` übergeben, und eventuell zusätzliche Information über den void-Zeiger `c`.

```
static inline void* df_lookup(FWBLOB *a, void *b, IP *c);
```

Mittels `df_lookup` ist es möglich, in der Datenstruktur `b` mit dem Präfix `c` die gewünschte Information zu bekommen. Diese wird über den void-Zeiger zurückgegeben.

```
static inline void df_delete(FWBLOB *a, void *b, IP *c);
```

`df_delete` löscht ein bestimmtes Präfix `c` aus der Datenstruktur `b`.

6. Evaluation

Es stellt sich die Frage, ob die implementierten Funktionen wirklich effizienter sind. Deshalb wurde eine Laufzeitanalyse der Funktionen erstellt. Es wurden hierbei der Brute-force-Ansatz, der Lösungsansatz über Patricia Trees, sowie der Elevator-Stairs-Algorithmus eingesetzt. Hierfür wurden Präfixe und Suchdaten eingelesen, die dann wiederholt eingefügt, gelöscht und abgefragt werden.

6.1 Beschreibung des Experiments

Bei der Erstellung der Testprogramme für den Brute-force, Patricia Tree und den Elevator-Stairs-Algorithmus wurden die Präfixfunktionen auf folgende Weise ausgeführt:

6.1.1 Beschreibung der Zeitmessung

In einem Durchlauf im Programm wurden erst die Datenstruktur initialisiert, die Präfixe eingefügt und nach allen IPs gesucht. Weiterhin wurden dann alle Präfixe aus der Datenstruktur gelöscht, und danach die Datenstruktur selbst auch. Davon wurden insgesamt 12 Durchläufe innerhalb des Programms erstellt, wobei die Testdaten, beziehungsweise Messwerte, vom zweiten bis zum zwölften Durchlauf für die Mittelwerte und Standardabweichung verwendet wurden. Für die Vorgänge des Einfügens, Suchens und Löschens wurden Zeitintervalle gemessen, wodurch der Zeitverbrauch der Funktionen ermittelt wird.

Das Zeitintervall für die Dauer einer Funktion ist über die Funktion `gettimeofday` ermittelt worden. Vor dem Aufruf einer jeden Prozedur im Testfall wurde der Zeitpunkt gemessen, sowie danach. Die Werte wurden mittels `convtms` in Werte vom Typ `long` konvertiert, und die Differenz ergibt jeweils die Dauer der Ausführung von `df_insert`, `df_lookup` oder `df_delete`.

Durch die zwölf Durchläufe bekommt man eine Reihe von Messdaten, die erst in einen Datentyp „Operationen pro Sekunde“ konvertiert wurde. Dabei hat man durch die Anzahl aller Operationen durch die Millisekunden geteilt, und dann mit 1000 multipliziert. Danach hat man mit den Messwerten, wobei man den ersten nicht verwendet, mittels folgender Formel den Mittelwert und die Standardabweichung errechnet.

$$\begin{aligned} \text{Mittelwert} &= \text{tmp_time_sum} / \text{cnt} \\ \text{Varianz} &= (\text{tmp_time_sum_square} - (\text{tmp_time_sum}^2 / \text{cnt})) / \text{cnt} \\ \text{Standardabweichung} &= \text{sqrt}(\text{Varianz}) \end{aligned}$$

`tmp_time_sum` ist hier bei die Summe der Messwerte, und `tmp_time_sum_square` ist die Summe der quadratischen Messwerte. Die Variable `cnt` ist die Anzahl der Durchläufe, beziehungsweise verwendeten Messwerte.

Es wurden dabei in jedem einzelnen Durchgang 178388 Präfixe eingefügt und gelöscht, und 200000 IPs abgefragt. Die Testergebnisse im folgenden Abschnitt beruhen auf diesen Daten. Die Ermittlung dieser Daten wird in 3.1 erläutert.

Die oben beschriebenen Tests wurden auf einem FreeBSD 6.1 System, auf einer AMD Athlon64 3000+ (2 GHz) Architektur und 1 GB RAM durchgeführt.

6.1.2 Messungen des Speicherverbrauchs der Datenstrukturen

Für die Messungen des Speicherverbrauchs wurde eine Funktion `pattreesize` für die Patricia Tree Datenstruktur und eine Funktion `hshtrsize` für die Elevator-Stairs-Datenstruktur implementiert.

Diese errechnen den Speicherverbrauch einer Datenstruktur, in dem sie durch alle Knoten des Baumes gehen, und für jeden Knoten und darin allokierte Objekte die belegten Bytes zählen und schließlich aufsummieren. Die Messung für den Speicherverbrauch des Bruteforce ist hierbei konstant.

6.2 Ergebnisse

In den folgenden Abschnitten wird erläutert, wie viele Operationen die einzelnen Prozeduren in einer Sekunde durchführen können; und wie oft der Lookup überhaupt Bitvektorfunktionen verwendet. Als letztes wird der Speicherverbrauch der Datenstrukturen angegeben.

