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

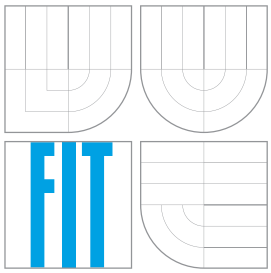
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FRANTIŠEK SEDLÁŘ

BRNO 2011

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

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FRANTIŠEK SEDLÁŘ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ TOBOLA

BRNO 2011

Abstrakt

V této bakalářské práci byly popsány základní algoritmy pro vyhledání nejdelšího shodného prefixu (LPM). K již existujícím implementacím v knihovně Netbench byl přidán další algoritmus – LC Trie. Všechny algoritmy, které knihovna obsahuje, byly testovány nad reálnými množinami IPv6 prefixů. Na základě zde získaných dat byly navzájem porovnány. Dále byly sepsány skripty pro stahování prefixů z významných zdrojů na internetu a testovací skripty k jednotlivým algoritmům.

Abstract

This bachelor's thesis deals with a description of basic longest prefix match (LPM) algorithms. Another algorithm – LC Trie – was added to existing implementations into the Netbench library. All the algorithms which the library includes were tested with real groups of IPv6 prefixes. They were compared on the basis of previously obtained data. Testing scripts for each of the algorithms were implemented as well as scripts for downloading groups of prefixes from significant sources on the internet.

Klíčová slova

nejdelší shodný prefix, trie, LC Trie, Controlled Prefix Expansion, Lulea Compressed Trie, Tree Bitmap, Shape Shifting Tree, 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, binary search on intervals, binary search on prefixes

Citace

František Sedlář: Algoritmy pro vyhledání nejdelšího shodného prefixu, bakalářská práce, Brno, FIT VUT v Brně, 2011

Algoritmy pro vyhledání nejdelšího shodného prefixu

Prohlášení

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

.....
František Sedlář
17. května 2011

Poděkování

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

Obsah

Kapitola 1

Úvod

S rozmachem a zrychlováním počítačových sítí rostou nároky nejenom na technologie zajišťující samotný přenos (metalické a optické kabely, bezdrátové prvky), ale také na další zařízení, která jsou v této infrastruktuře nezbytná. Ať už jde o prvek pracující na linkové vrstvě, síťové vrstvě, či vyšších vrstvách, 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.

Pakliže (dle [12]) na vytížené 10 Gb/s lince přicházejí pakety v průměru každých 67 ns, je třeba v tomto čase projít routovací 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 v routovacích 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 popisuje základní algoritmy, včetně příkladů sestavení potřebných struktur pro konkrétní množinu prefixů. Kapitola 3 obsahuje popis implementace algoritmu LC Trie a jeho včlenění do projektu Netbench výzkumné skupiny ANT na FIT VUT v Brně. V kapitole 4 jsou jednotlivé algoritmy testovány nad reálnými množinami IPv6 prefixů a na základě takto získaných dat porovnány.

Kapitola 2

Přehled algoritmů

V tomto přehledu se objeví základní LPM algoritmy, které jsou obsaženy v rámci knihovny Netbench, výzkumné skupiny ANT [1]. Stručný popis většiny z nich je součástí například [12].

Pro popis algoritmů byla zvolena malá množina velice krátkých prefixů, viz obr. 2.1. 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 2.1: Testovací množina prefixů

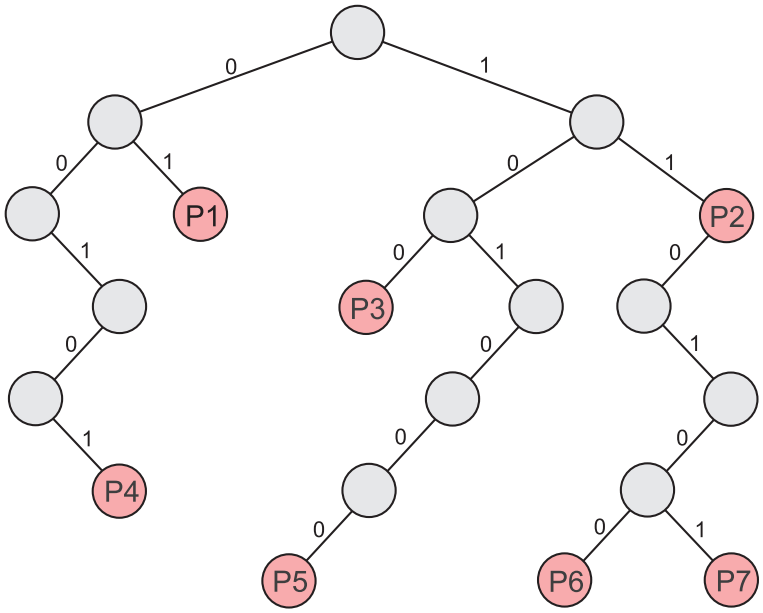
2.1 Algoritmy založené na struktuře trie

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 IP adrese. 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.

2.1.1 Trie

Trie pracuje s binárním stromem. Jedná se o singlebit algoritmus, což znamená, že vstupní adresu zpracováváme po jednotlivých bitech. Každý uzel trie označuje jeden bit prefixu. Prefix konkrétního uzlu je tvořen hodnotami všech uzlů po cestě od kořene. 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 2.1 můžete vidět na obrázku 2.2.



Obrázek 2.2: Trie

Při vyhledávání prefixu k dané IP adrese se postupuje, jak jsme si již řekli, 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 bereme v potaz levý podstrom.

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, pokud narazíme na listový uzel.

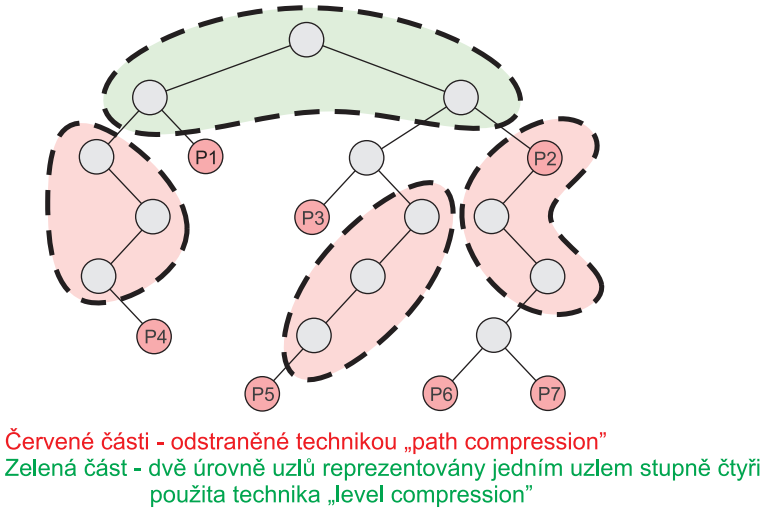
Nevýhodou algoritmu je jeho velká paměťová náročnost a velký počet náhodných přístupů do paměti (dáno délkou nejdelšího prefixu). Vzhledem k použití delších prefixů lze u IPv6 očekávat značné zhoršení obou těchto parametrů.

2.1.2 LC Trie

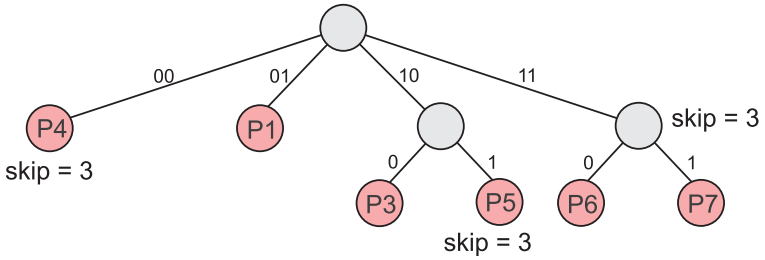
Jedná se o multibit algoritmus, který pracuje s proměnným počtem bitů v jednom taktu. Z toho vyplývá jeho nevhodnost pro hardwarovou implementaci. Byl představen v [8].

Pro úsporu paměti potřebné pro reprezentaci trie využívá algoritmus techniky známé jako path compression a level compression. Obě tyto techniky jsou patrné na obrázku 2.3, který znázorňuje klasickou binární trie, pouze jsou na ní vyznačeny úseky, kde jsou uvedené techniky použity. Tento obrázek lze srovnat s obrázkem 2.4, kde je finální podoba trie postavené algoritmem LC trie.

Path compression (na obrázku 2.3 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. To, kolik bitů je v každém uzlu přeskočeno, je označováno jako „skip value“.



Obrázek 2.3: Ukázka komprese trie použité v algoritmu LC Trie



Obrázek 2.4: Trie sestavená algoritmem LC Trie

„Skip value" 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 2.3. 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 odsekneme celý prefix, který je reprezentován tímto uzlem. Poté ještě odsekneme znaky, které jsou v tomto uzlu označeny jako „skip". Nastavíme faktor větvení na 1. Ze všech řetězců vysekneme 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, vyjádřenou binárně, podle abecedy. Množina nesmí obsahovat prefixy, které jsou platnými prefixy dalších (delších) prefixů.

