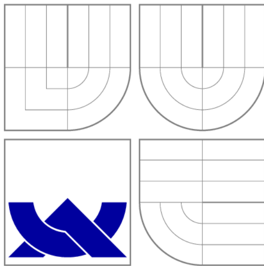
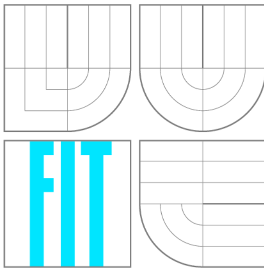


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ALGORITMY PRO VYHLEDÁNÍ NEJDELŠÍHO SHODNÉHO PREFIXU

LONGEST PREFIX MATCH ALGORITHMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. FRANTIŠEK SEDLÁŘ
AUTHOR

VEDOUCÍ PRÁCE Ing. JIŘÍ TOBOLA
SUPERVISOR

BRNO 2013

// Abstrakt

Tato diplomová práce nejprve uvádí čtenáře do problematiky vyhledávání nejdelších shodných prefixů. Analyzuje a popisuje vybrané algoritmy se zaměřením na jejich rychlost, paměťovou náročnost a vhodnost pro hardwarovou implementaci. Na základě získaných poznatků představuje nový algoritmus Generic Hash Tree Bitmap. Ten je mnohonásobně rychlejší než jiné používané metody, zatímco jeho paměťové nároky jsou mnohdy nižší. Implementace algoritmu se stala součástí knihovny Netbench [8].

Abstract

This master's thesis explains basics of the longest prefix match (LPM) problem. It analyzes and describes chosen LPM algorithms considering their speed, memory requirements and an ability to implement them in hardware. On the basis of former findings it proposes a new algorithm Generic Hash Tree Bitmap. It is much faster than many other approaches, while its memory requirements are even lower. An implementation of the proposed algorithm has become a part of the Netbench library [8].

Klíčová slova

nejdelší shodný prefix, trie, LC Trie, Controlled Prefix Expansion, Lulea Compressed Trie, Tree Bitmap, Shape Shifting Tree, Hash Tree Bitmap, Generic Hash Tree Bitmap, binární vyhledávání na intervalech, binární vyhledávání na prefixech

Keywords

longest prefix match, LPM, trie, LC Trie, Controlled Prefix Expansion, Lulea Compressed Trie, Tree Bitmap, Shape Shifting Tree, Hash Tree Bitmap, Generic Hash Tree Bitmap, Binary Search on Intervals, Binary Search on Prefixes

Citace

František Sedlář: Algoritmy pro vyhledání nejdelšího shodného prefixu, diplomová práce, Brno, FIT VUT v Brně, 2013

Algoritmy pro vyhledání nejdelšího shodného prefixu

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně pod vedením pana Ing. Jiřího Toboly. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
František Sedlář
16. května 2013

Poděkování

Rád bych poděkoval především vedoucímu práce Ing. Jiřímu Tobolovi za jeho ochotu a odbornou pomoc.

© František Sedlář, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Vyhledávání nejdelšího shodného prefixu **4**

3 Přehled algoritmů **5**

3.1 Algoritmy založené na struktuře trie . . . 5

3.1.1 Trie . . . 5

3.1.2 Controlled Prefix Expansion . . . 6

3.1.3 Lulea Compressed Tries . . . 8

3.1.4 LC Trie . . . 9

3.1.5 Tree Bitmap . . . 11

3.1.6 Shape Shifting Tree . . . 12

3.1.7 Hash Tree Bitmap . . . 13

3.2 Ostatní algoritmy . . . 14

3.2.1 Binární vyhledávání na intervalech . . . 14

3.2.2 Binární vyhledávání na prefixech . . . 15

4 Návrh algoritmu Generic Hash Tree Bitmap **17**

4.1 Odstranění nevětvených cest . . . 18

4.1.1 Princip . . . 18

4.1.2 Testování . . . 19

4.2 Adaptivní expanze prefixů . . . 21

4.2.1 Princip . . . 21

4.2.2 Testování . . . 22

4.3 Odstranění větvení bez prefixů . . . 24

4.3.1 Princip . . . 24

4.3.2 Testování . . . 25

4.4 Steinerovy stromy . . . 26

5 Implementace **29**

5.1 Netbench . . . 29

5.2 Generic Hash Tree Bitmap . . . 30

6 Testování algoritmů **31**

6.1 Testovací množiny . . . 31

6.1.1 Reálné . . . 31

6.1.2 Generované . . . 31

6.2 Jednotlivé algoritmy . . . 31

6.2.1 Trie . . . 31
6.2.2 Controlled Prefix Expansion . . . 32
6.2.3 Lulea Compressed Trie . . . 32
6.2.4 LC Trie . . . 33
6.2.5 Tree Bitmap . . . 34
6.2.6 Shape Shifting Tree . . . 35
6.2.7 Hash Tree Bitmap . . . 36
6.2.8 Generic Hash Tree Bitmap . . . 36
6.3 Srovnání na reálných množinách . . . 38
6.4 Srovnání na generovaných množinách . . . 40

7 Závěr **42**

A Obsah DVD **45**

Kapitola 1

Úvod

S rozmachem a zrychlováním počítačových sítí rostou nároky nejenom na technologie zajišťující samotný přenos informace (metalické a optické kabely, bezdrátové prvky), ale také na další zařízení, která jsou v této infrastruktuře nezbytná. Ať už jde o prvek na linkové vrstvě, síťové vrstvě či vyšších vrstvách komunikačního modelu, vždy musí pracovat takovou rychlostí, aby výrazně neomezoval kvalitu služeb využívaných v této síti. Kromě zrychlování současně využívaných technologií je samozřejmě třeba aplikovat nové prostředky, které dále zvyšují bezpečnost, spolehlivost a využitelnost.

Jednou z „novinek“, kterou mnozí správci sítí začínají v současné době používat, je protokol IPv6. Jeho nasazení je nezbytné zejména kvůli vyčerpání adresového prostoru protokolu IPv4. Zavedením IPv6 však vyvstávají nové problémy, na které je nutné se připravit. Příkladem může být problém s rychlostí vyhledání nejdelšího shodného prefixu, o kterém pojednává tato práce.

Na vytížené 10 Gb/s lince mohou přicházet pakety každých 67 ns. V tomto čase je třeba projít směrovací tabulku (ta dnes u IPv4 může obsahovat stovky tisíc záznamů) a vyhledat nejdelší prefix, který odpovídá cílové IP adrese. S přechodem na IPv6 vzroste maximální délka každého prefixu z 32 bitů na 128 bitů, což tento úkon dále znesnadní. Dá se navíc předpokládat, že rychlosti linek nadále porostou, stejně tak počet záznamů ve směrovacích tabulkách bude pravděpodobně stoupat.

Cílem vývoje algoritmů pro vyhledání nejdelšího shodného prefixu (LPM) je tedy zvyšovat jejich rychlost, kritická je ovšem i velikost struktury, kterou algoritmus vytvoří pro reprezentaci množiny prefixů. Zatímco při nižších rychlostech lze tyto algoritmy uplatnit v softwarové podobě na běžných počítačích, na kapacitnějších linkách je nutné využít HW implementaci, ať už s využitím architektury ASIC nebo dnes populární a pro vývoj vhodnější FPGA.

Tato práce nejprve v kapitole 2 přibližuje problém vyhledávání nejdelších shodných prefixů, dále v kapitole 3 popisuje základní algoritmy z knihovny Netbench [8], včetně příkladů sestavení potřebných struktur pro konkrétní množinu prefixů. Část 4 zahrnuje postupný návrh a testy nového algoritmu Generic Hash Tree Bitmap. Tento algoritmus byl implementován (viz kapitola 5) a stal se součástí knihovny. Kapitola 6 testuje jednotlivé algoritmy na různých množinách IPv6 prefixů a dosažené výsledky algoritmů vzájemně porovnává.

Kapitola 2

Vyhledávání nejdelšího shodného prefixu

Typickým příkladem použití algoritmů pro vyhledávání nejdelšího shodného prefixu (LPM algoritmů) je směrování a filtrování paketů v počítačových sítích. Směrovač (uvažujme tzv. packet-by-packet router dle RFC1812) obsahuje směrovací tabulku. Ta se skládá z množiny prefixů IP adres, ke kterým má (mimo jiné) přiřazeno výstupní rozhraní. Pro každý paket vstupující na rozhraní směrovače musí najít takový záznam, který je prefixem cílové IP adresy. Platných prefixů pro konkrétní cílovou adresu může směrovač ve své směrovací tabulce obsahovat obecně mnoho. Z těchto musí vybrat ten nejdelší.

Představme si pro jednoduchost, že IP adresa má pouze 8 bitů. Směrovač může obsahovat například tabulku z obrázku 2.1.

Prefix	Out
01	eth0
11	eth1
100	eth2
00101	eth1
101000	eth2
110100	eth3
110101	eth2

Obrázek 2.1: Zjednodušená směrovací tabulka

Pokud by na vstupní rozhraní tohoto směrovače dorazil paket s cílovou IP adresou 223 (to je binárně 11010101), platným prefixem by byl prefix 11, dle kterého se má paket předat na rozhraní eth1. Prefixem je ovšem i záznam 110101, který by paket poslal na rozhraní eth2. Protože je druhý prefix delší, paket se bude směrovat na rozhraní eth2.

Triviální LPM algoritmus by prošel celou množinou prefixů a vybral ten nejdelší shodný. Takový algoritmus by však měl lineární časovou složitost vzhledem k velikosti vstupní množiny, což je neakceptovatelné. Je tedy třeba hledat efektivnější metody. Základní z nich jsou popsány v následující kapitole.

Stejný princip je využíván také při filtrování dle zdrojové nebo cílové adresy ve firewallech, případně při klasifikaci paketů, například pro účely zajištění QOS, tvorbu statistik nebo bezpečnostní analýzy.

Kapitola 3

Přehled algoritmů

V tomto přehledu se objeví základní LPM algoritmy, jejichž implementace je obsažena v knihovně Netbench vyvíjené výzkumnou skupiny ANT [1] na FIT VUT v Brně. Stručná charakteristika většiny z nich je součástí například článku [11].

Pro popis algoritmů byla zvolena malá množina velice krátkých prefixů, která je na obrázku 3.1. U každého algoritmu bude pro názornost naznačen postup tvorby daných datových struktur právě pro tuto množinu.

P1 01
P2 11
P3 100
P4 00101
P5 101000
P6 110100
P7 110101

Obrázek 3.1: Testovací množina prefixů

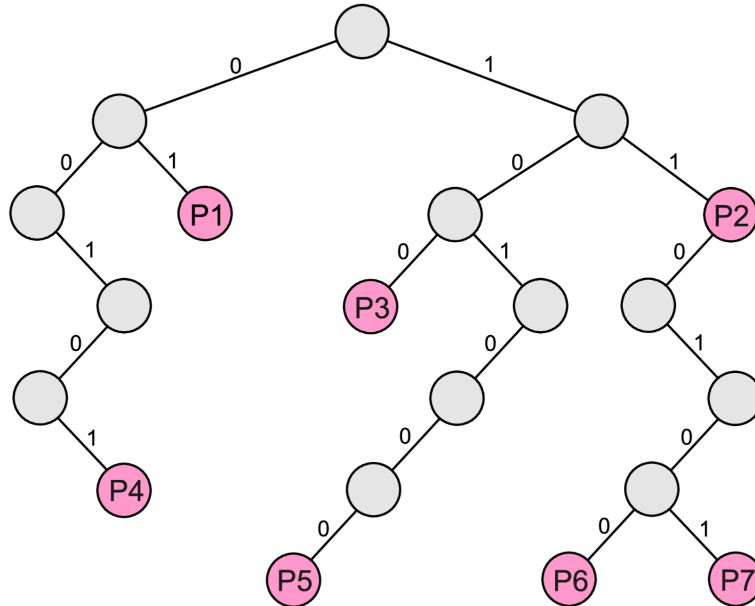
3.1 Algoritmy založené na struktuře trie

Slovo trie pochází z anglického „retrieval“ – získávání, nalezení. Jedná se o jednoduchou datovou strukturu, jejímž cílem je reprezentovat množinu prefixů s malými paměťovými nároky a zároveň tak, aby bylo možné v této struktuře velmi rychle vyhledávat nejdelší shodný prefix k dané vstupní hodnotě. Algoritmů pro vytváření těchto struktur je mnoho, každý je vhodný na jiné případy. Některé jsou velice efektivní, ale nehodí se pro hardwarovou implementaci. Jiné se v praxi téměř nepoužívají, avšak jsou velice názorné. V přehledu uvedeme ty nejzajímavější z nich.

3.1.1 Trie

Trie pracuje s binárním stromem. Jedná se o singlebit algoritmus, což znamená, že vstupní adresu zpracováváme po jednotlivých bitech. Každá hrana mezi uzly představuje jeden bit prefixu (některé hrany reprezentují jedničku, jiné nulu). Prefix reprezentovaný určitým uzlem je tvořen hranami mezi kořenem a tímto uzlem. Každý uzel může obsahovat odkazy na svoje nejvýše dva potomky. Pokud množina prefixů obsahuje tentýž prefix, který je uzlem

reprezentován, je v tomto uzlu uložen i odkaz do tabulky prefixů. V takovém případě říkáme, že uzel obsahuje platný prefix. Trie reprezentující testovací množinu prefixů z obrázku 3.1 můžete vidět na obrázku 3.2.



Obrázek 3.2: Trie

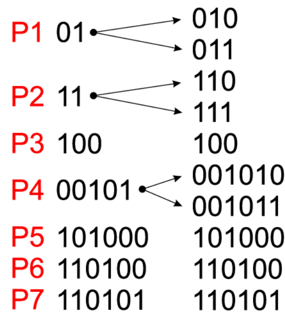
Při vyhledávání prefixu k dané IP adrese se tedy postupuje po jednom bitu adresy směrem od toho nejvýznamnějšího. Hledání začíná v kořenovém uzlu stromu. Pokud je právě zkoumaný bit prefixu jedničkový, dále vyhledáváme v pravém podstromu. V případě nulového bitu se přesuneme do levého podstromu.

Při procházení stromu je nutné kontrolovat, zda právě navštívený uzel obsahuje platný prefix. Pokud ano, poznačíme si jeho hodnotu. Stačí, když si pamatujeme vždy poslední nalezený (tedy nejdelší) prefix. Ten bude po skončení vyhledávání výsledkem. Vyhledávání končí, jakmile narazíme na listový uzel.

Nevýhodou algoritmu je jeho velká paměťová náročnost a velký počet náhodných přístupů do paměti při vyhledávání (to je dáno výškou stromu, tedy délkou nejdelšího prefixu). Vzhledem k použití delších prefixů lze u IPv6 očekávat značné zhoršení obou těchto parametrů.

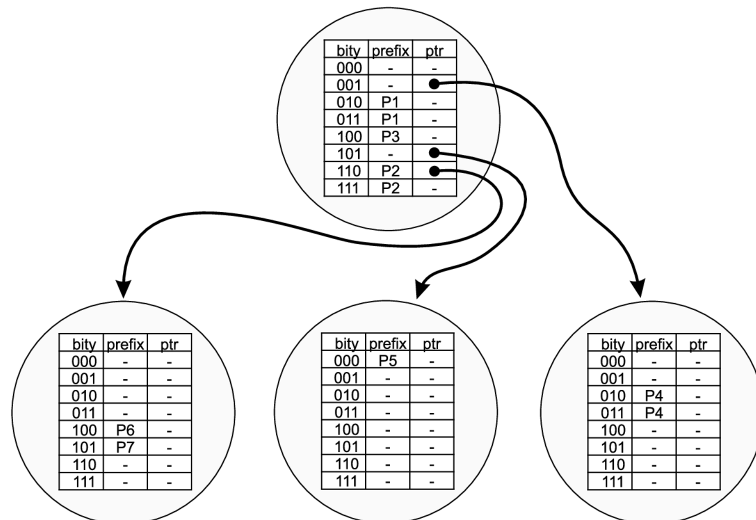
3.1.2 Controlled Prefix Expansion

Controlled Prefix Expansion [10] je v tomto textu také označován zkratkou „CPE“. Jedná se o multibit algoritmus, pro zrychlení se zde pracuje s více bity prefixu v jednom taktu. Pro ilustraci uvedeme trie, která pracuje po trojicích bitů. Protože množina prefixů, které je třeba do trie uložit, nemusí mít délky násobku tří, je třeba některé prefixy upravit, tzv. expandovat. Například prefix číslo 2 z testovací množiny, který má tvar 11, expandujeme na množinu prefixů 110 a 111. Pokud by se ve vstupní množině nacházel například i prefix 111, který je delší než prefix 11 (později expandovaný na 111), použili bychom při budování trie pro posloupnost bitů 111 přímo tento delší prefix 111 a expandovanou hodnotu bychom nebrali v potaz.