6.2.1 Präfix-Operationen pro Sekunde

	<i>Insert</i>		<i>Lookup</i>		<i>Delete</i>	
	Mean	Stddev*	Mean	Stddev*	Mean	Stddev*
Bruteforce	23891.00k	1592.00	0.27k	0.00	1.28k	0.00
Patricia Tree	929.82k	26.80	1326.61k	8.99	1066.54k	9.98
Elevator-Stairs (k=4)	450.14k	25.67	539.55k	21.26	966.43k	26.71

Tabelle 5: Operationen pro Sekunde für die einzelnen Prozeduren

(* bezieht sich auf Kilo Operationen pro Sekunde)

Die Zahlen in Tabelle 5 beschreiben, wie viele Operationen von einer der Prozeduren innerhalb einer Sekunde durchgeführt werden können.

6.2.2 Anzahl der aufgerufenen IP-Funktionen

Um vergleichen zu können, wie stark die Algorithmen die Bitvektorfunktionen beanspruchen, wird die Anzahl der Aufrufe der Bitvektorenoperationen im Mittel in Tabelle 6 dargestellt.

Die Werte stellen die gemittelte Anzahl der Bitvektoroperationen für einen Lookup auf einer Datenstruktur da, in die alle Präfixe eingefügt worden sind.

<i>Lookup</i>	<i>IP-Vergleich</i>	<i>BitCut</i>	<i>Offsetbitabfrage</i>
Bruteforce	178388	0.00	0.00
Patricia	16.25	16.25	17.08
Elevator-Stairs	0.96	6.95	5.56

Tabelle 6: gemittelte Anzahl der Aufrufe der wichtigen IP Funktionen

Wie man an den Zahlen leicht erkennen kann, braucht der Elevator-Stairs-Algorithmus viel weniger Bitvektorfunktionen, als der pure Patricia Tree Algorithmus. Diese wegfallenden Operationen verstecken sich in den Hashtable-Operationen auf der Datenstruktur, die sich allerdings effizienter als ein Patricia Tree Walk realisieren lassen, da die Suche auf Wörterbücher $O(1)$ beträgt.

6.2.3 Speicherverbrauch der Datenstrukturen

Die Datenstrukturen haben unterschiedliche Speicheranforderungen. Für die oben erwähnte Testdatendatei wurden im Schnitt die in Tabelle 7 aufgelisteten Blockgrößen verbraucht. Der Bruteforce läuft hier außer Konkurrenz, da er konstant $(N * (W + \text{PräfixInfo}))$ Bytes Platz braucht.

<i>Datenstruktur</i>	<i>Speicherverbrauch</i>
Bruteforce	8027
Patricia Tree	15951
Elevator-Stairs	19770

Tabelle 7: Speicherverbrauch der Datenstrukturen (in Kilobytes)

Man sieht hier, wie der Performancegewinn durch Speicherplatz erkauft wurde, wobei die Elevator-Stairs-Algorithmen noch verbessert werden könnten, da die reale Größe der

Wörterbücher dynamisch angepasst werden kann, wohingegen in der aktuellen Implementierung immer gleich viel Speicherplatz für eine Hashtabelle allokiert wurde.

6.3 Fazit der Evaluation

Der Elevator-Stairs-Algorithmus konnte leider nicht klar zeigen, dass er durchschnittlich schneller ist als der Patricia Tree Ansatz. Die Wahl der benutzten Wörterbuch-Implementierung ist in diesem Fall allerdings sehr wichtig, da vermutlich der Speicherverbrauch gesenkt werden kann. Die Effizienz des Algorithmus hängt vermutlich stark von der Verteilung der auftretenden IPs ab.

Weiterhin stellt sich die Frage, welche Datenstrukturen auf Rechnerarchitekturen mit anderer Speicherverwaltung bessere Ergebnisse bringen könnten.

7. Zusammenfassung und Ausblick

Das Framework und die Algorithmen, die im Rahmen dieser Abschlussarbeit programmiert wurden, erleichtern das zukünftige Testen von neuen IP/Präfix Lookup Algorithmen sehr. Aus diesem Grund kann, egal bei welchem Algorithmus, die Erhebung von Tests sehr erleichtert werden. Auch wenn der Elevator-Stairs-Algorithmus in den aktuellen Tests nicht effizienter war als der Patricia Tree Ansatz, ist es dennoch wahrscheinlich, dass dieser Potential für einen schnelleren Algorithmus bietet.

Es wäre hierbei interessant zu untersuchen, wie gut sich die einzelnen Algorithmen und Datenstrukturen direkt in die Hardware einbetten lassen, und welchen Leistungsgewinn diese bringen.