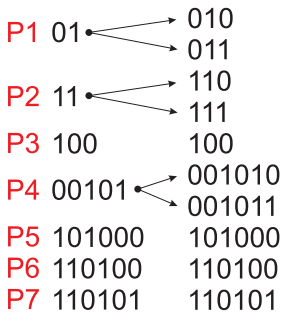
Kratší prefixy přesuneme do zvláštní tabulky. Podrobný popis budování trie je součástí [8].

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 znaků 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 uložených zde. 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.

2.1.3 Controlled Prefix Expansion

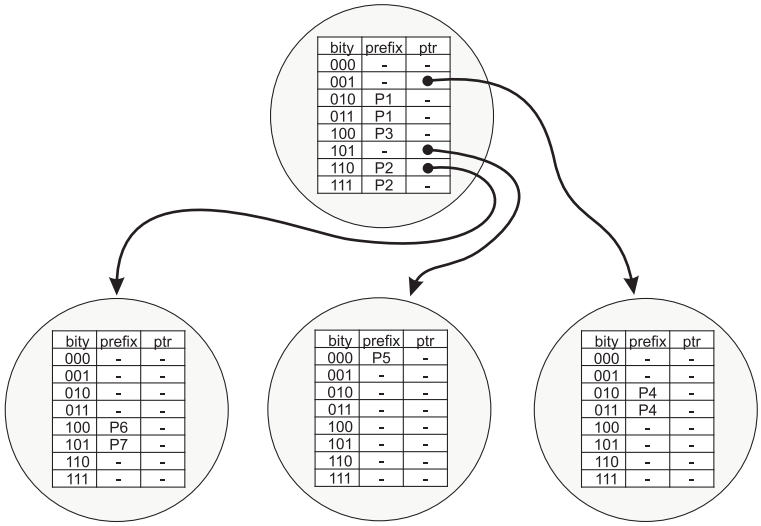
Tento algoritmus byl představený v [10], v tomto textu bývá označován zkratkou „CPE“. Jedná se o multibit algoritmus, pro zrychlení se zde pracuje s více bity prefixu v jednom taktu. Pro ilustraci uvedeme trie, která pracuje po trojicích bitů. Protože množina prefixů, které je třeba do trie uložit, nemusí 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 2.5: Ukázka expandování uzlů

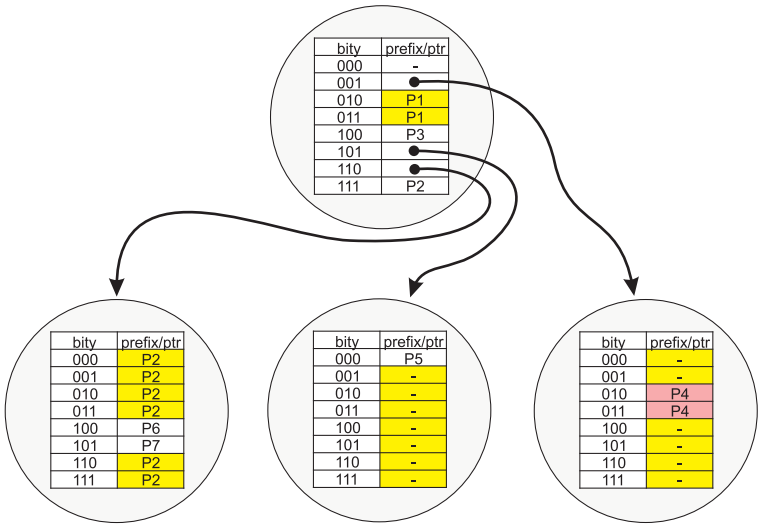
Expandovaná množina prefixů je znázorněna na obrázku 2.5. Každý uzel trie tedy obsahuje ukazatel do tabulky prefixů a ukazatel na následující uzly pro všechny varianty vstupu (při zpracování po trojicích bitů bude každý uzel obsahovat celkem 16 ukazatelů). To můžeme vidět na obrázku 2.6, kde je znázorněna celá trie pro expandovanou množinu prefixů z obrázku 2.1.

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 2.7. Zde jsou



Obrázek 2.6: Trie sestavená algoritmem CPE

zároveň barevně vyznačeny redundantní hodnoty, které se zde bohužel často ukládají. Tuto nevýhodu dále odstraňuje algoritmus Lulea Compressed Trie.



Obrázek 2.7: Trie sestavená algoritmem CPE s optimalizací „leaf pushing“

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. 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 nejpozději uložený (tedy nejdelší) nalezený prefix.

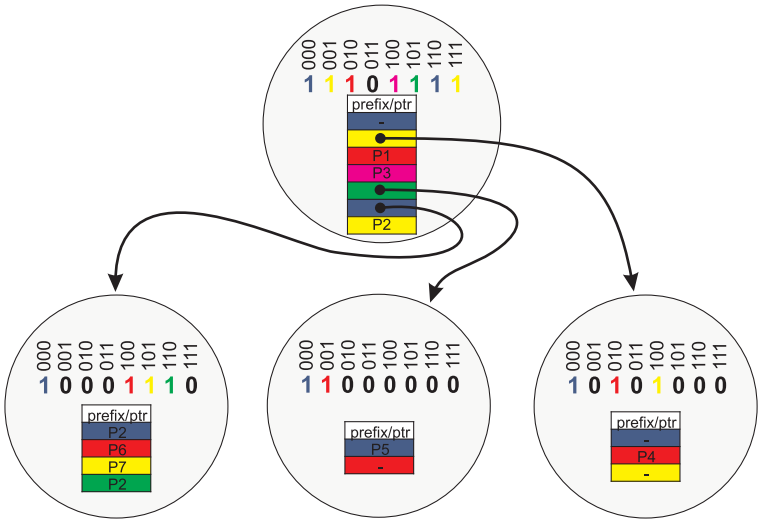
Obecně lze říci, že jeden uzel trie postavené pomocí základního neoptimalizovaného algoritmu se skládá z 2^n ukazatelů na další uzly + 2^n ukazatelů do tabulky prefixů, kde n je počet bitů zpracovávaných v jednom taktu. S využitím optimalizace „leaf pushing“ pak jeden uzel trie sestává „pouze“ z 2^n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

2.1.4 Lulea Compressed Tries

Tento algoritmus (představen v [5]) je další optimalizací CPE. U „leaf pushing“ se v uzlech často objevuje několik shodných ukazatelů pro různé varianty vstupu. Lulea compressed trie (v textu též zkráceně označován jako „Lulea“) se tuto redundanci snaží odstranit.

Každý uzel pak obsahuje navíc bitmapu. Ta obsahuje tolik bitů, kolik variant vstupů má každý uzel – při práci s trojicemi bitů prefixu tedy bitmapa obsahuje 8 bitů. Pokud jsou v uzlu uloženy shodné ukazatele v po sobě následujících kombinacích vstupů, je zachován pouze první výskyt tohoto ukazatele 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.

Trie reprezentující testovací množinu prefixů je na obrázku 2.8. Při vytváření byla použita expandovaná varianta prefixů z popisu algoritmu CPE (viz obr. 2.5). Úsporu paměti, kterou algoritmus oproti CPE přináší, lze pozorovat při porovnání s obrázkem 2.7.



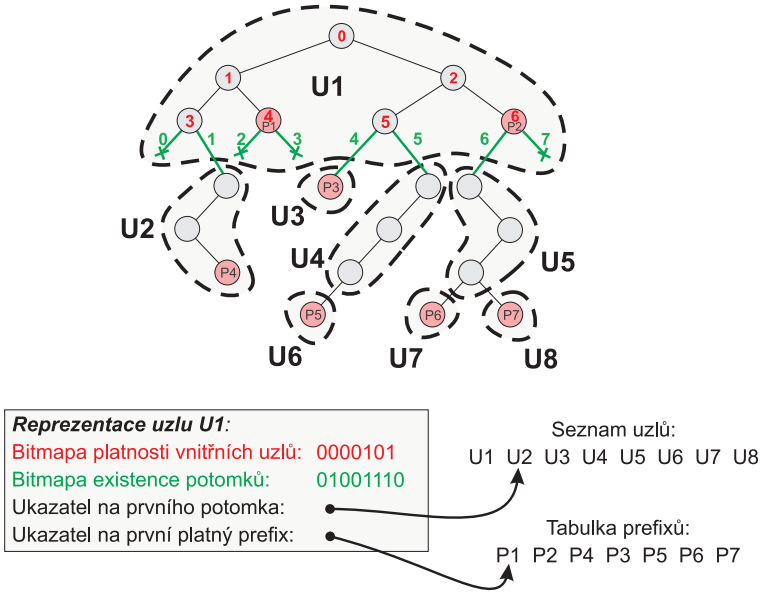
Obrázek 2.8: Trie sestavená algoritmem Lulea Compressed Tries

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 ukazately na další uzel, případně zde můžeme opět nalézt ukazatel do tabulky prefixů a vyhledávání končí. Vyhledávání rovněž (neúspěšně) končí, pokud narazíme na nulový ukazatel.