Obrázek 3.3: Ukázka expandování uzlů

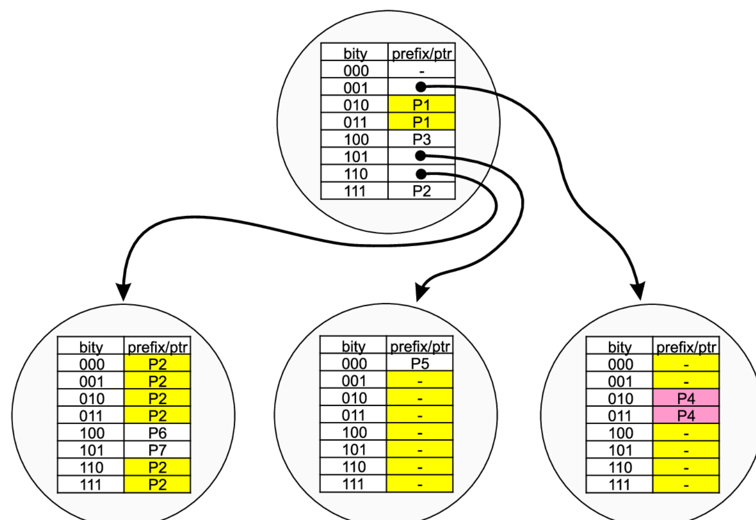
Expandovaná množina prefixů je znázorněna na obrázku 3.3. Každý uzel CPE struktury tedy obsahuje ukazatel do tabulky prefixů a ukazatel na následující uzly pro všechny varianty vstupu (při zpracování po trojicích bitů bude každý uzel obsahovat celkem 16 ukazatelů). To můžeme vidět na obrázku 3.4, kde je znázorněna celá trie pro expandovanou množinu prefixů z obrázku 3.1.



Obrázek 3.4: Trie sestavená algoritmem CPE

Tyto velké paměťové nároky částečně snižuje optimalizace „leaf pushing". Trie je upravena tak, aby uzly pro každý možný vstup obsahovaly pouze jeden ukazatel – buď ukazatel do tabulky prefixů, nebo ukazatel na následující uzel. Celá trie je na obrázku 3.5. Zde jsou zároveň barevně vyznačeny redundantní hodnoty, které se zde bohužel často ukládají. Tuto nevýhodu dále odstraňuje algoritmus Lulea Compressed Trie.

Při vyhledávání je nutné hledaný řetězec rovněž upravit. Zde však stačí prodloužit jej libovolným řetězcem tak, aby jeho délka byla násobkem počtu bitů zpracovávaných v jednom taktu. Po úpravě řetězce postupujeme ve vyhledávání podobně jako v případě algoritmu trie. Začínáme v kořenovém uzlu. Trojici bitů vstupního řetězce použijeme v každém uzlu jako index do tabulky ukazatelů, kde jsou uloženy ukazatele pro všechny možné vstupy. Přes ukazatel uložený na tomto indexu postupujeme k následujícímu uzlu. U optimalizace „leaf pushing" zde můžeme nalézt přímo ukazatel do tabulky prefixů a vyhledávání končí.



Obrázek 3.5: Trie sestavená algoritmem CPE s optimalizací „leaf pushing“

Vyhledávání s „leaf pushing“ může skončit i nalezením nulového ukazatele. V takovém případě žádný prefix pro hledaný řetězec neexistuje. U algoritmu bez optimalizace si postupně ukládáme nalezené shodné prefixy a postupujeme dále dle ukazatele na uzly, dokud tento ukazatel není nulový. V takovém případě vyhledávání končí a jako výsledek použijeme prefix uložený jako poslední (tedy nejdelší).

Obecně lze říci, že jeden uzel trie postavené pomocí základního neoptimalizovaného algoritmu se skládá z 2^n ukazatelů na další uzly + 2^n ukazatelů do tabulky prefixů, kde n je počet bitů zpracovávaných v jednom taktu. S využitím optimalizace „leaf pushing“ pak jeden uzel trie sestává „pouze“ z 2^n ukazatelů.

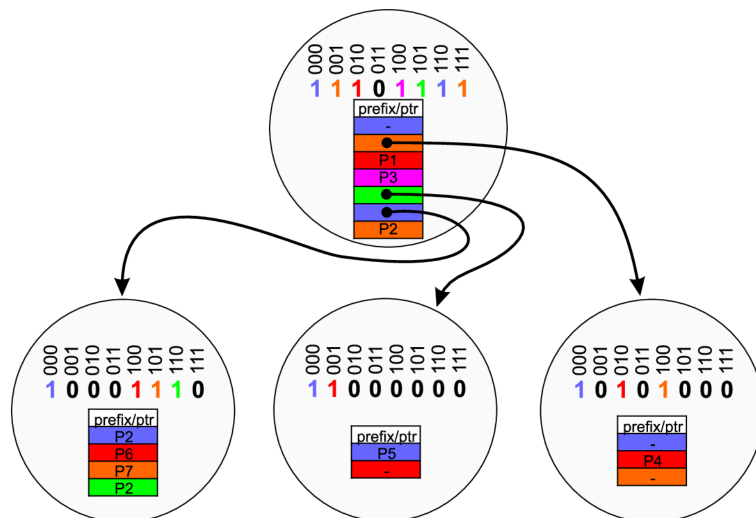
3.1.3 Lulea Compressed Tries

Lulea Compressed Tries [4] je další optimalizací CPE. U „leaf pushing“ se v uzlech často objevuje několik shodných ukazatelů pro různé varianty vstupu (viz obrázek 3.5). Lulea compressed trie (v textu též zkráceně označován jako „Lulea“) se tuto redundanci snaží odstranit.

Každý uzel obsahuje navíc bitmapu. Ta obsahuje 2^n bitů, kde n je počet bitů zpracovávaných v jednom taktu – při práci s trojicemi bitů prefixu tedy uzel obsahuje bitmapu délky 8. Pokud by v uzlu měly být vedle sebe uložené shodné ukazatele, je zachován pouze první výskyt a v bitmapě je na odpovídající pozici označen jedničkou. Další shodné ukazatele nejsou uchovány, pouze jsou v bitmapě označeny nulou. Vstupní prefixy je opět nutné expandovat.

Struktura reprezentující testovací množinu prefixů je na obrázku 3.6. Při vytváření byla použita expandovaná varianta prefixů z popisu algoritmu CPE (viz obr. 3.3). Úsporu paměti, kterou algoritmus oproti CPE přináší, lze pozorovat při porovnání s obrázkem 3.5.

Vyhledávání probíhá obdobně jako u algoritmu CPE. Trojici bitů vyhledávané adresy však použijeme jako index do bitmapy. Spočítáme počet jedniček v bitmapě od MSB až do tohoto indexu. Tento počet poté použijeme jako index do tabulky s ukazateli na další uzel, případně zde můžeme opět nalézt ukazatel do tabulky prefixů a vyhledávání končí. Vyhledávání je rovněž neúspěšné, pokud narazíme na nulový ukazatel.



Obrázek 3.6: Trie sestavená algoritmem Lulea Compressed Tries

Jeden uzel trie sestavené pomocí tohoto algoritmu tedy obsahuje bitmapu velikosti 2^n bitů, kde n je počet bitů zpracovávaných v jednom taktu. Dále obsahuje v nejhorším případě až 2^n ukazatelů, obecně je však ukazatelů méně.

3.1.4 LC Trie

LC Trie [7] je multibit algoritmus, který pracuje s proměnným počtem bitů v jednom taktu. Z toho vyplývá jeho nevhodnost pro hardwarovou implementaci.

Pro úsporu paměti potřebné pro reprezentaci trie využívá algoritmus techniky známé jako path compression a level compression. Obě tyto techniky jsou patrné na obrázku 3.7, který znázorňuje klasickou binární trie, pouze jsou na ní vyznačeny úseky, kde jsou uvedené techniky použity. Tento obrázek lze srovnat s obrázkem 3.8, kde je finální podoba trie postavené algoritmem LC trie.

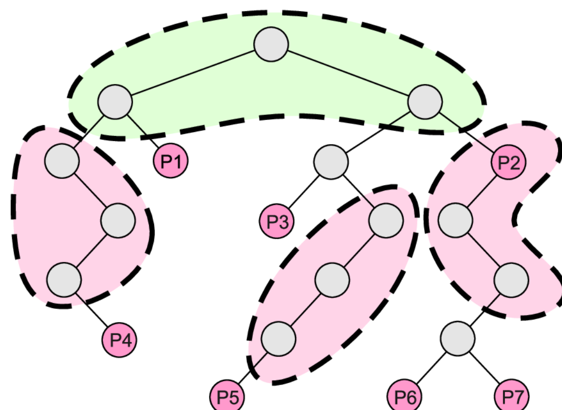
Path compression (na obrázku 3.7 označena červeně) redukuje hloubku stromu a počet uzlů tím, že do trie neukládá všechny bity, některé jsou ignorovány. To se děje v místech, kde by se trie nevětvila, a při vyhledávání v ní by se vždy procházelo přes stejné uzly. Počet bitů přeskočených v daném uzlu je označováno jako skip hodnota tohoto uzlu.

Hodnotu skip je možné v konkrétním uzlu trie určit z množiny ukládaných prefixů, jejichž prefix tento uzel reprezentuje. Prozkoumáváme znaky následující za tímto prefixem. Pokud se všechny řetězce na prvních i znacích shodují, není třeba tyto znaky do trie zahrnovat. Pouze se v tomto uzlu označí počet takto vynechaných bitů.

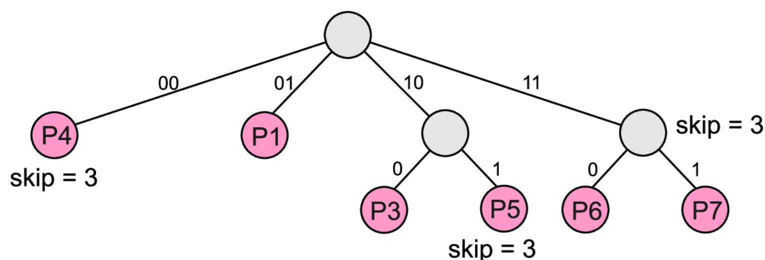
Technika level compression nahrazuje i nejvyšších kompletních úrovní binární trie jedním uzlem stupně 2^i. To je rovněž znázorněno na obrázku 3.7. Dvě nejvyšší kompletní úrovně trie byly nahrazeny jedním uzlem stupně čtyři.

Každý uzel má 2^b přímých potomků, číslo b se nazývá faktor větvení. Ten je rovněž možné určit z množiny řetězců, jejichž prefix tento uzel reprezentuje.

Pokud je uzel listový, faktor větvení je 0, nevětví se. Pokud není listový, je třeba opět zkoumat jednotlivé znaky na celé množině řetězců, které sdílí tento uzel. Nejprve odřízneme celý prefix, který je reprezentován tímto uzlem. Poté ještě odřízneme znaky, které jsou v tomto uzlu označeny jako skip. Nastavíme faktor větvení na 1. Ze všech řetězců vysekneme



Obrázek 3.7: Ukázka komprese trie použité v algoritmu LC Trie



Obrázek 3.8: Trie sestavená algoritmem LC Trie

pouze první dva znaky. Pokud jednotlivé dvojice pokryjí všechny kombinace (tj. hodnoty 00, 01, 10, 11), nastavíme faktor větvení na 2. Poté z řetězců vysekneme pouze první 3 znaky. Pokud obsahují všechny kombinace (000, 001, ..., 111), nastavíme faktor větvení na 3 atd.

Před budováním trie je nutné seřadit množinu prefixů (prefixy vyjádřené jako řetězec nul a jedniček) podle abecedy. Množina nesmí obsahovat prefixy, které jsou platnými prefixy dalších (delších) prefixů. Kratší prefixy přesuneme do zvláštní tabulky. Podrobný popis budování trie je součástí [7].

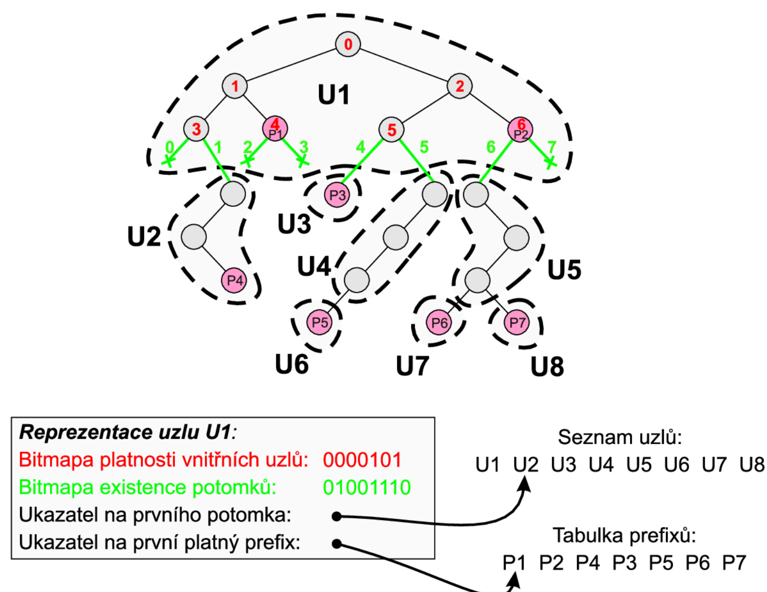
Při vyhledávání začínáme v kořenovém uzlu. Hodnota branch, která je uložená v tomto uzlu, udává, kolik potomků tento uzel má. To pro nás při vyhledávání znamená, kolik bitů vstupního řetězce použijeme pro rozhodnutí, ke kterému synovskému uzlu se přesuneme. Nesmíme zde také zapomenout na hodnotu skip. Pokud je například v kořenovém uzlu trie hodnota branch tři a skip je dva, použijeme třetí, čtvrtý a pátý bit vyhledávané adresy (počítáno od MSB). Tuto hodnotu (vyjádřenou desítkově) přičteme k indexu prvního potomka tohoto uzlu. Sečtená hodnota udává index uzlu, ke kterému se při vyhledávání přesuneme. Při prohledávání tohoto synovského uzlu pak nebereme v potaz prvních 5 bitů vstupního řetězce, které už byly prozkoumány.

Při neúspěšném vyhledání prefixu pro určitou vstupní adresu je třeba projít ještě tabulku, do které jsme při budování trie uložili prefixy, které byly prefixy jiných prefixů. Může totiž dojít ke shodě s nějakým z prefixů zde uložených. Tyto prefixy lze poté projít

lineárně, případně je předem seřadit a použít například binární vyhledávání. Řešením je ovšem i použití speciální hardwarové architektury typu TCAM.

3.1.5 Tree Bitmap

Tento multibit algoritmus byl představen v [5]. Pracuje se stále stejným počtem bitů v jednom taktu (počet je často označovaný jako střída). Při popisu budeme uvažovat variantu zpracovávající trojice bitů. Každý uzel pak má až $2^3 = 8$ následníků. V rámci uzlu je uložena odpovídající část binárního stromu. To je opět vidět na celé sestavené trie na obrázku 3.9. Je zde zobrazena celá binární trie, na které jsou ohraničené jednotlivé uzly Tree Bitmap.



Obrázek 3.9: Trie sestavená algoritmem Tree Bitmap

Každý tento uzel tedy může uvnitř obsahovat až 7 platných prefixů (reprezentovaných 7 vnitřními uzly). Každý uzel trie může mít až 8 potomků (potomci tohoto uzlu jsou potomky až čtyř listových uzlů binárního stromu, který odpovídá tomuto uzlu).