Anhang

Wenn man einen neuen Algorithmus mittels des Frameworks testen will, kann man das Testprogramm des Patricia Tree Algorithmus verwenden. Hierbei können die Funktionen zum Einlesen der Daten, zum Ausgeben der Testergebnisse sowie die Testfunktion verwendet werden, falls dies gewünscht ist. Als erstes wird die Framework Struktur via `void create(FWBLOB*)` initialisiert. Dann wird in das Testprogramm die Header-Datei des neuen Verfahrens mit den angepassten Schnittstellen für das Framework eingefügt. Die Funktionen werden dem Framework mittels `HookAlgorithm` übergeben. Wenn eine neue Testfunktion und I/O Funktion für den Algorithmus implementiert werden soll, so werden diese mittels `HookIOFunction` in das Framework integriert. Wenn alle benötigten Funktionen in das Framework integriert worden sind, so werden die Daten mittels `FWBLOB->GetSearchData()` und `FWBLOB->GetRecordData()` eingelesen, und die Testfunktion mit `FWBLOB->Begin()` gestartet. Die Ergebnisse werden dann mittels `FWBLOB->DataOutput()` in eine neue Datei geschrieben. Zum Abschluss wird die Framework-Struktur mittels `void destroy(FWBLOB*)` gelöscht.

```
void create(FWBLOB*);
void destroy(FWBLOB*);
void Process(FWBLOB*);
void GetRecordData(FWBLOB*, FILE*);
void GetSearchData(FWBLOB*, FILE*);
void DataOutput(FWBLOB*, FILE*);
```

Wenn eine eigene Testfunktion selbst programmiert würde, kann auf die in das Framework eingehängten Prozeduren über folgenden inline-Funktionen zugegriffen werden.

```
static inline void* df_init(FWBLOB *a, void *b, unsigned long c);
static inline void df_destroy(FWBLOB *a, void *b);
static inline void df_insert(FWBLOB *a, void *b, void *c, unsigned long d);
static inline void df_update(FWBLOB *a, void *b, void *c);
static inline void* df_lookup(FWBLOB *a, void *b, IP *c);
static inline void df_delete(FWBLOB *a, void *b, IP *c);
```

Wenn neue Verfahren programmiert werden, die im Framework getestet werden, so kann auf die Bitvektor-Funktionen mittels folgender Makros zugegriffen werden. Es muss nur jeder Funktion das Frameworkobjekt übergeben werden.

```

static inline int df_createip(FWBLOB *n, unsigned char *a, unsigned int b, unsigned int
c, char d, IP *e);
static inline int df_destroyip(FWBLOB *n, IP *a);
static inline int df_even(FWBLOB *n, IP *a, IP *b);
static inline int df_bitcompare(FWBLOB *n, IP *a, IP *b, unsigned int c, unsigned int
d);
static inline int df_bitcut(FWBLOB *n, IP *a, unsigned int b, unsigned int c, IP *d);
static inline int df_bitremove(FWBLOB *n, IP *a, unsigned int b);
static inline int df_bitjoin(FWBLOB *n, IP *a, IP *b);
static inline int df_iptext2struc(FWBLOB *n, char *a, unsigned int d, unsigned int b,
char e, IP *c);
static inline int df_thebit(FWBLOB *n, IP *a, unsigned int b);

```

Wenn eigene Input-Funktionen geschrieben werden, die das Einlesen von Präfixdaten und IPs erledigen, so müssen diese Informationen dann mittels `createkomp` in eine Struktur vom Typ `komponente` gepackt werden, und an einen vorher allokierten Platz im Framework abgelegt werden. Mittels `rmkomponente` und `dffree` können diese Informationen wieder freigegeben werden.

```

int create_komp(FWBLOB*, IP*, void*, unsigned int, komponente*);
IP* rmkomponente(FWBLOB*, komponente*);
void dffree(FWBLOB*, void*);

```

Literaturverzeichnis

- [1] Marcel Waldvogel, Georg Varghese, Jon Turner, Bernhard Plattner; 1997: Scalable High Speed IP Routing Lookups. In “Proceedings of SIGCOMM '97”, Seiten 25-26, 1993, <http://citeseer.ist.psu.edu/waldvogel97scalable.html>
- [2] <http://nms.lcs.mit.edu/software/bgp/bgptools>, Stand 12.09.2006
- [3] Miguel Á. Ruiz-Sánchez, Ernst W. Biersack, Walid Dabbous, 2001: Survey and Taxonomy of IP Address Lookup Algorithms. <http://mia.ece.uic.edu/~papers/Surveys/pdf00000.pdf>
- [4] Rama Sangireddy, Natsuhiko Futamura, Srinivas Aluru, Arun K. Somani, 2005: Scalable, Memory Efficient, High-Speed IP Lookup Algorithms, “IEEE/ACM Transactions on Networking”, Volume 13, Issue 4, Seiten 802-812, 2005, <http://portal.acm.org/citation.cfm?id=1088750>
- [5] Donald R. Morrison, 1968: PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric, “Journal of the ACM”, Volume 15, Issue 4, Seiten 514-534, 1968, <http://portal.acm.org/citation.cfm?id=321481>
- [6] Larry L. Peterson, Bruce S. Davie, dpunkt.Verlag 2000: Computernetze – Ein modernes Lehrbuch
- [7] Volker Heun, Vieweg Verlag, 2003: Grundlegende Algorithmen
- [8] http://en.wikipedia.org/wiki/Content_addressable_memory, Stand 12.09.2006