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.

2.1.5 Tree Bitmap

Tento multibit algoritmus byl představen v [6]. Pracuje se stejným počtem bitů v jednom taktu. Při popisu budeme uvažovat 3 bity v jednom taktu. Každý uzel pak má až $2^3 = 8$ následníků. V rámci uzlu je vlastně uložena odpovídající část binárního stromu. To je opět vidět na celé sestavené trie na obrázku 2.9. Je zde zobrazena celá binární trie, na které jsou ohraničené jednotlivé uzly Tree Bitmap.



Obrázek 2.9: Trie sestavená algoritmem Tree Bitmap

Každý tento uzel tedy může obsahovat až 7 platných prefixů uvnitř (reprezentovaných 8 vnitřními uzly). Každý uzel trie může mít až 8 potomků (potomci tohoto uzlu jsou vlastně potomky čtyřech 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 8 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 na prefix pouze u prvního platného vnitřního uzlu.

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

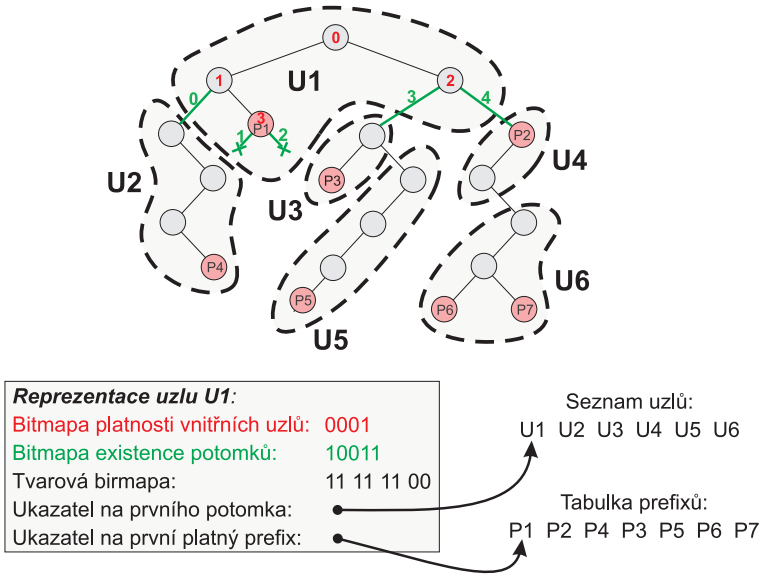
Při vyhledávání rozdělíme vstupní řetězec na části po takovém počtu znaků, které jsme zvolili při budování trie, v našem případě po trojicích. Začneme v kořenovém uzlu s první trojicí znaků. V bitmapě, která reprezentuje vnitřní uzly uzlu, zjistíme, zda se po cestě nachází nějaký platný prefix. Pokud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

mek 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 znaky vstupního řetězce nebo když v bitmapě potomků zjistíme, že potomek neexistuje.

Na obr. 2.9 je vidět, co vše je uložena v rámci jednoho uzlu trie 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.

2.1.6 Shape Shifting Tree

Algoritmus představený v [9] (v tomto textu často nazýván zkratkou „SST“) je velmi podobný algoritmu TreeBitmap, pouze se snaží o efektivnější rozdělení uzlů binárního stromu do uzlů SST. V rámci jednoho uzlu SST není vždy několik kompletních úrovní binárního stromu, ale (jak již název napovídá) úseky různého tvaru (viz obr. 2.10). 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 „h“).



Obrázek 2.10: Trie sestavená algoritmem Shape Shifting Tree

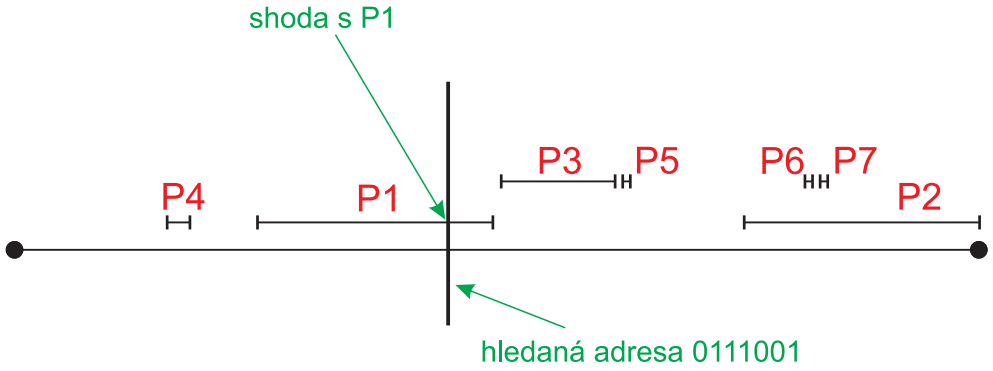
SST struktura se buduje z binárního stromu. Nejprve je nutné pro všechny uzly spočítat hodnotu p – počet uzlů, které obsahuje podstrom s tímto uzlem jako kořenem. Poté procházíme uzly v breadth-first pořadí, a pokud narazíme na uzel, který má hodnotu p menší nebo rovnu hranici h, odstraníme celý tento 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 $p \leq h$. 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.

2.2 Ostatní algoritmy

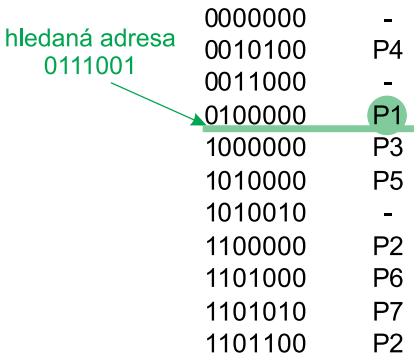
2.2.1 Binární vyhledávání na intervalech

Tento algoritmus (představený v [7], zde uváděný také zkratkou „BSI“ – z anglického „Binary Search on Intervals“) pohlíží na množinu vstupních prefixů 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 2.11: 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 2.11.



Obrázek 2.12: Počáteční adresy prefixů uložené v tabulce

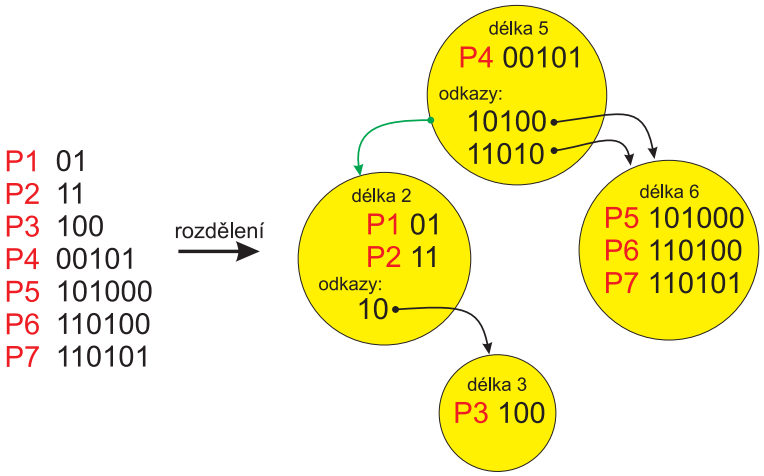
Z těchto intervalů sestaví algoritmus tabulku – pro každý interval vstupních hodnot je dopředu určen prefix, který je v tomto intervalu nejdelším shodným. Tyto intervaly jsou

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 2.12, pro jednoduchost jsou tu použity 7 bitové adresy. Na obou obrázcích je znázorněno vyhledání adresy 0111001.

2.2.2 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 2.13.



Obrázek 2.13: 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 skupiny, ze kterých byly 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 2.13 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 opět nalézt v [13].