Pokud nějaký vnitřní uzel binárního stromu reprezentuje platný prefix, je v něm uložen odkaz do tabulky prefixů. Do paměti ukládáme pouze ty potomky uzlu, které obsahují nějaký platný prefix. Pro úsporu paměti jsou ukazatele do tabulky prefixů uloženy pouze u těch vnitřních uzlů binárního stromu, které obsahují platné prefixy. Aby bylo toto možné, existuje v rámci každého uzlu bitmapa platnosti prefixů reprezentovaných jednotlivými vnitřními uzly. Pokud trie obsahuje například 7 vnitřních uzlů, jsou tyto očíslovány (v pořadí breadth-first) a v bitmapě je na odpovídajících pozicích buď jednička (tzn. existuje ukazatel do tabulky prefixů), nebo nula. Za předpokladu, že jsou prefixy uloženy v paměti za sebou v požadovaném pořadí, stačí uložit odkaz pouze na první platný prefix vnitřního uzlu.

Podobně je to řešeno s potomky. Uvažovaná trie může mít až 8 potomků – bitmapa potomků udává, kteří potomci existují, a v uzlu je uložen ukazatel pouze na prvního z nich.

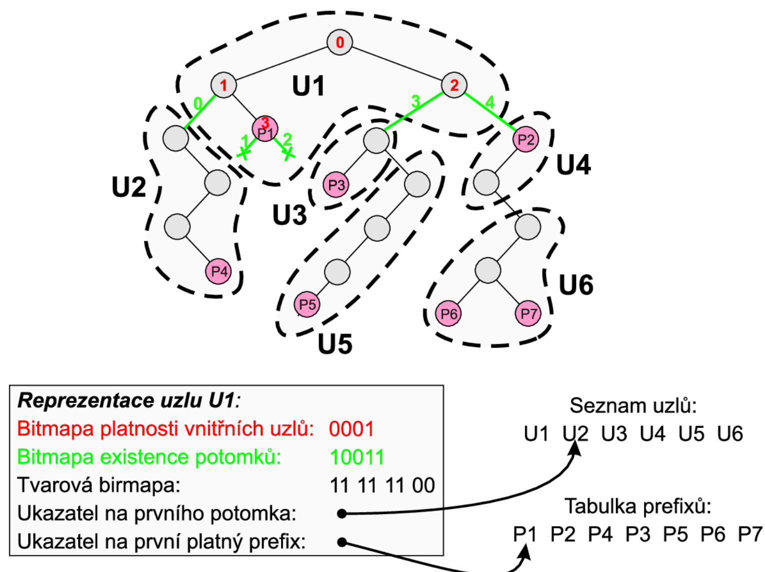
Při vyhledávání rozdělíme vstupní řetězec na části po takovém počtu bitů, který jsme zvolili při budování trie, v našem případě po trojicích. Začneme v kořenovém uzlu s první trojicí bitů. V bitmapě, která reprezentuje vnitřní uzly uzlu, zjistíme, zda se po cestě nachází nějaký platný prefix. Pokud ano, poznamenáme si jej. Stačí si pamatovat vždy ten

aktuálně nejdelší. V bitmapě následníků uzlu zjistíme, zda daným směrem existuje nějaký následník. Pokud v bitmapě na pozici reprezentované trojicí vstupních bitů je uložena jednička, potomek existuje. Jeho adresu určíme z adresy prvního potomka a z počtu jedniček v bitmapě až do pozice reprezentované vstupními bity (počítáno od MSB). Přesuneme se k potomkovi a algoritmus se opakuje. Vyhledávání končí, pokud prozkoumáme všechny bity vstupního řetězce, nebo když v bitmapě potomků zjistíme, že potomek neexistuje.

Na obr. 3.9 je vidět vše, co je uloženo v rámci jednoho uzlu struktury vytvořené algoritmem Tree Bitmap. Jsou to dvě bitmapy: jedna reprezentuje vnitřní uzly, druhá reprezentuje možné potomky uzlu. Dále ukazatel na první záznam v tabulce prefixů, který je platný pro tento uzel, a ukazatel na prvního potomka tohoto uzlu. Dále je možné doplnit ukazatel na rodiče, abychom umožnili procházení stromem v opačném směru.

3.1.6 Shape Shifting Tree

Algoritmus představený v [9] (v tomto textu často nazýván zkratkou „SST“) je velmi podobný algoritmu TreeBitmap, snaží se však o efektivnější rozdělení uzlů binárního stromu do uzlů SST. V rámci jednoho uzlu SST není vždy několik kompletních úrovní binárního (pod)stromu, ale (jak již název napovídá) úseky různého tvaru (viz obr. 3.10). Pro popis tohoto tvaru je v rámci každého uzlu přítomna další bitmapa. Ta obsahuje dva bity pro každý vnitřní uzel uzlu SST. Bity udávají, zda existuje levý/pravý potomek tohoto vnitřního uzlu. V bitmapě jsou zahrnuty všechny vnitřní uzly v pořadí breadth-first. Pro budování SST bývá stanovena hranice maximálního počtu vnitřních uzlů, které může jeden uzel SST obsahovat (tuto hodnotu označíme „k“).



Obrázek 3.10: Trie sestavená algoritmem Shape Shifting Tree

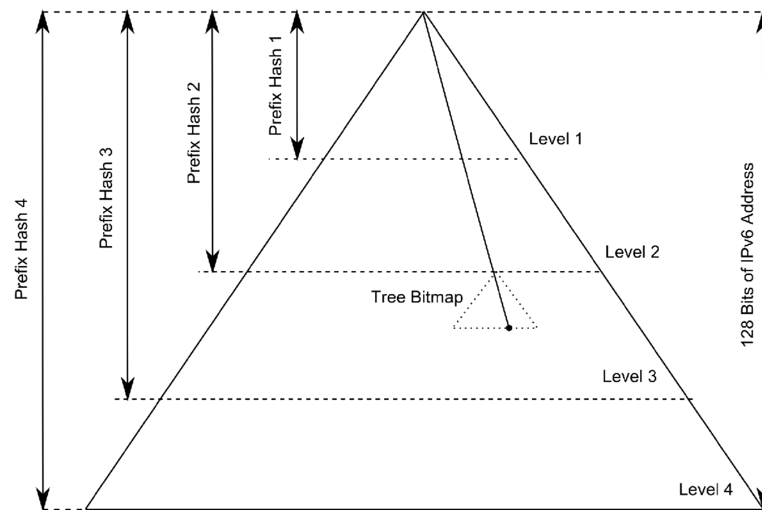
SST struktura se buduje z binárního stromu. Nejprve je nutné pro všechny uzly spočítat hodnotu p – počet uzlů, které obsahuje podstrom s tímto uzlem jako kořenem. Poté procházíme uzly v breadth-first pořadí, a pokud narazíme na uzel, který má hodnotu p menší nebo rovnu hranici k, odstraníme celý tento podstrom a místo něj vytvoříme uzel SST. Poté je nutné upravit hodnoty p u všech předků uzlů z tohoto podstromu a opět hledáme uzel

s hodnotou $p \leq k$. Algoritmus vyhledávání není triviální a je detailně popsán například v [9].

Každý uzel SST obsahuje celkem 3 bitmapy – ty reprezentují platnost prefixů ve vnitřních uzlech, existenci potomků uzlu a tvar binárního stromu uvnitř uzlu. Dále obsahuje (stejně jako uzel Tree Bitmap) ukazatel na první záznam v tabulce prefixů, který je platný pro tento uzel, a ukazatel na prvního potomka tohoto uzlu.

3.1.7 Hash Tree Bitmap

Algoritmus Hash Tree Bitmap [12] využívá principu algoritmu Tree Bitmap. Nevýhodou algoritmu Tree Bitmap je, že výška stromu je rovna délce nejdelšího prefixu v množině podělené konstantou (střídou Tree Bitmap) – vyhledávání je tedy u dlouhých prefixů poměrně pomalé. Prodloužení adresy ze 32 bitů na 128 bitů u IPv6 tedy při použití algoritmu Tree Bitmap znamená velké snížení rychlosti zpracování.



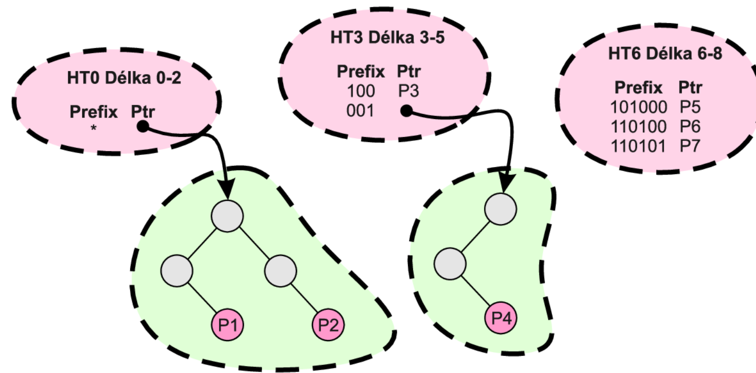
Obrázek 3.11: Princip algoritmu Hash Tree Bitmap

Hash Tree Bitmap tento nedostatek částečně odstraňuje tak, že velkou část trie přeskočí. Algoritmus se v první fázi pomocí hashovací funkce dostane co nejblíže listovému uzlu. V druhém kroku použije strukturu Tree Bitmap pouze na dohledání ve zbytku stromu, který je už výrazně nižší než celá trie.

Prefixy jsou rozděleny do několika skupin dle jejich délky (viz obrázek 3.11). Dále jsou oříznuty na spodní hranici délek prefixů ve své skupině. Takto oříznutý prefix označme jako „začátek“. Máme-li například skupinu prefixů délky 3 až 5 (viz HT3 na obr. 3.12), budou všechny prefixy této délky oříznuty na tři bity. Každý takto oříznutý prefix obsahuje i odkaz na kořenový uzel struktury Tree Bitmap, od kterého započne vyhledávání při shodě vstupní adresy s tímto začátkem. Část prefixu, kterou jsme v tomto kroku odřízli, bude reprezentována ve stromu Tree Bitmap.

Nyní sestavíme popsanou strukturu pro naši testovací množinu prefixů z obrázku 3.1. Zvolme, že chceme prefixy rozdělit do tří délkových kategorií. Nejkratší skupina bude obsahovat prefixy délky nula až dva, další délky tři až pět a poslední skupina délky šest až osm. Z prefixů P3 a P4, které padnou do druhé skupiny, tedy odřízneme první tři bity

a vytvoříme z nich dvě položky v hashovací tabulce délky 3-5 (označme ji HT3). Prefix P3 obsahuje pouze 3 bity (nic jsme neodřízli), takže najdeme-li tuto trojici bitů pomocí hashovací funkce ve vstupní adrese, nalezneme prefix v jediném kroku. Z prefixu P4 jsme ořízli dva bity (konkrétně 01) a tyto budou uloženy ve struktuře Tree Bitmap, na kterou bude odkazovat tato položka HT3. Pokud by v této délkové skupině byl ještě nějaký prefix začínající 001, uložili bychom ho do stejného Tree Bitmap stromu jako P4. Aplikujeme-li tento postup na všechny skupiny prefixů, dostaneme výslednou strukturu na obrázku 3.12.



Obrázek 3.12: Výsledná struktura Hash Tree Bitmap. Zeleně jsou označené Tree Bitmap uzly, hashovací tabulky jsou červené

Postup vyhledávání je zřejmý. Vstupní adresu 11010111 vložíme do hashovacích funkcí všech skupin prefixů. Platný záznam je nalezen v HT0 (prefix nulové délky) a v HT6 (prefix délky 6). Protože v HT6 je delší shoda, vybereme tuto. Nalezená položka obsahuje odkaz přímo na prefix P7, který je výsledkem. Při vyhledávání adresy 00101110 bychom nalezli shodu v HT0 a v HT3. V HT3 je delší shoda, proto bychom vybrali tuto. Nalezený záznam v HT3 obsahuje odkaz na uzel Tree Bitmap, ve kterém bude pokračovat vyhledávání počínaje čtvrtým bitem vstupní adresy. Algoritmem Tree Bitmap nalezneme prefix P4, který je výsledkem.

Při použití s IPv6 se délky klíčů v jednotlivých hashovacích tabulkách přizpůsobují typickým délkám prefixů – často jsou tedy voleny například tabulky s klíči délky 32, 48, 64 a 128 bitů. Díky tomu je pak mnoho prefixů objeveno v jediném kroku algoritmu (pouze pomocí hashovací funkce).

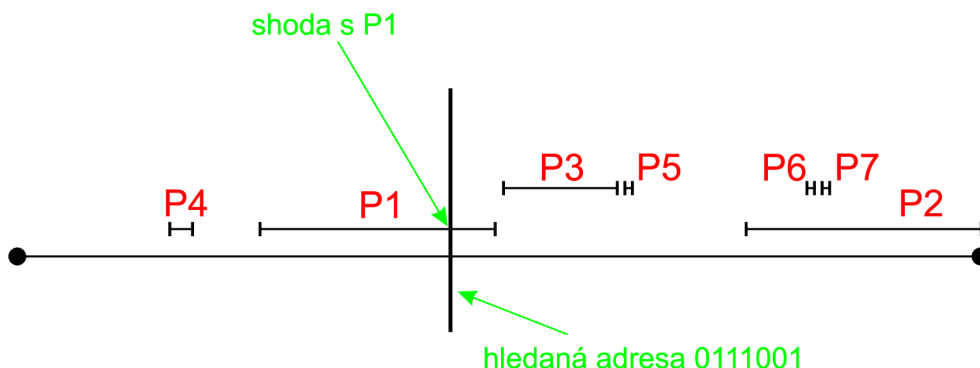
Algoritmus Hash Tree Bitmap oproti Tree Bitmap podstatně zrychluje výpočet LPM zejména u dlouhých IPv6 prefixů. Touto metodou je možné zpracovat mnoho bitů v jediném kroku pomocí hashovací funkce. Poměr mezi časovou a paměťovou složitostí lze upravovat změnou počtu skupin délek prefixů. Zvolíme-li více skupin, bude třeba více hashovacích tabulek, ale následné vyhledávání bude rychlejší, protože vzniklé Tree Bitmap stromy budou nižší. Nevýhodou algoritmu je nutná paralelizace výpočtu hashovacích funkcí, softwarová implementace algoritmu tedy není vhodná.

3.2 Ostatní algoritmy

3.2.1 Binární vyhledávání na intervalech

Algoritmus Binární vyhledávání na intervalech [6] je v této práci označovaný také zkratkou „BSI“ – z anglického „Binary Search on Intervals“. Na množinu vstupních prefixů pohlíží

jako na množinu intervalů adres. Protože jeden prefix může být prefixem jiného prefixu, mohou se tyto intervaly protínat. Pokud hledaný prefix leží v několika intervalech současně, algoritmus vybere nejmenší interval (ten odpovídá nejdelšímu prefixu).



Obrázek 3.13: Intervaly znázorněné na ose

Hranice intervalů získáme expandováním prefixů nulami (začátek intervalu) nebo jedničkami (konec intervalu) na počet bitů následně vyhledávaných adres. Takto vzniklé intervaly z testovací množiny prefixů jsou znázorněny na obrázku 3.13.

hledaná adresa 0111001

0000000	-
0010100	P4
0011000	-
0100000	P1
1000000	P3
1010000	P5
1010010	-
1100000	P2
1101000	P6
1101010	P7
1101100	P2

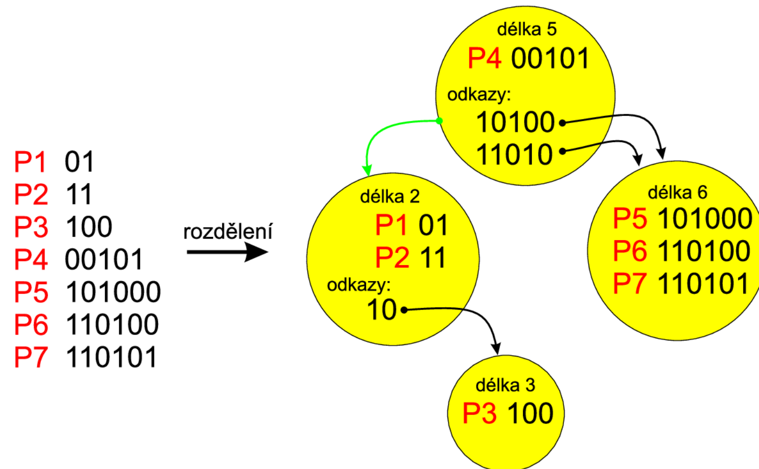
Obrázek 3.14: Počáteční adresy prefixů uložené v tabulce

Z těchto intervalů sestaví algoritmus tabulku – pro každý interval vstupních hodnot je dopředu určen prefix, který je v tomto intervalu nejdelším shodným. Tyto intervaly jsou seřazeny, abychom v nich mohli rychle vyhledávat za použití běžných vyhledávacích metod. Pokud zjistíme, že by hledaná adresa měla být mezi řádkem i a i+1, výsledkem je prefix z řádku i.

Sestavená tabulka je na obrázku 3.14, pro jednoduchost jsou tu použity 7bitové adresy. Na obou obrázcích je znázorněno vyhledání adresy 0111001.

3.2.2 Binární vyhledávání na prefixech

Binární vyhledávání na prefixech je LPM algoritmus využívající hashovací funkce, zástupce této metody byl představen v [13]. V této práci jej často označujeme zkráceně jako „BSP“ – z anglického názvu „Binary Search on Prefixes“. Zjednodušená sestavená struktura je na obrázku 3.15.



Obrázek 3.15: Zjednodušený princip algoritmu BSP

Hashování se zde uplatňuje pouze na prefixech shodné délky. Nejprve je tedy nutné vstupní množinu prefixů rozdělit do skupin podle jejich délky. Zároveň je třeba do každé skupiny zahrnout i patřičně zkrácené delší prefixy a tyto opatřit odkazy na skupiny, ze kterých byly prefixy vybrány. Skupiny seřadíme podle délky a vyhledávat začínáme na skupině se střední délkou, v našem případě na skupině prefixů s délkou 5. Pokud bychom hledali například adresu 11010100, vložili bychom do hashovací funkce tento řetězec zkrácený na délku 5, tj. 11010. Narazili bychom na odkaz směřující ke skupině s prefixy délky 6. Při vyhledávání v této skupině bychom do hashovací funkce vložili prvních 6 znaků řetězce, tzn. 110101, a narazili bychom na shodu s prefixem P7.

Pokud bychom vyhledávali například řetězec 10010100, začali bychom opět v uzlu s řetězci délky 5, ke shodě by však nedošlo. Proto bychom se podle odkazu (na obrázku 3.15 zeleně) přesunuli k prefixům délky 2. Zde bychom narazili na odkaz k prefixům délky 3, kde bychom správně nalezli prefix P3.

Opět existuje více podobných variant tohoto algoritmu, včetně optimalizací pro zmenšení paměťových nároků. Některé optimalizace lze nalézt v [13].

Přestože paměťové nároky algoritmu jsou zpravidla vyšší než u jiných LPM metod, vyhledávání samotné je poté poměrně rychlé.

Kapitola 4

Návrh algoritmu Generic Hash Tree Bitmap

Navržený algoritmus Generic Hash Tree Bitmap je optimalizací Hash Tree Bitmap využívající efektivní a generické použití hashovacích funkcí. Tento algoritmus umožňuje v rámci vyhledávání jedné adresy i několikrát vystřídat krok hashování s krokem Tree Bitmap.

Algoritmus Hash Tree Bitmap (viz kapitola 3.1.7) obsahoval několik hashovacích tabulek s klíči zadané délky. Nejprve rozdělil prefixy do několika skupin dle jejich délky. Poté do hashovací tabulky (pod klíč skládající se z počátečních bitů prefixu) uložil kořenový uzel Tree Bitmap stromu, do kterého uložil zbytek prefixu. Vyhledání probíhalo vždy tak, že v prvním kroku byl pomocí hashovací funkce nalezen Tree Bitmap strom, ve kterém proběhlo vyhledání zkráceného prefixu algoritmem Tree Bitmap.

Délky klíčů jednotlivých hashovacích tabulek u Hash Tree Bitmap bývají zvoleny dle histogramu délek typické množiny prefixů. Ten má lokální maxima na délkách 32, 48, 64 a 128 bitů. Proto je vhodné vložit do struktury hashovací tabulky s klíči těchto délek. V algoritmu Generic Hash Tree Bitmap se tyto hashovací tabulky objevují také a nazývají se zde inter-tree tabulky.

V reálných množinách ale existuje mnoho prefixů, které mají odlišnou délku (často o několik bitů menší než uvedené hodnoty) a tyto jsou poté vyhledávány v klasickém Tree Bitmap stromu, který stále může mít značnou výšku.

Algoritmus Generic Hash Tree Bitmap tyto snížené Tree Bitmap stromy dále snižuje (a tím urychluje vyhledávání) pomocí tzv. intra-tree hashovacích tabulek. Ty mohou (ale nemusí) být vložené v každém Tree Bitmap uzlu. Princip je shodný s inter-tree tabulkami – opět se snažíme část bitů vyhledávaného prefixu přeskočit pomocí hashovací funkce. Intra-tree tabulky však mají podstatně kratší klíče, protože typicky slouží k přeskakování částí Tree Bitmap stromů, které už jsou sníženy pomocí inter-tree tabulek. Pokud by například byly přítomny inter-tree tabulky s délkami klíčů 32, 48 a 64 bitů, byl by Tree Bitmap strom vysoký maximálně pouze 64 bitů (při střídě 4 to odpovídá výšce 16 uzlů). V rámci těchto 16 uzlů můžeme části přeskakovat pomocí intra-tree hashovacích tabulek.

Oproti algoritmu Hash Tree Bitmap obsahují uzly Generic Hash Tree Bitmap navíc datové položky spjaté s hashovacími tabulkami. Je to bitmapa přítomnosti intra-tree hashovacích tabulek různých délek, odkazy na intra-tree hashovací tabulky a dva bity uchovávající informaci, zda se v daném uzlu nachází hashovací tabulky, struktura Tree Bitmap uzlu, případně obojí.

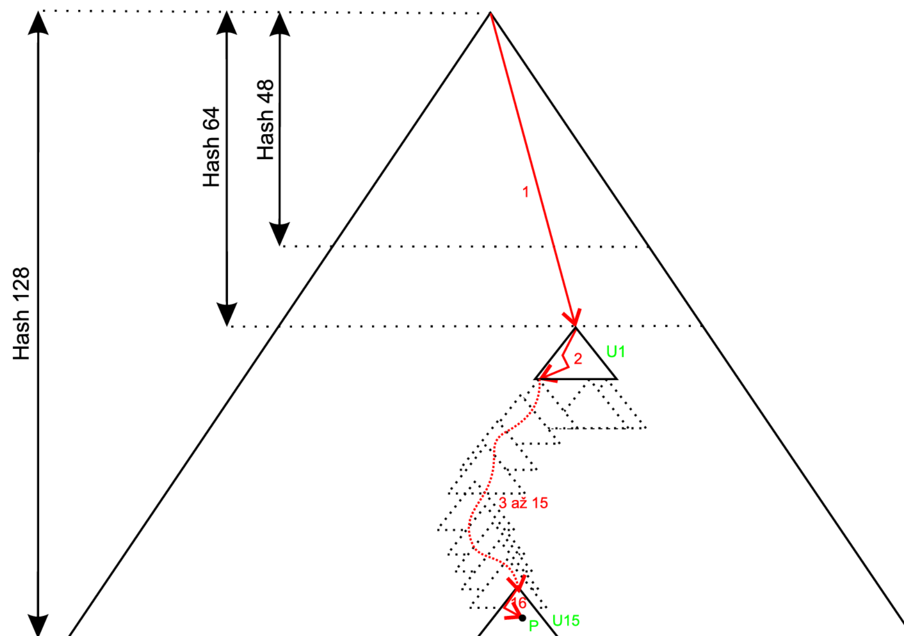
Navržené metody pro vkládání záznamů do intra-tree hashovacích tabulek popisují ná-

sledující kapitoly.

4.1 Odstranění nevětvených cest

4.1.1 Princip

Představme si, že máme vyhledat nejdelší shodný prefix k IP adrese $A1$ a že výsledný prefix má délku 126 bitů. Při použití algoritmu Tree Bitmap se střídou 4 bychom museli projít přes $\lceil 126/4 \rceil = 32$ uzlů. Pokud bychom použili algoritmus Hash Tree Bitmap s délkou hashovacích tabulek 32, 48, 64 a 128 bitů, přeskočili bychom prvních 64 bitů odkazem z inter-tree hashovací tabulky a zbylých $126 - 64 = 62$ bitů bychom museli projít přes $\lceil 62/4 \rceil = 16$ Tree Bitmap uzlů (viz obrázek 4.1).

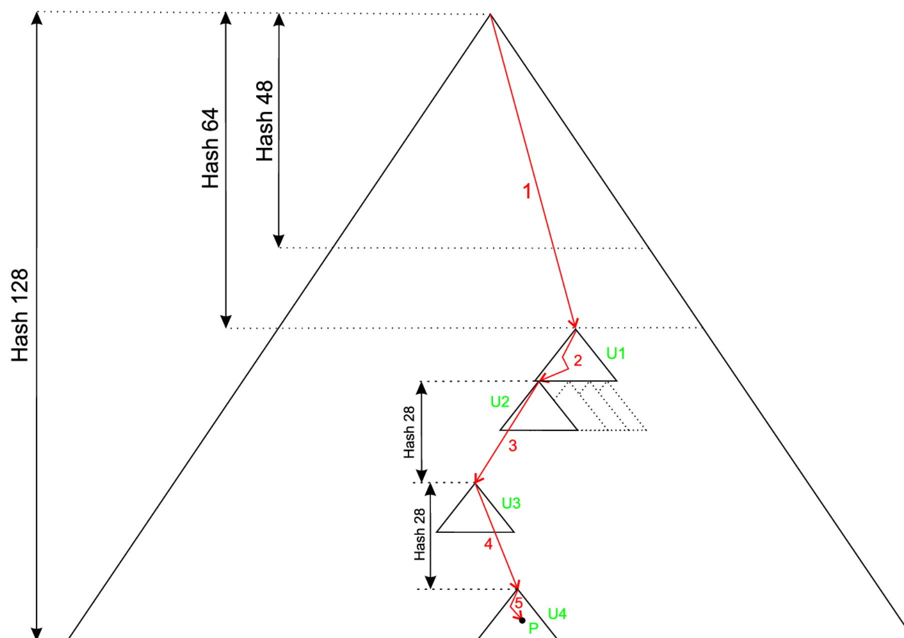


Obrázek 4.1: Princip Hash Tree Bitmap

Tuto vlastnost navržená metoda odstraňuje. Prostor IPv6 adres je opravdu široký (obsahuje asi 360 sextilionů adres). Díky tomu je stromová struktura Tree Bitmap typické množiny IPv6 prefixů poměrně řídká, a proto k mnohým prefixům vede v Tree Bitmap stromu dlouhá nevětvená cesta. Nabízí se proto vkládat do prvního uzlu nevětvené cesty inter-tree hashovací tabulku s odkazem na uzel na konci této cesty. Klíčem k takovému záznamu je ta část prefixu, která byla reprezentována cestou mezi prvním a posledním uzlem. Pro usnadnění HW implementace umožníme v každém uzlu vznik pouze několika hashovacích tabulek s pevně danými délkami klíčů.

Princip vyhledávání v této struktuře znázorňuje obrázek 4.2. Struktura v ilustrovaném příkladě obsahuje inter-tree hashovací tabulky délky 48, 64 a 128 bitů. Nejdelším prefixem k vyhledávané IP adrese je prefix P (dlouhý 126 bitů). Vyhledávání IP adresy začíná stejně jako u algoritmu Hash Tree Bitmap – nejprve jsou prohledány inter-tree hashovací tabulky a dle nejdelšího nalezeného záznamu (64 bitů) se dostáváme k uzlu U1. Tento uzel obsahuje více potomků (větvení), a proto intra-tree hashovací tabulky obsahovat nebude. Algoritmem

Tree Bitmap se dostaneme k uzlu U2. Z tohoto už vede nevětvená cesta délky 56 bitů k poslednímu uzlu, který v hloubce 2 bitů obsahuje hledaný prefix P. Při budování Generic Hash Tree Bitmap struktury bylo zadáno, aby byly použity intra-tree hashovací tabulky délky 28 bitů. Do uzlu U2 byl tedy nejprve vložen odkaz na uzel U3 a do tohoto následně odkaz na uzel U4. V uzlu U4 již nejsou hashovací tabulky a prefix je vyhledán algoritmem Tree Bitmap. Všimněme si, že bylo zapotřebí pouze 5 kroků, zatímco při použití Hash Tree Bitmap na obrázku 4.1 to bylo 16 kroků. V případě Tree Bitmap by to bylo 32 kroků. Navíc lze odstraněním přeskočených uzlů ušetřit paměť.



Obrázek 4.2: Princip algoritmu Generic Hash Tree Bitmap

4.1.2 Testování

Pro otestování byla vybrána reálná veřejně dostupná množina prefixů z autonomního systému AS6447 [2]. Ta v současné době obsahuje téměř 13000 prefixů, a je tak největší množinou, kterou knihovna Netbench obsahuje. Z této množiny byly sestaveny všechny potřebné struktury pro vyhledávání a vypočítána paměť potřebná pro reprezentaci množiny. Následně byl vyhledán nejdelší shodný prefix pro 4000 IPv6 adres. Při vyhledávání každé adresy byl počítán počet Tree Bitmap a Hash kroků nutných k nalezení prefixu. Test byl opakován s různým nastavením střídy a délek odstraňovaných nevětvených cest.

Data z tohoto testu zachycuje tabulka 4.1. Paramtetry Generic Hash Tree Bitmap jsou označeny následovně:

- `n` Střída Tree Bitmap uzlů
- `h` Délka klíčů (v bitech) v jednotlivých intra-tree hashovacích tabulkách

Druhý sloupec tabulky uvádí paměť potřebnou pro reprezentaci testovací množiny prefixů, třetí sloupec průměrný počet kroků pro vyhledání prefixu ke zvoleným IP adresám.

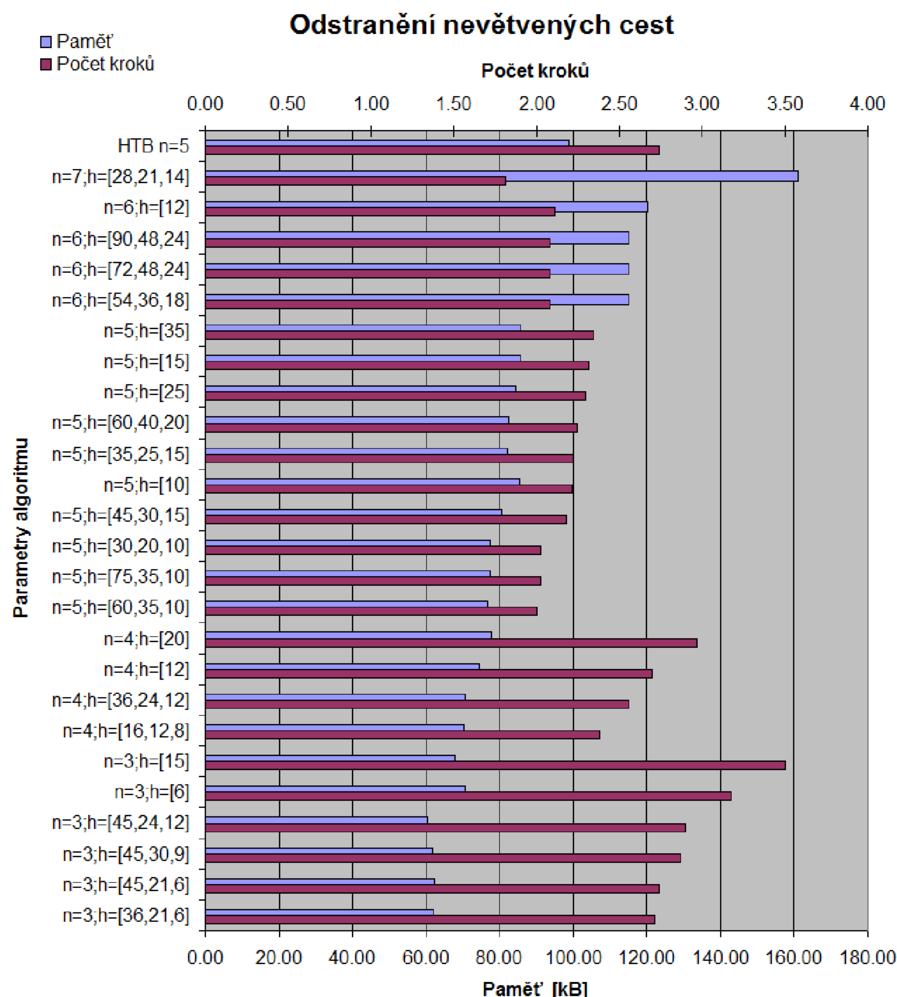
Čtvrtý a pátý sloupec ukazuje relativní spotřebu paměti a počet nutných kroků oproti algoritmu Hash Tree Bitmap se stejnou střídou. Poslední řádky tabulky obsahují pro porovnání výsledky algoritmu Hash Tree Bitmap (nulová délka intra-tree hashovacích tabulek).