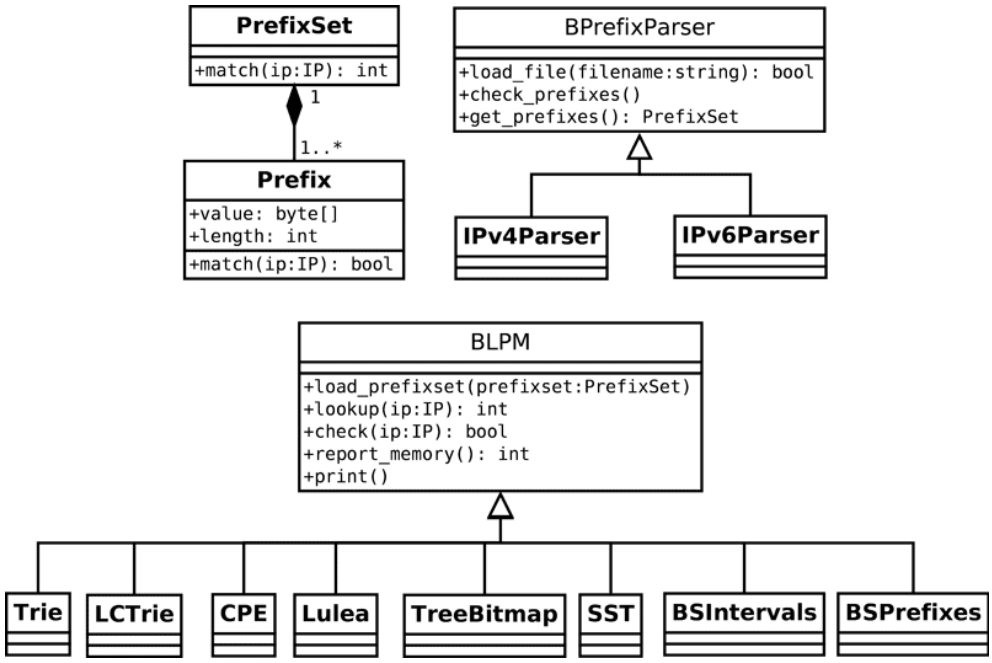
Přestože paměťové nároky algoritmu jsou zpravidla vyšší než u jiných LPM metod, vyhledávání samotné je poté velice rychlé.

Kapitola 3

Implementace

3.1 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 3.1: Objektový model LPM části Netbench

Netbench obsahuje mnoho modulů pro různá odvětví, v naší práci je podstatná část LPM, jejíž objektový model je na obrázku 3.1. Skládá se z bázové třídy `BLPM`, ze které dědí třídy implementující jednotlivé LPM algoritmy. V rámci této práce byla implementována především třída `LCTrie`, její detailnější popis je uveden v kapitole 3.2. 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.

3.2 Algoritmus LC Trie

Jádrem implementace algoritmu LC Trie je třída `LCTrie`. Metoda `load_prefixset`, která vyžaduje jediný parametr, objekt třídy `PrefixSet`, nejdříve zajistí seřazení vstupních prefixů. Poté z množiny prefixů odstraní prefixy, které jsou prefixy jiných prefixů, a tyto vloží do speciální tabulky. Po vytvoření instance třídy `Trie` je volána její metoda `build`. Ta je při budování trie volána rekurzivně – vždy pro různé prefixy.

Metoda `build` vyžaduje celkem čtyři parametry. Druhým parametrem je předán počet prefixů, které mají být zpracovány, počínaje indexem prefixu předaného prvním parametrem. Třetí parametr značí, kolik počátečních bitů těchto prefixů má být ignorováno (což je v podstatě hloubka, ve které bychom se aktuálně nacházeli při použití klasické trie). Poslední parametr je ukazatelem na první volnou pozici v seznamu uzlů trie. Protože je trie implementována jako seznam objektů třídy `Node`, ukazatelem je index prvního volného prvku tohoto seznamu.

Pokud byl metodě `build` parametrem předán pouze jeden prefix, je tento uložen jako uzel s nulovou hodnotou branch. V takovém listovém uzlu je dále uložena hodnota skip (předem vypočtena metodou `computeSkip`) a „ukazatel“ – index prefixu ve struktuře prefixset, který tomuto uzlu odpovídá. Tento uzel je uložen na první volné místo seznamu uzlů.

Pakliže je zpracováváno více prefixů, je pomocí metody `computeBranch` spočítána hodnota branch, podle které se trie rozvětví. Následně je alokována paměť pro všechny uzly, které tímto větvením vzniknou. Na první volné místo v seznamu uzlů je pak uložen uzel obsahující patřičné informace o větvení – hodnota branch, skip a ukazatel na prvního potomka tohoto uzlu. Dále se rozhodne, které prefixy budou jednotlivé větve obsahovat, a metoda `build` je rekurzivně volána pro tyto prefixy (větve).

Pro vyhledání prefixů pro danou IP adresu se použije metoda `lookup` třídy `LCTrie`. Jediným povinným parametrem je zkoumaná IP adresa. Při vyhledávání se prochází trie, dokud se nenarazí na listový uzel. Protože několik bitů mohlo být při vyhledávání přeskočeno, je nutné po nalezení listového uzlu provést ještě jedno porovnání, zda nalezený prefix skutečně odpovídá v celé svojí délce zkoumané IP adrese. Pokud ano, je tento zařazen na začátek seznamu jako nejdelší prefix. Nakonec je nutné projít tabulkou prefixů, protože prefixy zde uložené mohou být také platnými prefixy pro danou IP.

Kapitola 4

Simulace a testování algoritmů

Pro testování všech algoritmů byly sesbírány reálné množiny IPv6 prefixů. Významným zdrojem byl server [4], některé další množiny pocházejí z [2] a [3]. Jména souborů, ve kterých jsou jednotlivé množiny uloženy, jsou ve tvaru „pocet_zdroj_datum“, kde „pocet“ udává počet prefixů v dané množině, „zdroj“ udává zdroj, odkud prefixy pocházejí a „datum“ je datum ve formátu RRRR_MM_DD, kdy byly prefixy z daného zdroje staženy.

Pro stahování prefixů z jednotlivých zdrojů byly v jazyce python sepsány jednoduché skripty, které tuto práci usnadnily. Dále byly pro jednotlivé algoritmy vytvořeny testovací skripty, které postupně ze všech testovaných množin prefixů zadaným algoritmem sestavily potřebné struktury a uložily informace o využité paměti.

V následujících testech budou paměťové nároky daného algoritmu uváděny jak s použitím 32bitové reprezentace ukazatele, tak s nejmenším možným ukazatelem.

4.1 Jednotlivé algoritmy

4.1.1 Trie

Jeden 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.

Paměťové nároky tohoto algoritmu jsou zobrazeny v tabulce 4.1. Je vidět, že zvolením nejmenšího možného ukazatele lze ušetřit v průměru až 60 % paměti, přesto jsou však paměťové nároky vysoké.

prefixů	hloubka	32-bit [B]	nejmenší [B]	ušetřeno
517	48	57 648	21 618	62,5 %
736	48	76 128	28 548	62,5 %
815	48	83 244	31 216	62,5 %
933	128	73 776	27 666	62,5 %
1 914	48	134 856	54 785	59,4 %
3 433	48	289 824	126 798	56,3 %
5 498	128	473 820	222 103	53,1 %

Tabulka 4.1: Paměťové nároky metody Trie na různých množinách IPv6 prefixů

4.1.2 LC Trie

Dle kapitoly 2.1.2 každý uzel takto sestavené trie obsahuje:

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

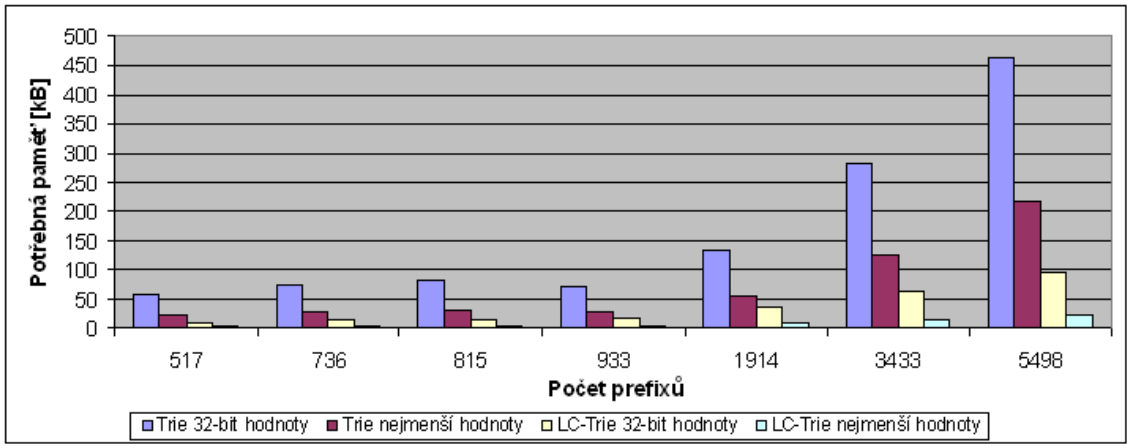
Pokud bychom tedy na hodnoty branch, skip i pro ukazatel použili 32bitové číslo, dostali bychom se na hodnotu 96 bitů pro uložení každého uzlu. Při představení algoritmu v [8] autoři uvádějí verzi, kdy jeden uzel trie vyžaduje pouze 32 bitů. Pro hodnotu branch je zde použito 5 bitů, 7 bitů pro hodnotu 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 $2^3 1$, což je jistě dostačující. Hodnotu skip lze uložit v rozsahu 0–127, což je opět dostačující i pro IPv6.