	Výsledky GHTBM		**Oproti HTBM**	
Parametry algoritmu	**Paměť [kB]**	**Počet kroků**	**Paměť**	**Počet kroků**
n=3;h=[36,21,6]	62.11	2.72	79 %	62 %
n=3;h=[45,21,6]	62.43	2.74	80 %	63 %
n=3;h=[45,30,9]	61.61	2.87	79 %	66 %
n=3;h=[45,24,12]	60.53	2.90	77 %	67 %
n=3;h=[6]	70.67	3.18	90 %	73 %
n=3;h=[15]	67.81	3.50	86 %	81 %
n=4;h=[16,12,8]	70.31	2.38	79 %	68 %
n=4;h=[36,24,12]	70.73	2.56	79 %	72 %
n=4;h=[12]	74.41	2.70	83 %	76 %
n=4;h=[20]	77.83	2.97	87 %	84 %
n=5;h=[60,35,10]	76.60	2.00	77 %	73 %
n=5;h=[75,35,10]	77.40	2.02	78 %	74 %
n=5;h=[30,20,10]	77.29	2.03	78 %	74 %
n=5;h=[45,30,15]	80.42	2.18	81 %	80 %
n=5;h=[10]	85.31	2.21	86 %	81 %
n=5;h=[35,25,15]	82.07	2.22	83 %	81 %
n=5;h=[60,40,20]	82.39	2.24	83 %	82 %
n=5;h=[25]	84.35	2.29	85 %	84 %
n=5;h=[15]	85.78	2.32	87 %	85 %
n=5;h=[35]	85.77	2.34	87 %	86 %
n=6;h=[54,36,18]	115.18	2.08	80 %	82 %
n=6;h=[72,48,24]	115.24	2.08	80 %	82 %
n=6;h=[90,48,24]	115.24	2.08	80 %	82 %
n=6;h=[12]	120.27	2.11	83 %	83 %
n=7;h=[28,21,14]	161.09	1.81	74 %	78 %
HTB n=3	78.46	4.35		
HTB n=4	89.22	3.53		
HTB n=5	98.94	2.74		
HTB n=6	144.31	2.55		
HTB n=7	217.65	2.31		

Tabulka 4.1: Paměťové nároky a rychlost algoritmu po odstranění nevětvených cest

Data byla vynesena do grafu 4.3. Můžeme si všimnout, že s rostoucí střídou Tree Bitmap stoupá množství využité paměti. U střídy 3 bylo pro reprezentaci množiny potřeba 64 kB, u střídy 8 průměrně 250 kB, se střídou 9 dokonce 400 kB paměti. Zároveň ale s rostoucí střídou klesá počet nutných kroků pro vyhledání prefixu. Pokud bychom totiž u střídy 3 potřebovali prozkoumat 12 bitů prefixu algoritmem Tree Bitmap, potřebovali bychom na to čtyři kroky, zatímco u střídy 6 pouze dva kroky.

Pro další experimenty vybereme zástupce s vhodnými parametry (kompromis mezi paměťovými nároky a rychlostí), kterým by mohl být například $n = 5; h = [30, 20, 10]$. Generic



Obrázek 4.3: Paměťové nároky a rychlost algoritmu po odstranění nevětvených cest

Hash Tree Bitmap s těmito parametry vyžaduje pro testovací množinu 77 kB paměti a na vyhledání prefixů k testovaným adresám mu stačí průměrně dva kroky. Hash Tree Bitmap se stejnou střídou by vyžadoval 99 kB paměti a 2,7 kroků. Zlepšení je tedy na této množině v obou aspektech o cca 25 %.

4.2 Adaptivní expanze prefixů

4.2.1 Princip

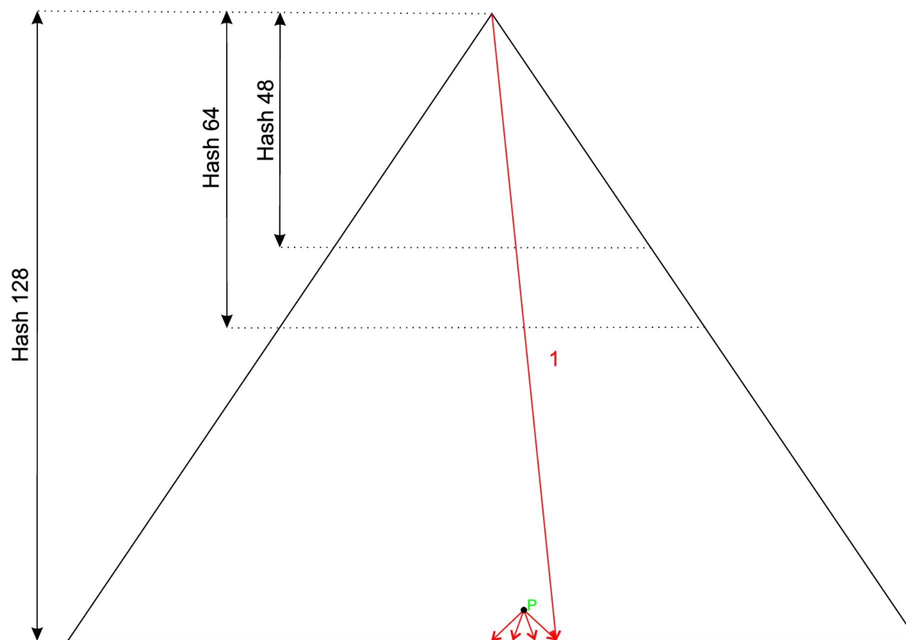
Další realizovanou optimalizací je adaptivní expanze prefixů. Představme si situaci z obrázku 4.2. Prefixy, které jsou jenom o několik bitů kratší, než je délka některé inter-tree hashovací tabulky, musí být vloženy do hashovací tabulky společně s prefixy kratších délek. Takové prefixy tvoří nejvíce uzlů, protože se vyskytují v nejhlubších částech Tree Bitmap stromu (odkazovaného z kratší hashovací tabulky), a tím zvětšují velikost struktury v paměti. Zároveň zpomalují vyhledávání, protože zvyšují výšku stromů.

Řešením je adaptivní expanze prefixů. Pokud při načítání množiny prefixů zjistíme,

že se zpracovávaný prefix kvůli několika chybějícím bitům nedostane do nějaké hashovací tabulky, vložíme do hashovací tabulky odkazy na tento prefix pod více klíči, které vzniknou expanzí původního záznamu. Samotná expanze je popsána v kapitole 3.1.2.

Potřebujeme-li doexpandovat n bitů, vznikne nám 2^n klíčů. Na těchto 2^n pozic v hashovací tabulce již neukládáme odkazy na uzly, ale přímo odkaz na platný prefix. Zároveň není zapotřebí uchovávat v paměti uzly, které by byly nutné při vložení prefixu do kratší hashovací tabulky.

Pokud bychom tedy měli v množině prefix P délky 126 bitů tak jako na obrázku 4.2, byl by doexpandován na 4 klíče délky 128 bitů. Na odpovídající pozice hashovací tabulky by byly vloženy odkazy na prefix P. Vyhledání by proběhlo v jednom jediném kroku tak jako na obrázku 4.4.



Obrázek 4.4: Princip adaptivní expanze prefixů

Stejně tak je možné expandovat prefixy v rámci jednotlivých snížených stromů při odstraňování nevětvených cest. Pokud bychom například povolili vkládání intra-tree hashovacích tabulek s klíči délky 16 bitů a byla by objevena nevětvená cesta k listu délky 15 bitů, je možné jeden bit doexpandovat a poté vložit do hashovací tabulky dva odkazy na expandovaný prefix.

4.2.2 Testování

Algoritmus Generic Hash Tree Bitmap s odstraňováním nevětvených cest s expanzí prefixů byl otestován na stejné množině prefixů při vyhledávání stejných adres jako v kapitole 4.1.2. Pro test byla zvolena střída 5 s délkami hashovacích klíčů v intra-tree tabulkách 30, 20 a 10 bitů. Pro toto nastavení byly otestovány různé délky expanze. V tabulce 4.2 jsou výsledky testů. Čtvrtý a pátý sloupec tabulky ukazují relativní spotřebu paměti a počet vyhledávacích kroků oproti algoritmu Hash Tree Bitmap se střídou 5 (ten by vyžadoval 99 kB paměti a 2,7 kroků). Parametry algoritmu jsou zde označeny následovně:

- `s` Největší povolená délka expanze, která umožní, aby se prefix dostal do delší intra-tree hashovací tabulky
- `l` Největší povolená délka expanze, která umožní, aby se prefix dostal do delší inter-tree hashovací tabulky

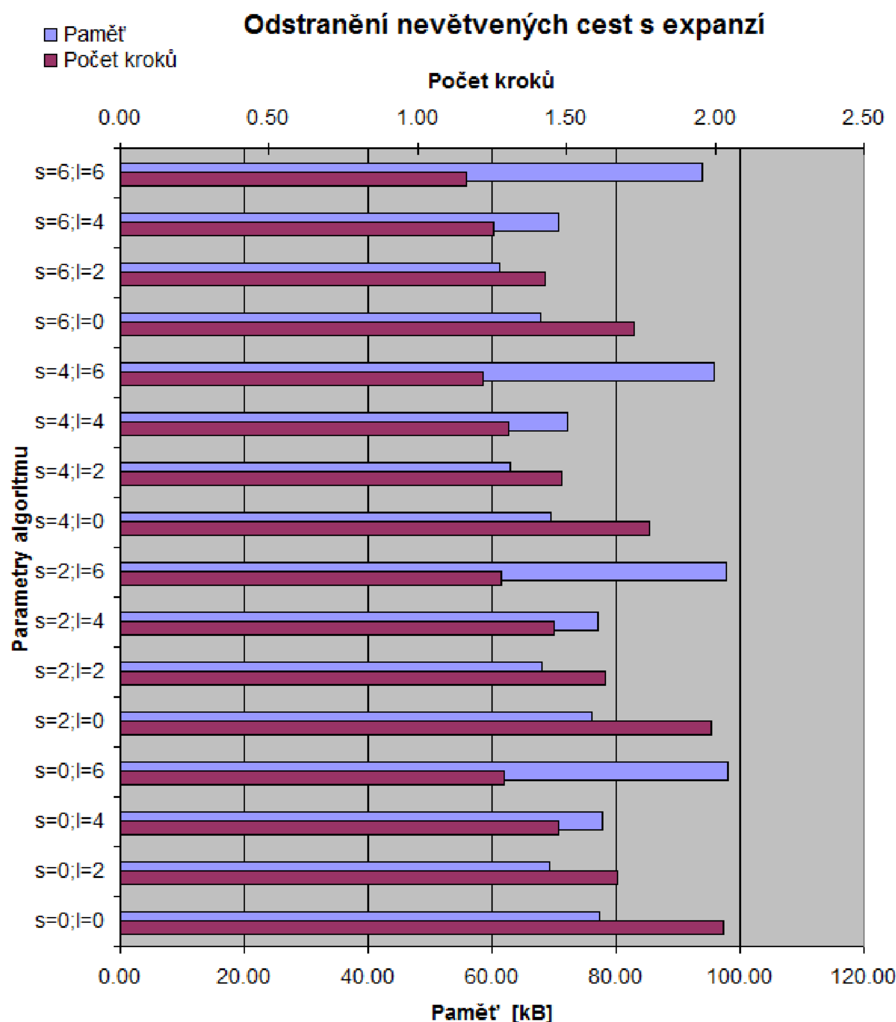
	Výsledky GHTBM		**Oproti HTBM**	
Parametry algoritmu	**Paměť [kB]**	**Počet kroků**	**Paměť**	**Počet kroků**
s=0;l=0	77.29	2.03	78 %	74 %
s=0;l=2	69.20	1.67	70 %	61 %
s=0;l=4	77.90	1.47	79 %	54 %
s=0;l=6	98.08	1.29	99 %	47 %
s=2;l=0	76.06	1.99	77 %	73 %
s=2;l=2	67.97	1.63	69 %	60 %
s=2;l=4	77.15	1.46	78 %	53 %
s=2;l=6	97.70	1.28	99 %	47 %
s=4;l=0	69.49	1.78	70 %	65 %
s=4;l=2	62.84	1.48	64 %	54 %
s=4;l=4	72.31	1.31	73 %	48 %
s=4;l=6	95.79	1.22	97 %	45 %
s=6;l=0	67.85	1.73	69 %	63 %
s=6;l=2	61.16	1.43	62 %	52 %
s=6;l=4	70.65	1.25	71 %	46 %
s=6;l=6	94.00	1.17	95 %	43 %

Tabulka 4.2: Paměťové nároky a rychlost algoritmu po odstranění nevětvených cest s expanzí

Data z tabulky 4.2 byla vynesena do grafu na obrázku 4.5.

Na grafu 4.5 v posledním řádku vidíme původní výsledky algoritmu s nulovou expanzí. Vyžadoval 77 kB paměti a průměrně 2 kroky. Vidíme, že při expandování maximálně dvou bitů potřebná paměť klesne. To je dáno tím, že po expanzi je možné odstranit nepotřebné uzly. Pokud umožníme delší expanzi (4 bity), paměťové nároky zase stoupnou zhruba na stejnou úroveň, protože každý prefix ukládáme do hashovací tabulky pod 16 klíči. U delších expanzí už paměť roste, přestože počet kroků nutný pro vyhledání dle očekávání dále klesá.

Dle grafu 4.5 můžeme za vhodného zástupce s kompromisem mezi paměťovými nároky a rychlostí prohlásit řádek s parametry $s = 6; l = 4$. Tato expanze snížila paměťové nároky ze 77 kB na 70 kB a zároveň snížila průměrný počet kroků při vyhledávání z 2,0 na 1,3. Je to dáno tím, že většina prefixů má délku shodnou s délkami klíčů inter-tree hashovacích tabulek. Ostatní prefixy jsou většinou blízko těchto délek – pokud jsou o několik bitů delší, po vyhledání v inter-tree hash tabulkách následuje vyhledání v jednom až dvou Tree Bitmap uzlech. Pokud je ovšem prefix o několik bitů kratší, než některá intra-tree hashovací tabulka, musí být uložen v kratší tabulce a cesta k tomuto uzlu je dlouhá. V kapitole 4.1 jsme tuto cestu odstraňovali pouze odstraněním dlouhých nevětvených cest. Pakliže ale chybí pouze několik bitů, je mnohem efektivnější tyto bity doexpandovat, odstranit nepotřebné uzly a vyhledání proběhne v jediném kroku.



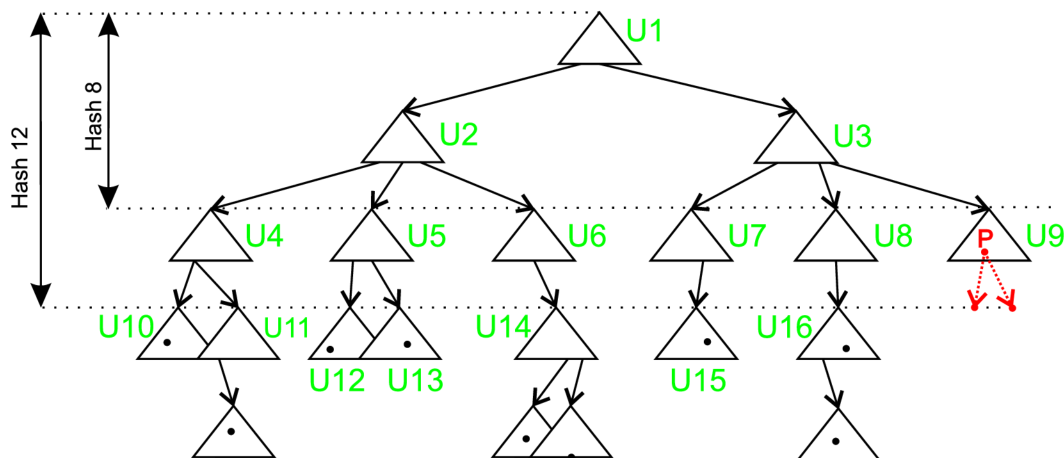
Obrázek 4.5: Paměťové nároky a rychlost algoritmu po odstranění nevětvených cest s expanzí

4.3 Odstranění větvení bez prefixů

4.3.1 Princip

Odstranění nevětvených cest samozřejmě není jediný způsob využití hashování uvnitř snížených stromů. Ještě zajímavější myšlenkou je odstraňovat i takové části stromu, které sice jsou větvené, ale všechny větve až do jisté hloubky neobsahují prefixy.

Příklad je na obrázku 4.6. Je použita střída 4, uzel U1 může obsahovat intra-tree hashovací tabulky s klíči délky 8 nebo 12 bitů. Zkoumáme prefixy ležící v uzlech, které jsou potomky uzlu U1. Nejkratší prefix P leží v uzlu U9. Ostatní prefixy leží v uzlech U10 až U16 nebo ještě hlouběji. První možností, jak odstranit větvení bez prefixů, je odstranit uzly U2 a U3 a do intra-tree tabulky délky 8 bitů uzlu U1 vložit odkazy na uzly U4 až U9 s patřičnými klíči. Odstranili bychom tedy pouze 2 uzly a do hashovací tabulky bychom vložili 6 odkazů. Vyhledání by proběhlo rychleji (místo procházení dvou uzlů metodou Tree Bitmap bychom se na stejnou hloubku dostali jedním odkazem v hashovací tabulce).



Obrázek 4.6: Princip odstranění větvení bez prefixů

I zde se ale dá použít expandování prefixů vysvětlené v kapitole 4.2.1. Pokud by například k prefixu uloženému v uzlu U9 stačilo doexpandovat jeden bit (na obrázku 4.6 označeno červeně), aby se dostal v uzlu U1 do delší hashovací tabulky (délky 12 bitů), byla by situace podstatně lepší. Mohli bychom odstranit 8 uzlů (U2 až U9) a do hashovací tabulky uzlu U1 bychom vložili odkaz na uzly U10 až U16 a dále dva odkazy na prefix P. Úspora paměti by byla větší a vyhledávání by proběhlo ještě rychleji (jeden odkaz přes hashovací tabulku oproti procházení tří Tree Bitmap uzlů).

4.3.2 Testování

Při testování odstraňování větvení bez prefixů byla použita expanze. Testy byly provedeny na stejné množině prefixů a IP adres jako v kapitole 4.1.2. Opět byla zvolena střída 5. Byly otestovány různé délky klíčů intra-tree hashovacích tabulek a různé povolené délky expanze.

V tabulce 4.7 jsou výsledky testů. V posledních sloupcích je opět relativní střeba paměti a nutný počet kroků oproti algoritmu Hash Tree Bitmap. Parametry algoritmu jsou zde označeny následovně:

- `h` Délka klíčů (v bitech) v jednotlivých intra-tree hashovacích tabulkách
- `s` Největší povolená délka expanze, která umožní, aby se prefix dostal do delší intra-tree hashovací tabulky
- `l` Největší povolená délka expanze, která umožní, aby se prefix dostal do delší inter-tree hashovací tabulky

Data z tabulky 4.7 byla vynesena do grafu 4.7. Výsledky se při použití vyobrazených parametrů algoritmu příliš neliší. Například při zvolení parametrů $h = [30, 20, 10]; s = 2; l = 2$ je spotřebováno 66,2 kB paměti a pro vyhledání prefixu je zapotřebí provést průměrně 1,49 kroků. Srovnejme to s výsledky z kapitoly 4.2.2 z testování odstraňování nevětvených cest. Tam při použití stejných parametrů $h = [30, 20, 10]; s = 2; l = 2$ bylo obsazeno 68 kB paměti a vyhledání proběhlo průměrně v 1,7 krocích.

Tato metoda tedy přináší další zlepšení výkonnosti algoritmu, přestože změna není příliš výrazná. To je dáno především tím, že struktura algoritmu Hash Tree Bitmap neobsahuje

	Výsledky GHTBM		Oproti HTBM	
Parametry algoritmu	**Paměť [kB]**	**Počet kroků**	**Paměť**	**Počet kroků**
[30,20,10]; s=2; l=2	66.23	1.49	67 %	54 %
[30,20,10]; s=2; l=4	75.35	1.37	76 %	50 %
[30,20,10]; s=4; l=2	67.60	1.49	68 %	54 %
[30,20,10]; s=4; l=4	76.68	1.37	77 %	50 %
[50,25,10]; s=2; l=2	65.35	1.47	66 %	54 %
[50,25,10]; s=2; l=4	74.47	1.35	75 %	49 %
[50,25,10]; s=4; l=2	66.73	1.47	67 %	54 %
[50,25,10]; s=4; l=4	75.80	1.35	77 %	49 %
[60,40,20]; s=2; l=2	66.16	1.62	67 %	59 %
[60,40,20]; s=2; l=4	73.62	1.38	74 %	50 %
[60,40,20]; s=4; l=2	66.13	1.62	67 %	59 %
[60,40,20]; s=4; l=4	73.60	1.38	74 %	50 %
[35,25,10]; s=2; l=2	66.56	1.49	67 %	54 %
[35,25,10]; s=2; l=4	75.68	1.37	76 %	50 %
[35,25,10]; s=4; l=2	67.93	1.49	69 %	54 %
[35,25,10]; s=4; l=4	77.01	1.37	78 %	50 %

Tabulka 4.3: Paměťové nároky a rychlost algoritmu po odstranění větvení bez prefixů s expanzí

příliš mnoho rozvětvených uzlů. Větvení je velice často nahrazeno mnoha záznamy v inter-tree hashovacích tabulkách. Z uzlu uloženého v hashovací tabulce pak již vede zpravidla nevětvená cesta až k jedinému prefixu na této cestě.

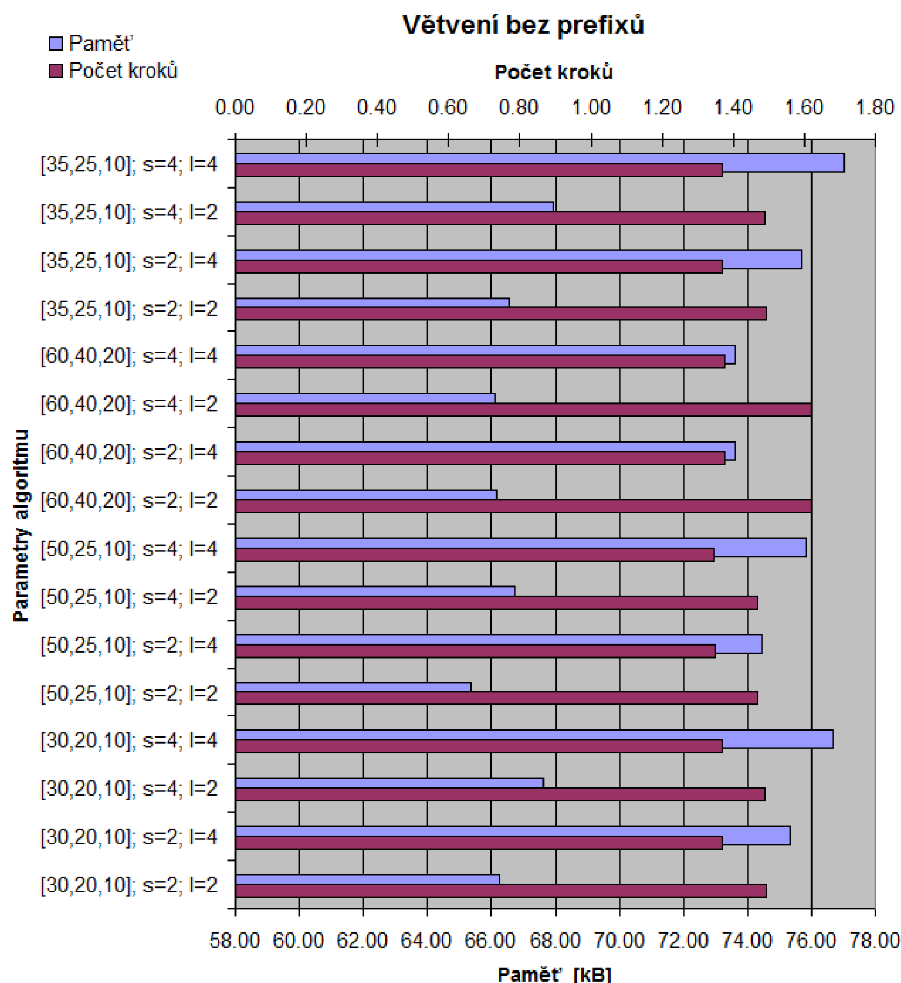
Jako příklad uveďme testovací množinu prefixů z autonomního systému AS6447, která obsahuje 12765 záznamů. Po sestavení Hash Tree Bitmap struktury vznikne celkem 10319 odkazů na uzly v inter-tree hashovacích tabulkách. Z toho vyplývá, že z každého uzlu uloženého v hashovací tabulce vede cesta průměrně pouze k 1,24 prefixům. Při odstranění větvení bez prefixů tedy nepřeskočíme o mnoho více uzlů než při zpracovávání nevětvených cest.

Se zvětšujícím se počtem prefixů bude pravděpodobně tato metoda mírně efektivnější, protože větvení se v takovém případě může objevovat častěji. Na generované množině obsahující 170000 prefixů (viz kapitola 6.1.2) bylo v intra-tree hashovacích tabulkách 119672 záznamů, což odpovídá průměrně 1,42 prefixům na záznam. Ani u velkých množin tedy nelze předpokládat zásadní změnu.

4.4 Steinerovy stromy

O tom, kdy použít hashovací funkci a kdy krok dle algoritmu Tree Bitmap, lze rozhodovat i za pomoci Steinerových minimálních stromů. Ty jsou pojmenovány podle švýcarského matematika Jakoba Steinera. Problém je popsán například v sedmé kapitole knihy [14].

Mějme ohodnocený neorientovaný graf $G = (V, E, w)$, kde V je množina vrcholů, E je množina hran a w je nezáporná funkce přiřazující ohodnocení hranám z množiny E. Dále mějme množinu $L \subset V$ terminálních vrcholů. Vrcholy z množiny V, které nepatří do



Obrázek 4.7: Paměťové nároky a rychlost algoritmu po odstranění větvení bez prefixů s expanzí

podmnožiny L, se nazývají neterminální. Steinerův minimální strom je graf $T \subset G$, který obsahuje všechny terminální uzly a zároveň má minimální součet ohodnocení svých hran. Graf T může obsahovat i neterminální uzly. Protože funkce w je nezáporná, výsledný graf T je stromem.

Ukažme nyní, jak by bylo možné využít vlastností těchto stromů v navrženém algoritmu Generic Hash Tree Bitmap. Struktura Tree Bitmap je orientovaným stromem. Množina vrcholů Tree Bitmap by tedy mohla tvořit množinu vrcholů V jakožto vstup algoritmu pro výpočet Steinerova minimálního stromu. Jako terminální uzly označme všechny Tree Bitmap uzly, které obsahují platný prefix. Transportní uzly (bez prefixu) budou tvořit množinu neterminálních uzlů.

Dále je potřeba vytvořit množinu hran. Steinerovy stromy není problém popsat i na orientovaných grafech. Množinu hran mezi Tree bitmap uzly (orientovaných směrem k listům) lze přímo vložit do množiny hran E. Předpokládejme nyní, že chceme vytvořit strukturu Generic Hash Tree Bitmap s minimálními paměťovými nároky. Pak všechny tyto hrany ohodnotíme množstvím paměti, které zabere jeden Tree bitmap krok (velikost Tree Bitmap

uzlu).

Pokud bychom například chtěli umožnit vznik intra-tree hashovací tabulky s klíči délky 20 bitů a použít Tree Bitmap se střídou 5 bitů, vložíme do množiny E ještě hrany spojující uzly, které v Tree Bitmap struktuře mají vzdálenost $20/5 = 4$. Ohodnocením takové hrany bude množství paměti potřebné pro záznam v hashovací tabulce. Obdobně bychom postupovali pro případné další intra-tree tabulky. Funkci w lze samozřejmě definovat i tak, aby odrážela kromě paměťových nároků i následnou rychlost vyhledávání.

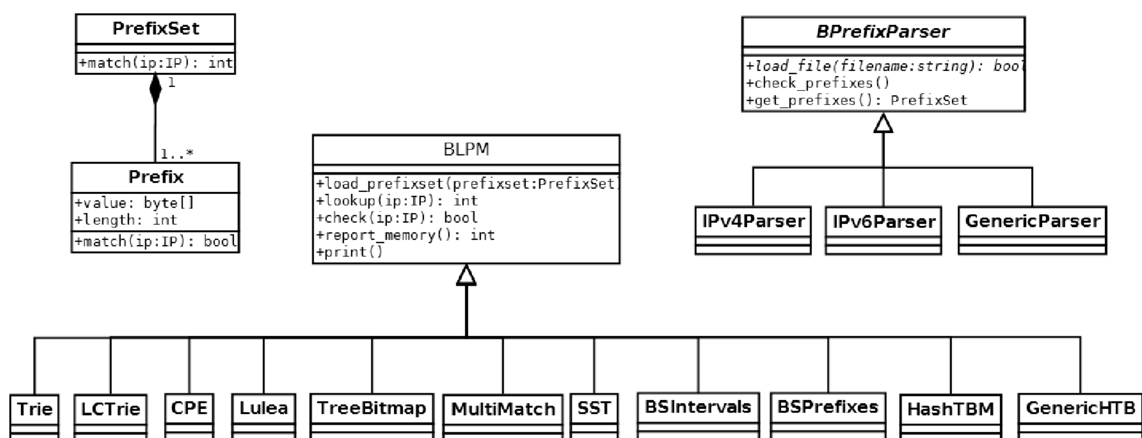
Nad takto vytvořeným grafem poté spustíme algoritmus pro nalezení Steinerova minimálního stromu, což je NP-těžký problém. Existuje však mnoho aproximací s nižší složitostí.

Kapitola 5

Implementace

5.1 Netbench

Netbench je knihovna postavená na jazyku Python, kterou vyvíjí výzkumná skupina ANT na FIT VUT v Brně [1]. ANT (Accelerated Network Technologies) se zabývá výzkumem především v oblasti monitorování počítačových sítí, jejich bezpečnosti a rychlosti.



Obrázek 5.1: Objektový model LPM části Netbench

Netbench obsahuje mnoho modulů pro různá odvětví. Pro tuto práci je podstatná část LPM, jejíž objektový model je na obrázku 5.1. Skládá se z bázové třídy `BLPM`, ze které dědí třídy implementující jednotlivé LPM algoritmy. Všechny algoritmy musí obsahovat metody:

- `load_prefixset`, která z objektu třídy `PrefixSet` vytvoří tímto LPM algoritmem všechny potřebné struktury
- `lookup`, která ve vytvořené struktuře vyhledá LPM pro zadanou IP adresu
- `check` zkontroluje, zda ve vytvořené struktuře pro zadanou IP adresu existuje nějaké prefix
- `report_memory` vrátí seznam obsahující informace o paměti potřebné k reprezentaci struktury

- `print` vytiskne náhled vytvořené struktury na standardní výstup

Třída `BPrefixParser` je bázovou třídou pro třídy IPv4Parser a IPv6Parser. Ty zajišťují načtení prefixů ze vstupního souboru do objektu třídy `PrefixSet`. Ten může obsahovat mnoho objektů třídy `Prefix`, které reprezentují jednotlivé načtené prefixy.

V rámci této práce byla implementována třída `GenericHTB`, její detailnější popis je uveden v následující kapitole 5.2.