prefixů	hloubka	32-bit [B]	nejmenší [B]	ušetřeno	nejmenší reprezentace [b] branch	skip	ukazatel
517	15	10 188	1 910	81,3 %	2	4	10
736	16	14 388	2 847	80,2 %	2	4	11
815	15	15 780	3 123	80,2 %	2	4	11
933	20	17 052	3 730	78,1 %	2	6	11
1 914	16	35 508	7 027	80,2 %	2	3	12
3 433	18	65 100	14 240	78,1 %	2	4	13
5 498	21	98 244	23 537	76,0 %	2	6	13

Tabulka 4.2: Paměťové nároky metody LC Trie na různých množinách IPv6 prefixů

V tabulce 4.2 lze kromě paměťových nároků pro reprezentaci testovacích množin IPv6 prefixů vidět i počet bitů nutných k uložení jednotlivých hodnot v rámci uzlu. Algoritmus LC Trie vychází z algoritmu Trie, používá však metody k její kompresi. Porovnáním tabulek 4.2 a 4.1 zjistíme, že algoritmus LC Trie vyžaduje k reprezentaci stejných množin v průměru

10krát méně paměti. To je ostatně vidět i na grafu v obrázku 4.1, který oba algoritmy porovnává.



Obrázek 4.1: Paměťové nároky metod LC Trie a Trie na různých množinách IPv6 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. U struktury LC Trie je však ještě nutno počítat s tím, že některé prefixy mohou být z trie vyloučeny a zařazeny do speciální tabulky prefixů.

Zajímavá je následující tabulka 4.3, která ukazuje velice jednoduché porovnání obou algoritmů pro 2 reálné množiny prefixů. Obě obsahovaly 933 záznamů, avšak jedna množina byla tvořena IPv6 prefixy a druhá IPv4.

množina	**32-bit [B]**	**nejmenší [B]**
IPv4	143268,0	56710,3
IPv6	73776,0	27666,0

Tabulka 4.3: Porovnání paměti potřebné pro reprezentaci IPv4 a IPv6 prefixů

Dalo by se očekávat, že (především u algoritmu Trie) bude pro reprezentaci IPv6 množiny prefixů třeba podstatně více paměti než pro množinu IPv4. Jak se ovšem ukázalo, opak byl pravdou. Zatímco pro kratší prefixy bylo zapotřebí 143 kB, pro delší prefixy to bylo 74 kB.

Vzhledem k tomu, že jsem tento výsledek neočekával, zaměřil jsem se na zmíněný test více. Podrobnější výstup z tohoto testu reprezentuje tabulka 4.4.

množina	**prefixů**	**32-bit [B]**	**nejmenší [B]**	**uzlů**	**hloubka**	**délka prefixů**
IPv4	933	143268,0	56710,3	11880	27	22,5
IPv6	933	73776,0	27666,0	6148	128	36,9

Tabulka 4.4: Detailnější porovnání paměti potřebné pro reprezentaci IPv4 a IPv6 prefixů

U IPv4 obsahuje trie téměř dvakrát více uzlů, než je tomu u IPv6. Přitom hloubka trie (podle předpokladů) je mnohem nižší (dána nejdelším prefixem v množině). I průměrná délka prefixu je menší.

Ukázalo se, že tato vlastnost je způsobena systémem přidělování IPv6 adres. Protože se dnes pro unicast přidělují adresy s prefixem 2000::/3, jsou v reálné množině pouze tyto. Navíc jsou prefixy pěkně zarovnány a tak se drtivá většina prefixů na svých počátečních 15 až 20 bitech shoduje. Kromě toho jsou zmíněné IPv6 prefixy poměrně krátké, průměrná délka je 37 bitů. Z toho můžeme usuzovat, že IPv4 strom je „košatější“ a tím pádem zahrnuje větší množství uzlů. Obecně z tohoto testu v žádném případě nelze odvodit žádné pravidlo. Na druhou stranu (vzhledem k tomu, že jde o reálné množiny) tento test ukázal, že stejně velké množiny skládající se z kratších a delších prefixů nemusí mít vždy paměťové nároky takové, jaké bychom čekali. Pokud bychom zvolili místo reálné množiny IPv6 prefixů množinu syntetickou, jejíž prefixy by byly rovnoměrně rozloženy po celém adresovém prostoru, byly by paměťové nároky algoritmu Trie pro tuto množinu mnohem vyšší než pro množinu IPv4.

4.1.3 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 3 bity v jednom taktu tedy 8 ukazatelů. Jedná se buď o ukazatele na další uzel trie nebo na ukazatele do množiny prefixů.

Pokud bychom tedy například pro 3 bity v jednom taktu volili velikost ukazatele 32 bitů, jeden uzel trie by pak vyžadoval $2^3 * 32 = 256$ bitů paměti.

V tabulce 4.5 jsou uvedeny paměťové nároky pro testovací množiny IPv6 prefixů. Použit byl algoritmus s optimalizací „leaf pushing“. Pokud zvyšujeme parametr n, snižuje se hloubka 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.

Jelikož se každý uzel skládá pouze z ukazatelů, lze zmenšením ukazatele i pro vyšší střídy dosáhnout vysoké úspory paměti, průměrně 63 %.

prefixů	n	hloubka	32-bit [B]	Nejmenší [B]	Ušetřeno
517	2	24	18 272	6 339	65,3 %
517	4	12	37 904	12 017	68,3 %
517	8	6	512 952	160 297	68,8 %
736	2	24	25 056	8 665	65,4 %
736	4	12	53 428	16 928	68,3 %
736	8	6	723 908	226 353	68,7 %
815	2	24	27 564	9 527	65,4 %
815	4	12	59 148	18 737	68,3 %
815	8	6	802 816	251 026	68,7 %
933	2	64	33 604	11 314	66,3 %
933	4	32	78 212	24 671	68,5 %
933	8	16	975 248	304 900	68,7 %
1 914	2	24	54 656	20 300	62,9 %
1 914	4	12	135 076	46 868	65,3 %
1 914	8	6	1 935 960	665 755	65,6 %
3 433	2	24	105 976	42 952	59,5 %
3 433	4	12	245 332	92 907	62,1 %
3 433	8	6	3 438 236	1 289 884	62,5 %
5 498	2	64	204 092	88 133	56,8 %
5 498	4	32	493 304	201 864	59,1 %
5 498	8	16	6 174 220	2 508 276	59,4 %

Tabulka 4.5: Paměťové nároky metody CPE na různých množinách IPv6 prefixů

4.1.4 Lulea Compressed Tries

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

Každý uzel pak 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. Průměrný počet ukazatelů, které uzly obsahují pro jednotlivé množiny prefixů, lze vidět v tabulce 4.6. V té jsou dále uvedeny celkové paměťové nároky této metody.

Z tabulky 4.6 je patrné, že pro střídu 2 obsahuje jeden uzel trie průměrně pouze 1,3 ukazatele (zatímco u algoritmu CPE by ze byly vždy 4). U střídy 8 je v každém uzlu přítomno v průměru 1,8 ukazatele, zatímco u CPE by zde bylo vždy 256 ukazatelů. Paměťové nároky při střídě 8 jsou však oproti střídě 4 několikanásobně vyšší. To je způsobeno tím, že každý uzel zde musí obsahovat 256bitovou bitmapu oproti 16bitové bitmapě u střídy 4.

Zavedení bitmapy zde způsobuje, že pro vyšší střídy nelze dosáhnout příliš velké úspory paměti zmenšením ukazatele.

prefixů	n	hloubka	32-bit [B]	nejmenší [B]	ušetřeno	ukazatelů/uzel
517	2	24	13 398	5 681	57,6 %	1,2
517	4	12	10 326	5 292	48,8 %	1,4
517	8	6	31 908	28 209	11,6 %	1,6
736	2	24	17 987	7 605	57,7 %	1,2
736	4	12	14 094	7 192	49,0 %	1,4
736	8	6	41 064	36 262	11,7 %	1,7
815	2	24	19 712	8 330	57,7 %	1,2
815	4	12	15 454	7 878	49,0 %	1,4
815	8	6	45 376	40 066	11,7 %	1,7
933	2	64	21 106	8 527	59,6 %	1,5
933	4	32	16 472	7 913	52,0 %	1,7
933	8	16	43 880	37 784	13,9 %	2,1
1 914	2	24	34 872	15 484	55,6 %	1,3
1 914	4	12	28 604	14 852	48,1 %	1,5
1 914	8	6	85 160	74 755	12,2 %	1,9
3 433	2	24	71 551	34 058	52,4 %	1,3
3 433	4	12	57 294	31 469	45,1 %	1,5
3 433	8	6	170 940	151 988	11,1 %	1,8
5 498	2	64	131 070	64 668	50,7 %	1,4
5 498	4	32	103 722	57 468	44,6 %	1,7
5 498	8	16	284 088	248 923	12,4 %	2,1

Tabulka 4.6: Paměťové nároky metody Lulea na různých množinách IPv6 prefixů

4.1.5 Tree Bitmap

Každý uzel musí obsahovat:

- 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 opět počet zpracovávaných bitů v jednom taktu. Při použití 32bitových reprezentací tedy jeden uzel vyžaduje 128 bitů paměti. V tabulce 4.7 jsou opět paměťové nároky této metody.

Jak je vidět, zvětšením parametru n se zmenší počet uzlů trie a tím i její maximální hloubka. To má pozitivní vliv na rychlost vyhledání, protože se sníží maximální počet náhodných přístupů do paměti. Na druhou stranu paměť potřebná pro reprezentaci dané množiny prefixů značně stoupá, protože se exponenciálně zvětšuje velikost bitmap. S tím souvisí také snižování rozdílu potřebné paměti mezi variantou s 32bitovým a nejmenším možným ukazatelem. Při zvolení parametru $n = 8$ se paměťové nároky sníží v průměru pouze o 6 %, protože naprostou většinu paměti potřebné k reprezentaci jednoho uzlu zabírají právě bitmapy (511 bitů bitmapy + maximálně 64 bitů ukazatele).

prefixů	n	uzlů	hloubka	32-bit [B]	nejmenší [B]	ušetřeno
517	3	1 465	16	14 467	6 593	54,4 %
517	5	931	9	14 780	9 659	34,6 %
517	8	829	6	59 584	55 025	7,7 %
736	3	1 997	16	19 720	8 987	54,4 %
736	5	1 216	9	19 304	12 768	33,9 %
736	8	1 059	6	76 116	70 424	7,5 %
815	3	2 172	16	21 449	10 046	53,2 %
815	5	1 332	9	21 146	13 986	33,9 %
815	8	1 170	6	84 094	77 805	7,5 %
933	3	1 887	42	18 634	8 492	54,4 %
933	5	1 190	25	18 891	12 495	33,9 %
933	8	1 088	16	78 200	72 352	7,5 %
1 914	3	3 230	16	31 896	15 343	51,9 %
1 914	5	2 206	9	35 020	23 715	32,3 %
1 914	8	2 153	6	154 747	143 713	7,1 %
3 433	3	7 205	16	71 149	36 025	49,4 %
3 433	5	4 701	9	74 628	51 711	30,7 %
3 433	8	4 367	6	313 878	292 589	6,8 %
5 498	3	12 868	42	127 072	67 557	46,8 %
5 498	5	7 582	25	120 364	84 350	29,9 %
5 498	8	7 027	16	505 066	471 687	6,6 %

Tabulka 4.7: Paměťové nároky metody Tree Bitmap na různých množinách IPv6 prefixů

4.1.6 Shape Shifting Tree

Tento algoritmus do každého uzlu přidává ještě tvarovou bitmapu, takže kompletní uzel obsahuje:

- Bitmapa vnitřních uzlů, velikost minimálně k bitů
- Bitmapa potomků, velikost minimálně $k + 1$ bitů
- Bitmapa tvarová, 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. V tabulce 4.8 jsou opět paměťové nároky metody.

Při porovnávání hodnot z tabulek tabulek 4.8 a 4.7 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.

prefixů	k	uzlů	hloubka	32-bit [B]	nejmenší [B]	ušetřeno
517	4	1 379	18	13 962	6 550	53,1 %
517	16	412	9	6 644	4 326	34,9 %
517	32	238	8	5 742	4 373	23,8 %
736	4	1 864	19	18 873	8 854	53,1 %
736	16	571	10	9 207	6 067	34,1 %
736	32	301	8	7 262	5 569	23,3 %
815	4	2 038	19	20 635	9 681	53,1 %
815	16	622	10	10 030	6 609	34,1 %
815	32	333	9	8 034	6 161	23,3 %
933	4	1 846	32	18 691	8 769	53,1 %
933	16	570	11	9 191	6 056	34,1 %
933	32	311	10	7 503	5 754	23,3 %
1 914	4	3 495	21	35 387	17 475	50,6 %
1 914	16	1 001	12	16 141	10 761	33,3 %
1 914	32	516	10	12 449	9 675	22,3 %
3 433	4	7 274	22	73 649	38 189	48,1 %
3 433	16	2 142	13	34 540	23 830	31,0 %
3 433	32	1 149	11	27 720	21 831	21,2 %
5 498	4	11 533	36	116 772	63 432	45,7 %
5 498	16	3 600	15	58 050	40 500	30,2 %
5 498	32	2 036	11	49 119	38 939	20,7 %

Tabulka 4.8: Paměťové nároky metody SST na různých množinách IPv6 prefixů

Zároveň si můžeme všimnout konkrétních hodnot, například pro reprezentaci největší množiny prefixů při maximální hloubce trie 15 potřebuje algoritmus SST alespoň cca 38 kB paměti. Naproti tomu algoritmus Tree Bitmap vyžaduje pro vytvoření trie o hloubce 16 pro stejnou množinu prefixů 470 kB.

I zde však platí, že při zvyšování parametru k klesá úspora při použití minimálního počtu bitů k reprezentaci ukazatelů. Opět je to dáno tím, že se mění poměr mezi pamětí potřebnou k uložení bitmap (tato se velikostí ukazatele nemění) a pamětí pro uložení ukazatelů (tuto lze ovlivnit).

4.1.7 Binární vyhledávání na intervalech

U tohoto algoritmu každý řádek tabulky obsahuje:

- Daný prefix patřičně rozšířený na počet bitů vyhledávané adresy (u IPv6 tedy 128 bitů)
- Ukazatel do tabulky prefixů

Nároky algoritmu pro testovací množinu prefixů jsou v tabulce 4.9. V té můžeme pozorovat, že zmenšením ukazatele nelze dosáhnout přílišné úspory paměti. Každý řádek tabulky totiž vždy musí obsahovat 128bitovou adresu, pouze ukazatel je možné z původních 32 bitů zmenšit. Pro největší množinu čítající necelých 5 500 záznamů je možné použít ukazatel

délky minimálně 13 bitů. Jeden řádek tabulky tímto tedy zmenšíme ze $128 + 32 = 160$ bitů na $128 + 13 = 141$ bitů.

prefixů	**řádků**	**32-bit [B]**	**nejmenší [B]**	**ušetřeno**
517	1032	20 640	17 802	13,8 %
736	1471	29 420	25 375	13,8 %
815	1629	32 580	28 100	13,8 %
933	1855	37 100	31 999	13,8 %
1 914	3828	76 560	66 512	13,1 %
3 433	6863	137 260	120 103	12,5 %
5 498	10877	217 540	191 707	11,9 %

Tabulka 4.9: Paměťové nároky metody BSI na různých množinách IPv6 prefixů

Lze předpokládat, že při použití s IPv4 by bylo možné ušetřit zmenšením ukazatele více. Pro stejně velkou množinu 32bitových IPv4 adres bychom jeden řádek tabulky zmenšili ze $32 + 32 = 64$ bitů na $32 + 13 = 45$ bitů, ušetřili bychom 30 % paměti oproti necelým 12 % ušetřeným na IPv6.

Maximální počet náhodných přístupů do paměti je dán algoritmem použitým pro vyhledávání. Pokud bychom použili klasické binární vyhledávání nad seřazeným polem, bude se počet přístupů do paměti rovnat maximálně $\log_2 n$, kde n je počet řádku tabulky. Tuto hodnotu je samozřejmě nutné zaokrouhlit nahoru na celé jednotky.

4.1.8 Binární vyhledávání na prefixech

Dle [11] každý uzel obsahuje:

- Délka prefixů uložených v tomto uzlu
- Samotné prefixy této délky
- K nim značky, zda existují delší prefixy, které začínají tímto kratším prefixem
- Oříznuté delší prefixy na délku prefixů uložených v tomto uzlu
- K nim odkazy na jejich LPM

Tabulka 4.10 opět shrnuje paměťové nároky. Uvedena je i maximální hloubka struktury pro jednotlivé množiny.

prefixů	uzlů	hloubka	32-bit [B]	nejmenší [B]	ušetřeno
517	13	3	18 241	11 423	37,4 %
736	15	3	25 496	16 116	36,8 %
815	15	3	28 030	17 784	36,6 %
933	29	4	32 217	20 393	36,7 %
1 914	18	4	54 375	38 933	28,4 %
3 433	24	4	134 505	85 018	36,8 %
5 498	36	5	218 955	141 559	35,3 %

Tabulka 4.10: Paměťové nároky metody BSP na různých množinách IPv6 prefixů

4.2 Porovnání algoritmů

Porovnání algoritmů bylo rozděleno na dvě části – zvlášť pro malé množiny prefixů a zvlášť pro velké. Protože největší množina IPv6 prefixů, která byla nalezena, čítá „pouze“ 5 500 záznamů, je za velkou považována pouze tato. Malé množiny v této práci obsahují maximálně 1000 záznamů.