5.2 Generic Hash Tree Bitmap

Třída `GenericHTB` přidává do knihovny implementaci algoritmu Generic Hash Tree Bitmap. Konstruktor této třídy je možné volat s těmito parametry::

- `stride` střída Tree Bitmap uzlů
- `inter-tree-len` seznam s délkami klíčů inter-tree hashovacích tabulek
- `intra-tree-len` seznam s délkami klíčů intra-tree hashovacích tabulek
- `method` metoda použitá pro vkládání inter-tree hashovacích tabulek – buď `unbranched` nebo `embranchment`
- `inter-tree-expand` maximální délka inter-tree expanze
- `intra-tree-expand` maximální délka intra-tree expanze

Metoda `load_prefixset` vyžaduje jediný parametr, objekt třídy `PrefixSet`. Z těchto vstupních prefixů sestaví strukturu Generic Hash Tree Bitmap – prefixy vloží do příslušných Tree Bitmap stromů (objektů třídy `Tree`), jejichž kořeny uloží do inter-tree hashovacích tabulek. Je-li to požadováno, zkusí tyto prefixy nejprve doexpandovat do delších tabulek.

Zároveň je nutné do kořenu každého vzniklého Tree Bitmap stromu předpočítat tzv. `defaultLPM`, tedy prefix, který je nejdelším prefixem prefixu reprezentovaného kořenem tohoto stromu. Pokud totiž při vyhledávání v prvním kroku najdeme záznam v inter-tree hashovací tabulce a skočíme do příslušné části, není jisté, že zde prefix opravdu najdeme. Pokud bychom ho nenašli, je nutné najít prefix ve stromech uložených v kratších hashovacích tabulkách. Uložením prefixu přímo do kořene dosáhneme toho, že při nenalezení prefixu v daném stromu nám k nalezení stačí už jenom jediný krok navíc.

Dále je možné ještě aplikovat nějakou metodu vkládání intra-tree hashovacích záznamů. Je tedy volána buď metoda `find_and_del_unbranched` pro odstranění nevětvených cest, nebo metoda `find_and_del_embranchment` pro odstranění větvení bez prefixů.

Pro vyhledání prefixů pro danou IP adresu se použije metoda `lookup` třídy `GenericHTB`. Jediným povinným parametrem je zkoumaná IP adresa. Metoda zkusí nalézt záznam v inter-tree hashovacích tabulkách – najde-li záznam, zavolá metodu `tree_lookup` pro vyhledání v tomto stromě.

Metoda `tree_lookup` pracuje s jednotlivými uzly. V každém uzlu nejprve zkusí nalézt záznam v intra-tree hashovacích tabulkách a případně se takto přesunout k dalšímu uzlu. Pokud záznam není nalezen, použije se krok Tree Bitmap. Postupně si ukládá aktuální nejdelší nalezený prefix. Vyhledávání končí, jakmile metoda narazí na listový uzel.

Metody `find_and_del_unbranched` a `find_and_del_embranchment` procházejí uzly v pořadí preorder, provádějí případnou expanzi prefixů, vkládají záznamy do intra-tree hashovacích tabulek a odstraňují uzly, které ve výsledné struktuře nejsou nutné.

Kapitola 6

Testování algoritmů

V této kapitole jsou jednotlivé algoritmy testovány na několika množinách IPv6 prefixů. Každý algoritmus nejprve z množiny sestavil svoje specifické struktury pro vyhledávání. U každé množiny byla vypočítána spotřeba paměti pro její reprezentaci touto strukturou. Následně byly každým algoritmem vyhledány nejdelší shodné prefixy k testovací množině IPv6 adres. Při vyhledávání byl počítán počet kroků, které bylo nutné vykonat při vyhledávání.

6.1 Testovací množiny

6.1.1 Reálné

Pro testy byly použity 3 reálné množiny IPv6 prefixů. První (a největší) z nich je množina prefixů z autonomního systému AS6447 [2]. Ta obsahuje téměř 13 000 záznamů. Menší množina „SIXXS-ALL" byla získána z [3] a obsahuje více než 6 000 prefixů. Poslední množina „RIPE" pochází ze stejného zdroje a poskytuje necelých 3 500 prefixů.

6.1.2 Generované

Protože reálné množiny jsou stále velice malé (přestože poměrně rychle rostou), byly pro další testování vygenerovány větší množiny (v testech označovaná jako „GEN80", „GEN120" a „GEN170"), které obsahují 80 000, 120 000 a 170 000 IPv6 prefixů. Byl použit generátor V6Gene [15] vyvíjený na Tsinghua University v Číně.

6.2 Jednotlivé algoritmy

6.2.1 Trie

Každý uzel Trie obsahuje:

- dva ukazatele na další uzly
- jeden ukazatel do tabulky prefixů

Při použití 32bitového ukazatele jeden uzel vyžaduje 96 bitů paměti. Maximální počet náhodných přístupů do paměti při vyhledávání je dán hloubkou stromu. Hloubku stromu určuje nejdelší prefix, který je tímto stromem reprezentován. Pro IPv6 prefixy tedy může

algoritmus vyžadovat až 128 náhodných přístupů, což je ve většině aplikací neakceptovatelné.

Množina	Paměť [kB]	Počet kroků
AS6447	638.74	48.59
ALL	246.67	41.55
RIPE	108.95	41.07
GEN80	11 824.32	47.52
GEN120	11 548.90	51.78
GEN170	16 964.51	59.97

Tabulka 6.1: Paměťové nároky a rychlost algoritmu Trie

Data z testů algoritmu Trie ukazuje tabulka 6.1. V této tabulce jsou paměťové nároky počítané s předpokladem použití nejmenšího možného ukazatele. Při použití 32bitových hodnot by paměťové nároky stouply o cca 60 %. Všimněme si, že bylo vyžadováno průměrně 44 kroků. To je dáno tím, že naprostá většina vyhledaných prefixů měla délku kolem 48 bitů. Paměťové nároky na reprezentaci generovaných množin jsou již značné.

6.2.2 Controlled Prefix Expansion

Metoda CPE (zde s optimalizací „leaf pushing“) ukládá do každého uzlu:

- 2^n ukazatelů,

kde n je počet bitů zpracovávaných v jednom taktu. Pro 5 bitů v jednom taktu tedy uzel obsahuje 32 ukazatelů. Jedná se buď o ukazatele na další uzel struktury, nebo o ukazatele do množiny prefixů.

Tabulka 6.2 shrnuje výsledky testů při použití nejmenšího možného ukazatele. Můžeme si opět všimnout, že 14 nutných kroků při střídě 3 odpovídá průměrné délce nejdelšího nalezeného prefixu asi 42 bitů, což je očekávaný výsledek.

Pokud zvyšujeme parametr n, snižuje se výška stromu (tím se teoreticky zrychluje vyhledávání), paměť potřebná k uložení uzlu však roste exponenciálně. Při parametru $n = 4$ jsou paměťové nároky srovnatelné s algoritmem Trie.

6.2.3 Lulea Compressed Trie

Tento algoritmus redukuje paměťové nároky metody CPE zavedením bitmapy a odstraněním redundantních ukazatelů.

Každý uzel obsahuje:

- Bitmapu o velikosti 2^n bitů, kde n je počet bitů zpracovávaných v jednom taktu
- Několik (maximálně 2^n) ukazatelů na další uzly nebo do tabulky prefixů

Uzel může obsahovat maximálně 2^n ukazatelů, v praxi jich však bývá mnohem méně.

Výsledky testování zobrazuje tabulka 6.3. Srovnáním s výsledky metody CPE v tabulce 6.2 vidíme, že (dle očekávání) vyžadují naprosto shodný počet kroků. Algoritmus Lulea Compressed Trie přináší výraznou úsporu paměti.

Množina	Parametry	Paměť [kB]	Počet kroků
AS6447	n=3	252.29	16.48
	n=5	483.58	10.13
	n=6	1 463.69	8.43
ALL	n=3	55.58	14.07
	n=5	85.08	8.60
	n=6	261.73	7.33
RIPE	n=3	26.98	14.08
	n=5	42.19	8.60
	n=6	136.34	7.29
GEN80	n=3	4 600.73	17.18
	n=5	20 661.47	10.37
	n=6	35 377.88	8.88
GEN120	n=3	4 266.69	17.64
	n=5	19 903.79	10.69
	n=6	36 226.55	9.04
GEN170	n=3	10 218.99	20.39
	n=5	29 115.02	12.26
	n=6	52 816.56	10.39

Tabulka 6.2: Paměťové nároky a rychlost algoritmu CPE

6.2.4 LC Trie

Dle kapitoly 3.1.4 každý uzel takto sestavené trie obsahuje:

- Hodnotu branch
- Hodnotu skip
- Ukazatel
  - na prefix, jedná-li se o listový uzel
  - na prvního potomka tohoto uzlu, pokud uzel není listový

Pokud bychom na hodnoty branch, skip i pro ukazatel použili 32bitové číslo, dostali bychom se na hodnotu 96 bitů pro uložení každého uzlu. Při představení algoritmu v [7] autoři uvádějí verzi, kdy jeden uzel trie vyžaduje pouze 32 bitů. Pro hodnotu branch je zde použito 5 bitů, 7 bitů pro hodnotu skip a 20 bitů pro uložení ukazatele. Protože počet potomků, které každý uzel má, je vždy mocnina dvou, může jich takto mít každý uzel až 2^31, což je jistě dostačující. Hodnotu skip lze uložit v rozsahu 0–127, což je opět dostatečné i pro IPv6.

Tabulka 6.4 prezentuje výsledky testů, opět s použitím nejmenšího možného ukazatele. Můžeme si všimnout, že oproti Trie zde použité metody komprese mnohonásobně snížily paměťové nároky i nutný počet kroků k vyhledání prefixů.

Maximální počet náhodných přístupů do paměti je i u tohoto algoritmu dán maximální hloubkou stromu. Ta je však výrazně nižší než u algoritmu Trie.

Množina	Parametry	Paměť [kB]	Počet kroků
AS6447	n=3	144.58	16.48
	n=5	157.02	10.13
	n=6	240.74	8.43
ALL	n=3	46.56	14.07
	n=5	58.44	8.60
	n=6	90.55	7.33
RIPE	n=3	21.32	14.91
	n=5	27.74	9.03
	n=6	42.65	7.78
GEN80	n=3	2 690.40	17.18
	n=5	3 382.75	10.37
	n=6	3 998.48	8.88
GEN120	n=3	2 538.58	17.64
	n=5	3 368.61	10.69
	n=6	4 093.38	9.04
GEN170	n=3	4 684.93	20.39
	n=5	4 787.89	12.26
	n=6	5 728.73	10.39

Tabulka 6.3: Paměťové nároky a rychlost algoritmu Lulea Compressed Trie

Množina	Paměť [kB]	Počet kroků
AS6447	57.85	9.25
ALL	25.86	9.46
RIPE	13.05	8.29
GEN80	423.05	5.15
GEN120	686.56	6.47
GEN170	900.72	10.27

Tabulka 6.4: Paměťové nároky a rychlost algoritmu LC Trie

6.2.5 Tree Bitmap

Každý uzel musí obsahovat:

- Bitmapu vnitřních uzlů, velikost minimálně $2^n - 1$ bitů
- Bitmapu potomků, velikost minimálně 2^n bitů
- Ukazatel do tabulky prefixů
- Ukazatel na prvního potomka

Číslo n je opět počet zpracovávaných bitů v jednom taktu.

Výsledky v tabulce 6.5 ukazují, že zvětšením parametru n se sníží výška stromové struktury a tím se sníží počet kroků nutných k nalezení prefixů. Na druhou stranu paměť potřebná pro reprezentaci dané množiny prefixů značně stoupá, protože se exponenciálně zvětšuje velikost bitmap.

Množina	**Parametry**	**Paměť [kB]**	**Počet kroků**
AS6447	n=3	158.83	16.68
	n=5	222.26	10.22
	n=6	379.36	8.60
	n=8	1 177.06	6.66
ALL	n=3	58.32	14.08
	n=5	91.35	8.62
	n=6	155.93	7.40
	n=8	529.33	5.64
RIPE	n=3	26.76	14.94
	n=5	43.39	9.05
	n=6	73.84	7.90
	n=8	257.08	6.00
GEN80	n=3	2 511.61	17.20
	n=5	3 297.91	10.38
	n=6	4 931.65	8.91
	n=8	13 747.79	6.60
GEN120	n=3	2 560.38	17.70
	n=5	3 322.22	10.71
	n=6	5 166.06	9.06
	n=8	14 302.71	7.06
GEN170	n=3	3 601.16	20.57
	n=5	4 637.43	12.33
	n=6	7 130.52	10.45
	n=8	19 780.39	8.02

Tabulka 6.5: Paměťové nároky a rychlost algoritmu Tree Bitmap

6.2.6 Shape Shifting Tree

Struktura tohoto algoritmu v každém uzlu oproti Tree Bitmap obsahuje navíc tvarovou bitmapu. Uzel SST tedy obsahuje:

- Bitmapu vnitřních uzlů, velikost minimálně k bitů
- Bitmapu potomků, velikost minimálně $k + 1$ bitů
- Tvarobou bitmapu, velikost minimálně $2k$ bitů
- Ukazatel do tabulky prefixů
- Ukazatel na prvního potomka

Číslo k tu reprezentuje maximální počet uzlů binární trie, které mohou být uloženy v rámci jednoho uzlu SST.

Při porovnávání hodnot z tabulek tabulek 6.6 a 6.5 si můžeme všimnout, že algoritmus SST pracuje s pamětí hospodárněji než Tree Bitmap. U SST se zvyšováním parametru k opět snižuje výška stromu a tím i teoretická rychlost vyhledávání. Oproti Tree Bitmap zde však tímto paměť potřebná k sestavení trie klesá. Je to dáno tím, že při zvýšení parametru k o a velikost uzlu vzroste pouze o $4a$, tedy lineárně. Celkový počet uzlů trie tím však klesne a v celé struktuře je tedy uloženo méně ukazatelů na potomky.

Množina	Parametry	Paměť [kB]	Počet kroků
AS6447	k=8	132.20	14.00
	k=32	106.41	8.91
	k=64	109.30	8.06
ALL	k=8	57.77	11.34
	k=32	40.08	7.11
	k=64	35.95	6.29
RIPE	k=8	26.86	12.34
	k=32	17.97	8.19
	k=64	16.42	6.74
GEN80	n=8	2 321.48	14.32
	n=32	1 624.36	9.06
	n=64	1 452.95	8.24
GEN120	k=8	2 355.55	14.53
	k=32	1 722.02	9.56
	k=64	1 523.45	8.56
GEN170	k=8	3 313.07	16.34
	k=32	2 376.84	11.37
	k=64	2 145.22	10.25

Tabulka 6.6: Paměťové nároky a rychlost algoritmu Shape Shifting Tree

6.2.7 Hash Tree Bitmap

Uzly jsou v tomto algoritmu shodné s uzly Tree Bitmap, v každém uzly je tedy uloženo:

- Bitmapa vnitřních uzlů, velikost minimálně $2^n - 1$ bitů
- Bitmapa potomků, velikost minimálně 2^n bitů
- Ukazatel do tabulky prefixů
- Ukazatel na prvního potomka

Číslo n je počet bitů zpracovávaných v jednom taktu.

Struktura navíc obsahuje hashovací tabulky různých délek a každý strom struktury ještě ukazatel do tabulky prefixů (kvůli „defaultLPM“, viz kapitola 5.2).

Tabulka 6.7 obsahuje výsledky testů. Oproti algoritmu Tree Bitmap můžeme opět pozorovat velkou úsporu paměti i počtu kroků. Stejně jako v Tree Bitmap i zde se zvyšující se střídou znatelně roste spotřeba paměti a zvyšuje se rychlost. Snížené požadavky na paměť plynou z možnosti odstranění mnoha uzlů přeskočených hashovací funkcí. Tím zároveň snížíme výšku struktury, a proto vyhledávání proběhne rychleji.