Nad těmito množinami byly pro každý algoritmus sledovány 3 hodnoty – počet náhodných přístupů do paměti a paměť potřebná pro reprezentaci daných množin jak s 32bitovým ukazatelem, tak s nejmenším možným.

V porovnání se objeví algoritmy Trie, LC Trie, CPE s optimalizací „leaf pushing“ s parametrem $n = 4$, Lulea s parametrem $n = 4$, Tree Bitmap s parametrem $n = 5$, SST s parametrem $k = 32$, BSI a BSP.

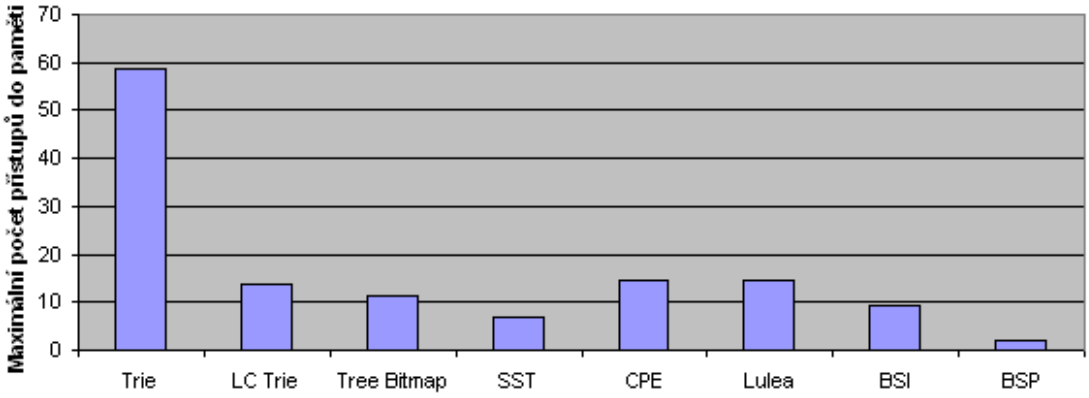
4.2.1 Malé množiny IPv6 prefixů

Pro porovnání algoritmů zde bylo použito celkem 15 menších množin IPv6 prefixů, které obsahovaly od 60 do 933 prefixů. U každého algoritmu byly ze všech použitých množin vypočítány průměry těchto hodnot. Ty můžeme vidět v tabulce 4.11.

algoritmus	přístupů do paměti	32-bit [B]	nejmenší [B]
Trie	58,7	9 088	6 147
LC Trie	13,7	3 032	2 296
CPE	14,7	12 037	10 328
Lulea	14,7	7 929	5 127
Tree Bitmap	11,1	5 985	1 131
SST	7,0	23 096	6 774
BSI	9,2	5 832	2 850
BSP	2,1	30 744	10 775

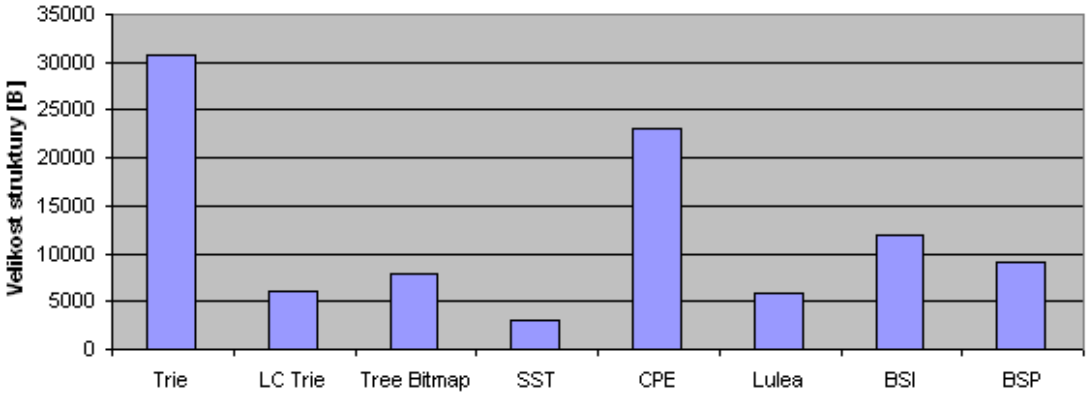
Tabulka 4.11: Průměrné hodnoty ze všech menších množin prefixů

V grafu na obrázku 4.2 je porovnání algoritmů podle maximálního počtu náhodných přístupů do paměti. Zde je vidět, jak v tomto parametru zaostává algoritmus Trie. Se svými téměř 59 přístupy je na tom jednoznačně nejhůře. Zároveň si můžeme všimnout vyrovna-



Obrázek 4.2: Maximální počet náhodných přístupů do paměti pro jednotlivé algoritmy

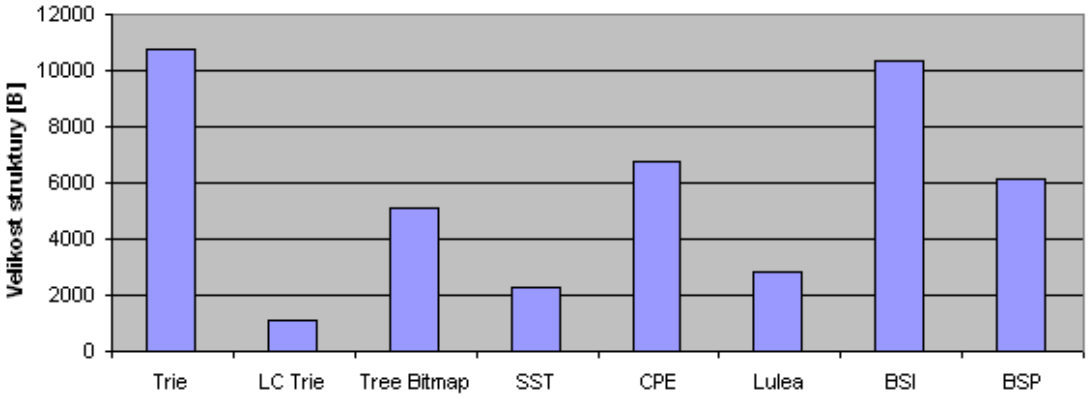
nosti algoritmů CPE a Lulea. Byla použita stejná střída, proto je hloubka stromu u obou algoritmů stejná. Lulea Compressed Tries má nižší pouze paměťové nároky. Algoritmus LC Trie vyžaduje v průměru téměř 14 přístupů, je však nutno počítat s tím, že je třeba ještě projít tabulku prefixů. Tree Bitmap s parametrem $n = 5$ vyžaduje v průměru až 11 přístupů do pamětí, zatímco SST s parametrem $k = 32$ pouze 7. Metoda BSI vyžaduje v průměru 9 přístupů, zatímco BSP pouze 2.



Obrázek 4.3: Paměťové nároky pro jednotlivé algoritmy při použití 32bitového ukazatele

Na obrázku 4.3 je graf znázorňující průměrnou velikost struktury pro reprezentaci menších testovacích množin prefixů, zde při použití pevného 32bitového ukazatele. Opět vidíme, že algoritmus Trie je na tom nejhůře. Velké paměťové nároky má také algoritmus CPE. To je dáno tím, že ukládá mnoho ukazatelů v každém uzlu (zde pro parametr $n = 4$ celkem 16 ukazatelů). Oproti algoritmu Lulea, který se liší pouze úspornější prací s ukazately, vyžaduje CPE téměř čtyřikrát více paměti. SST dosahuje i zde výrazně lepších výsledků než Tree Bitmap. LC Trie je se svými paměťovými nároky mezi těmito algoritmy.

Další graf (viz obr. 4.4) shrnuje opět průměrné paměťové nároky nad všemi množinami,



Obrázek 4.4: Paměťové nároky pro jednotlivé algoritmy při použití nejmenšího ukazatele

ovšem zde při použití nejmenšího možného ukazatele. Vzhledem k nízké úspoře, kterou je možné dosáhnout u metody BSI (patrné i z tabulky 4.9) a poměrně velké úspoře u algoritmu Trie (viz tab. 4.1), se zde tyto algoritmy téměř rovnají. Kvůli nutnosti udržovat bitmapy, které se tímto neredukují, se zmenšil rozdíl jak mezi algoritmy CPE a Lulea, tak mezi Tree Bitmap a SST. Lulea a SST však stále vyžadují méně než polovinu paměti oproti algoritmům CPE a Tree Bitmap. Největší úspory se dosáhlo u algoritmu LC Trie, který zmenšil paměťové nároky z průměrných téměř 6 kB na 1 kB, tedy o 80% a se stal nejúspornějším.