6.2.8 Generic Hash Tree Bitmap

Uzly zde oproti uzlům Tree Bitmap navíc obsahují další dvě položky spjaté s hashovacími tabulkami:

- Bitmapa vnitřních uzlů, velikost minimálně $2^n - 1$ bitů

Množina	Parametry	Paměť [kB]	Počet kroků
AS6447	n=3	78.46	4.35
	n=5	98.94	2.74
	n=6	144.31	2.55
	n=8	336.31	2.02
ALL	n=3	21.27	5.92
	n=5	23.37	3.67
	n=6	26.78	3.31
	n=8	43.68	2.41
RIPE	n=3	10.83	3.30
	n=5	12.01	2.26
	n=6	13.58	2.00
	n=8	22.66	1.62
GEN80	n=3	1 065.06	4.66
	n=5	1 502.25	3.06
	n=6	2 093.34	2.68
	n=8	4 975.36	1.81
GEN120	n=3	1 241.61	3.79
	n=5	1 741.21	2.57
	n=6	2 373.73	2.16
	n=8	5 471.67	1.84
GEN170	n=3	1 792.08	3.65
	n=5	2 443.43	2.46
	n=6	3 305.36	2.10
	n=8	7 387.55	1.72

Tabulka 6.7: Paměťové nároky a rychlost algoritmu Hash Tree Bitmap

- Bitmapa potomků, velikost minimálně 2^n bitů
- Ukazatel do tabulky prefixů
- Ukazatel na prvního potomka
- Bitmapa přítomnosti hashovacích tabulek
- Odkazy na intra-tree hashovací tabulky

Číslo n je počet bitů zpracovávaných v jednom taktu. Do velikosti struktury musíme tedy připočítat i velikost intra-tree hashovacích tabulek. Dále je obsažena jedna inter-tree hashovací tabulka (převzatá z algoritmu Hash Tree Bitmap) a odkaz na defaultLPM v každém stromu.

Struktura navíc obsahuje hashovací tabulky různých délek a každý strom struktury ještě ukazatel do tabulky prefixů (kvůli „defaultLPM“, viz kapitola 5.2).

V tabulce 6.8 jsou data z testu algoritmu. Na základě testování v kapitole 4 byla použita varianta odstranění větvení bez prefixů, parametry $n = 5; h = [30, 20, 10]$ a různé délky povolené expanze.

Data můžeme opět srovnat s algoritmem Tree Bitmap, ze kterého algoritmus vychází (tabulka 6.5), případně s algoritmem Hash Tree Bitmap (tabulka 6.7). Vidíme, že oproti

Množina	Parametry	Paměť [kB]	Počet kroků
AS6447	n=5; h=[30,20,10]; s=0; l=0	76.88	1.85
	n=5; h=[30,20,10]; s=2; l=2	66.25	1.49
	n=5; h=[30,20,10]; s=4; l=4	76.68	1.37
ALL	n=5; h=[30,20,10]; s=0; l=0	22.87	2.63
	n=5; h=[30,20,10]; s=2; l=2	23.93	1.86
	n=5; h=[30,20,10]; s=4; l=4	22.11	1.58
RIPE	n=5; h=[30,20,10]; s=0; l=0	11.98	1.78
	n=5; h=[30,20,10]; s=2; l=2	12.01	1.41
	n=5; h=[30,20,10]; s=4; l=4	11.60	1.09
GEN80	n=5; h=[30,20,10]; s=0; l=0	1 297.00	2.17
	n=5; h=[30,20,10]; s=2; l=2	835.03	1.47
	n=5; h=[30,20,10]; s=4; l=4	942.79	1.41
GEN120	n=5; h=[30,20,10]; s=0; l=0	1 550.35	2.13
	n=5; h=[30,20,10]; s=2; l=2	1 026.46	1.52
	n=5; h=[30,20,10]; s=4; l=4	1 117.27	1.35
GEN170	n=5; h=[30,20,10]; s=0; l=0	2 116.94	1.81
	n=5; h=[30,20,10]; s=2; l=2	1 443.33	1.35
	n=5; h=[30,20,10]; s=4; l=4	1 581.88	1.24

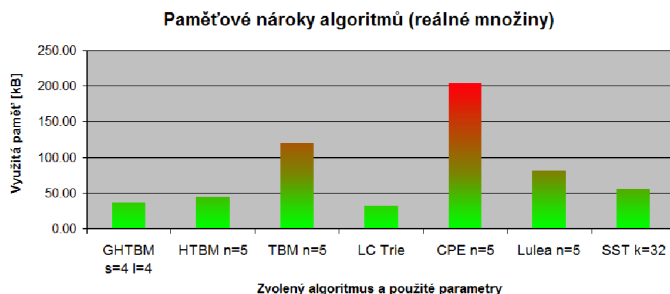
Tabulka 6.8: Paměťové nároky a rychlost algoritmu Generic Hash Tree Bitmap

oběma těmto algoritmům dosahuje generická varianta nejlepších výsledků jak v množství využité paměti, tak v rychlosti vyhledávání prefixů.

6.3 Srovnání na reálných množinách

První srovnání bylo provedeno na reálných množinách prefixů. Byla použita zprůměrovaná data z tabulek z kapitoly 6.2. Průměrné počty kroků a množství spotřebované paměti pro reprezentaci reálných množin prefixů shrnuje tabulka 6.9.

Průměrné paměťové nároky jednotlivých algoritmů (se střídou $n = 5$) pro reprezentaci množin prefixů byly vyneseny do grafu 6.1. U algoritmu GHTBM byla vybrána varianta s parametry $s = 4; l = 4$. Pro zvýšení přehlednosti byl z grafu vypuštěn algoritmus Trie, který by kvůli svým značným paměťovým nárokům výrazně rozšířil rozsah hodnot na ose.

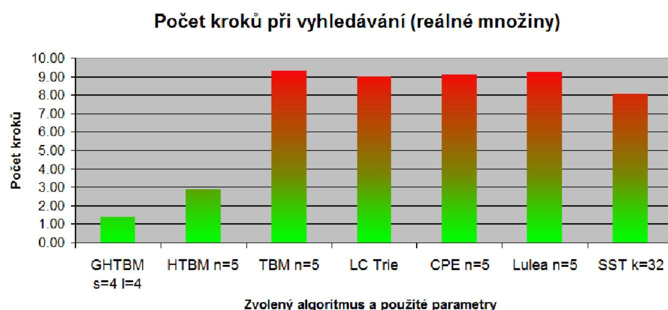


Obrázek 6.1: Srovnání paměťových nároků jednotlivých algoritmů na reálné množině prefixů

Algoritmus	Parametry	Paměť [kB]	Počet kroků
GHTBM	n=5; h=[30,20,10]; s=0; l=0	37.24	2.09
	n=5; h=[30,20,10]; s=2; l=2	34.06	1.59
	n=5; h=[30,20,10]; s=4; l=4	36.80	1.35
HTBM	n=3	36.85	4.52
	n=5	44.77	2.89
	n=8	134.22	2.02
TBM	n=3	81.30	15.23
	n=5	119.00	9.30
	n=8	654.49	6.10
Trie	–	331.45	43.74
LC Trie	–	32.26	9.00
CPE	n=3	111.62	14.88
	n=5	203.62	9.11
	n=6	620.58	7.69
Lulea	n=3	70.82	15.16
	n=5	81.07	9.25
	n=6	124.65	7.85
SST	k=8	72.28	12.56
	k=32	54.82	8.07
	k=64	53.89	7.03

Tabulka 6.9: Srování algoritmů na reálných množinách prefixů

Do grafu na obrázku 6.2 byl vynesen průměrný počet kroků pro vyhledání prefixů k testovacím IPv6 adresám pomocí jednotlivých algoritmů. Pro zvýšení přehlednosti byl opět vypuštěn algoritmus Trie.



Obrázek 6.2: Srovnání rychlosti jednotlivých algoritmů na reálné množině prefixů

Z grafů je patrné, že implementovaný algoritmus Generic Hash Tree Bitmap je v tomto testu nejrychlejší ze všech srovnávaných algoritmů. Vyžaduje o 50 % méně kroků než Hash Tree Bitmap a je dokonce téměř 7x rychlejší než (v dnešní době hojně používaný) Tree Bitmap. Přitom paměťové nároky má z těchto tří algoritmů nejnižší. Paměťově úspornější je pouze algoritmu LC Trie, který ovšem při vyhledávání prefixů vyžadoval téměř 7x více kroků.

6.4 Srovnání na generovaných množinách

Přestože množina prefixů z AS6447 je největší množina dostupná v rámci knihovny Netbench, obsahuje pouze necelých 13000 záznamů. Dá se předpokládat, že všechny množiny v následujících letech rychle porostou. Proto bylo srovnání provedeno i na generovaných množinách GEN80, GEN120 a GEN170 (viz kapitola 6.1.2). Průměrné počty vyhledávacích kroků a množství spotřebované paměti pro reprezentaci těchto množin shrnuje tabulka 6.10.

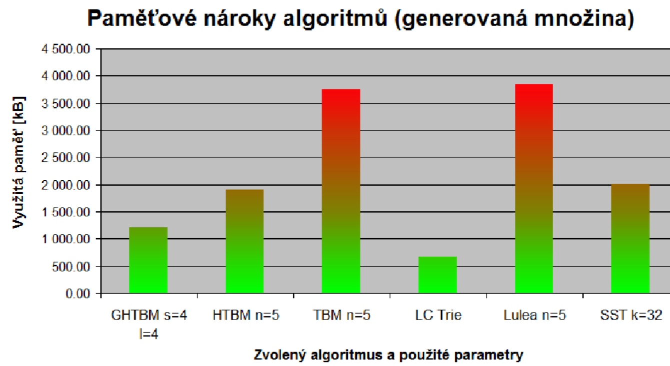
Algoritmus	**Parametry**	**Paměť [kB]**	**Počet kroků**
GHTBM	n=5; h=[30,20,10]; s=0; l=0	1 654.76	1.97
	n=5; h=[30,20,10]; s=2; l=2	1 101.61	1.45
	n=5; h=[30,20,10]; s=4; l=4	1 213.98	1.33
HTBM	n=3	1 366.25	4.03
	n=5	1 895.63	2.70
	n=8	5 944.86	1.79
TBM	n=3	2 891.05	18.49
	n=5	3 752.52	11.14
	n=8	15 943.63	7.22
Trie	–	13 445.91	53.09
LC Trie	–	670.11	7.30
CPE	n=3	6 362.14	18.40
	n=5	23 226.76	11.11
	n=6	41 473.66	9.44
Lulea	n=3	3 304.64	18.40
	n=5	3 846.42	11.11
	n=6	4 606.86	9.44
SST	k=8	2 663.21	15.06
	k=32	1 907.62	10.00
	k=64	1 706.89	9.02

Tabulka 6.10: Srování algoritmů na generovaných množinách prefixů

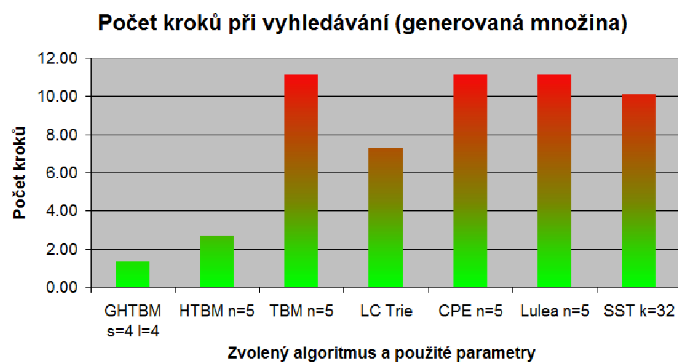
Paměťové nároky jednotlivých algoritmů pro reprezentaci generovaných množin prefixů byly vyneseny do grafu 6.3. Pro zvýšení přehlednosti byly z grafu vypuštěny algoritmy CPE a Trie, které by kvůli své značné spotřebě paměti výrazně rozšířily rozsah hodnot na ose. U algoritmu GHTBM byla do grafu opět vybrána varianta s parametry $s = 4; l = 4$.

Obrázek 6.4 srovnává průměrný počet kroků nutných pro nalezení prefixů v testovacích množinách IPv6 adres při použití jednotlivých algoritmů. V tomto grafu opět (pro lepší přehlednost) chybí algoritmus Trie.

Vidíme, že i na větších množinách prefixů stále nový algoritmus Generic Hash Tree Bitmap svojí rychlostí dominuje. Je rychlejší o více než 50 % než Hash Tree Bitmap a opět vyžaduje znatelně méně paměti.



Obrázek 6.3: Srovnání paměťových nároků jednotlivých algoritmů na generovaných množinách prefixů



Obrázek 6.4: Srovnání rychlosti jednotlivých algoritmů na generovaných množinách prefixů

Kapitola 7

Závěr

Práce analyzuje a popisuje algoritmy pro vyhledávání nejdelšího shodného prefixu se zaměřením na jejich rychlost, paměťovou náročnost a vhodnost pro hardwarovou implementaci. Je vysvětleno, jak reprezentují množinu prefixů pomocí svých datových struktur a jak v ní následně vyhledávají.

Na základě získaných poznatků je navržen nový algoritmus Generic Hash Tree Bitmap. Ten rychlostí vyhledávání mnohonásobně předčí například i algoritmus Tree Bitmap, který se v současné době hojně využívá. Značně menší jsou i jeho paměťové nároky. Klíčem k vynikajícím výsledkům představeného řešení je možnost vkládání malých hashovacích tabulek přímo do jednotlivých uzlů stromové struktury reprezentující množinu prefixů. Algoritmus navíc umožňuje nastavením svých parametrů do značné míry upravovat poměr mezi paměťovými nároky a rychlostí vyhledávání.

Práci je možné v budoucnu rozšířit například o využití Steinerových minimálních stromů (viz kapitola 4.4) a srovnání této metody s ostatními. Lze také vyvíjet další optimalizace současných algoritmů, případně hledat zcela nové přístupy.

Implementace Generic Hash Tree Bitmap byla začleněna do knihovny Netbench [8], a tak tato práce bude podporovat další vývoj LPM algoritmů (nejen) na FIT VUT v Brně.

Literatura

[1] Accelerated Network Technologies [online]. `http://merlin.fit.vutbr.cz/ant/`, [cit. 2013-04-26].

[2] BGP Reports [online]. `http://bgp.potaroo.net/index-bgp.html`, [cit. 2013-04-26].

[3] SixXS, IPv6 Deployment and IPv6 Tunnel Broker [online]. `http://www.sixxs.net/`, [cit. 2013-04-26].

[4] Degermark, M.; Brodnik, A.; Carlsson, S.; aj.: Small forwarding tables for fast routing lookups. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '97, New York, NY, USA: ACM, 1997, ISBN 0-89791-905-X, s. 3–14.

[5] Eatherton, W.; Varghese, G.; Dittia, Z.: Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, ročník 34, April 2004: s. 97–122, ISSN 0146-4833.

[6] Lampson, B.; Srinivasan, V.; Varghese, G.: IP lookups using multiway and multicolumn search. *IEEE/ACM Trans. Netw.*, ročník 7, June 1999: s. 324–334, ISSN 1063-6692.

[7] Nilsson, S.; Karlsson, G.: IP-Address Lookup Using LC-Tries. *IEEE Journal on Selected Areas in Communications*, ročník 17, 1999, ISSN 0733-8716.

[8] Puš, V.; Tobola, J.; Kaštil, J.; aj.: Netbench – the Framework for Evaluation of Packet Processing Algorithms. In *Proceedings of the 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, IEEE Computer Society, 2011, ISBN 978-0-7695-4521-9, s. 95–96.

[9] Song, H.; Turner, J.; Lockwood, J.: Shape Shifting Tries for Faster IP Route Lookup. In *Proceedings of the 13TH IEEE International Conference on Network Protocols*, Washington, DC, USA: IEEE Computer Society, 2005, ISBN 0-7695-2437-0, s. 358–367.

[10] Srinivasan, V.; Varghese, G.: Faster IP lookups using controlled prefix expansion. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '98/PERFORMANCE '98, New York, NY, USA: ACM, 1998, ISBN 0-89791-982-3, s. 1–10.

[11] Tobola, J.: Vyhledání nejdelšího prefixu. Pojednání k tématu dizertační práce, FIT VUT v Brně, 2009.

[12] Tobola, J.; Kořenek, J.: Effective Hash-based IPv6 Longest Prefix Match. In *IEEE Design and Diagnostics of Electronic Circuits and Systems DDECS'2011*, IEEE Computer Society, 2011, ISBN 978-1-4244-9753-9, s. 325–328.

[13] Waldvogel, M.; Varghese, G.; Turner, J.; aj.: Scalable high speed IP routing lookups. 1997: s. 25–36.

[14] Wu, B. Y.; Chao, K.-M.: *Spanning Trees and Optimization Problems.* Chapman & Hall, 2004, iSBN 1-58488-436-3.

[15] Zheng, K.; Liu, B.: V6Gene: A Scalable IPv6 Prefix Generator for Route Lookup Algorithm Benchmark. In *AINA '06 Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, IEEE Computer Society, 2006, ISBN 0-7695-2466-4-01, s. 147–152.

Příloha A

Obsah DVD

Přiložené DVD obsahuje celý repozitář projektu Netbench a elektronickou verzi tohoto textu.