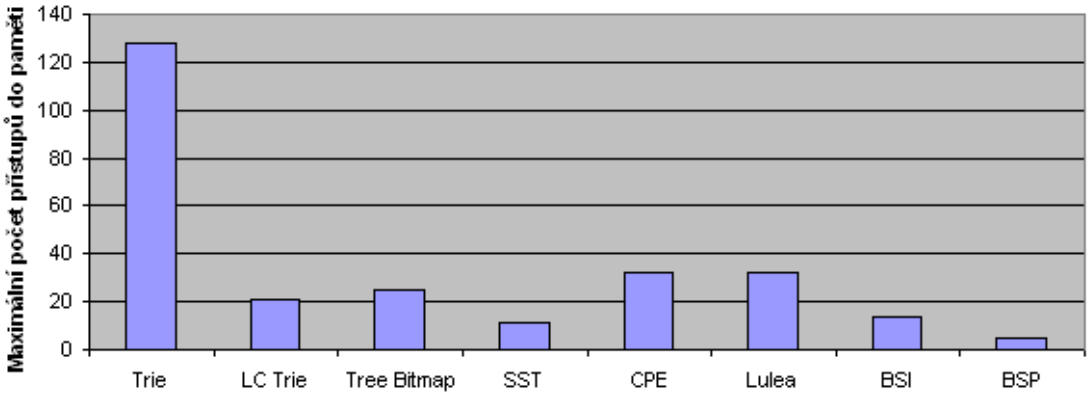
4.2.2 Velké množiny IPv6 prefixu

Jak již bylo zmíněno, do testů v této kapitole byla zahrnuta pouze jedna množina IPv6 prefixů, protože více množin podobné velikosti se nepodařilo najít. Tato testovací množina pochází z [3] a obsahuje téměř 5 500 záznamů. Vlastnosti výsledných struktur sestavených pomocí různých LPM algoritmů jsou v tabulce 4.12.

algoritmus	**přístupů do paměti**	**32-bit [B]**	**nejmenší [B]**
Trie	128	473 820	222 103
LC Trie	21	98 244	23 538
CPE	32	493 304	201 865
Lulea	32	103 722	57 468
Tree Bitmap	25	120 364	84 350
SST	11	49 119	38 939
BSI	14	217 540	191 707
BSP	5	218 955	141 559

Tabulka 4.12: Hodnoty z velké množiny prefixů

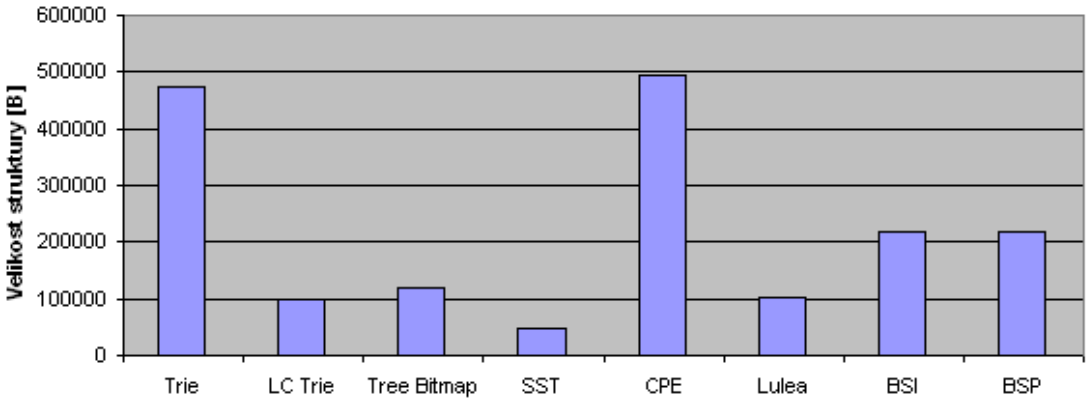
Na obrázku 4.5 jsou z tabulky 4.12 vyneseny do grafu maximální počty náhodných přístupů do paměti pro jednotlivé algoritmy. Podle očekávání je na tom opět nejhůře algoritmus Trie se svými 128 přístupy do paměti. CPE a Lulea jsou vyrovnané – podobně



Obrázek 4.5: Maximální počet náhodných přístupů do paměti pro jednotlivé algoritmy

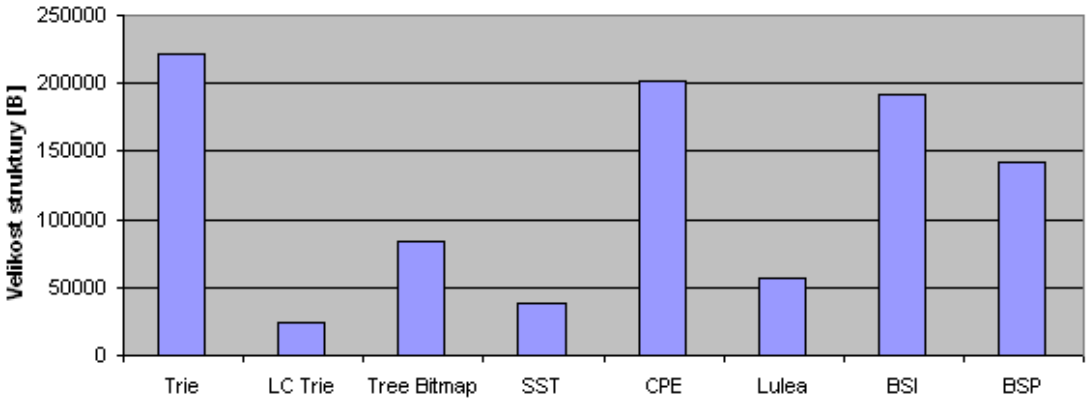
jako v případě malých množin – kvůli tomu, že je použita stejná střída. SST (11 přístupů) opět předčí Tree Bitmap (25 přístupů). LC Trie zde (narozdíl od malých množin prefixů) vychází také lépe než Tree Bitmap. To je pravděpodobně způsobeno účinnější aplikací „level compression“ při použití s větším množstvím prefixů. V maximálním počtu přístupů do paměti je na tom opět nejlépe algoritmus BSP (5 přístupů).

Graf na obrázku 4.6 ukazuje paměťové nároky jednotlivých algoritmů při použití 32bitového ukazatele. Algoritmus CPE v tomto testu vyžadoval dokonce ještě více paměti než Trie. BSI i BSP potřebovaly k uložení struktury shodně více než 200 kB. Velká testovací množina totiž obsahovala větší množství prefixů různých délek, a proto byly paměťové nároky metody BSP (ve srovnání s menšími množinami) poměrně velké. Algoritmu SST stačila opět necelá polovina paměti oproti algoritmu Tree Bitmap. Lulea vyžadovala pouze zlomek paměti oproti CPE. LC Trie na tom byla v porovnání s ostatními v tomto ohledu stejně jako u menších množin.



Obrázek 4.6: Paměťové nároky pro jednotlivé algoritmy při použití 32bitového ukazatele

V posledním grafu (viz obr. 4.7) jsou paměťové nároky při použití nejmenšího možného ukazatele. Jelikož (dle tabulek 4.1 a 4.5) lze u algoritmu CPE uspořit zmenšením ukazatele více než u Trie, nejvíce paměti spotřeboval algoritmus Trie. CPE a BSI byly poměrně vyrovnané. Nejlépe dopadl algoritmus LC Trie, kterému stačilo na reprezentaci 5 500 prefixů z testovací množiny pouze 23 kB paměti.



Obrázek 4.7: Paměťové nároky pro jednotlivé algoritmy při použití nejmenšího ukazatele

Kapitola 5

Závěr

Práce rozšiřuje knihovnu Netbench o metodu LC Trie. Dále byly doplněny skripty pro testování dostupných LPM algoritmů a reálné množiny IPv6 prefixů, které sloužily jako základ při následném porovnávání vhodnosti jednotlivých metod pro použití s protokolem IPv6.

Jak se ukázalo, každý algoritmus je vhodný pro jiné případy. Například algoritmy LC Trie, SST a Lulea Compressed Trie slibují poměrně malé paměťové nároky. Algoritmus BSP na druhou stranu vystačí s malým počtem náhodných přístupů do paměti, jeho paměťové nároky jsou však v porovnání s dříve zmíněnými vyšší. Algoritmy Trie a CPE se zdají být pro praktické využití nevhodné.

V kapitole 4 je vidět, jak u algoritmů CPE, Lulea, Tree Bitmap, SST lze změnou parametrů upravovat poměr potřebné paměti a počtu přístupů do paměti. Pokud zvolíme parametr příliš vysoký, rostou paměťové nároky neakceptovatelně (viz například algoritmus Tree Bitmap pro $n \geq 8$, případně CPE pro $n \geq 8$).

Zároveň si v této kapitole můžeme všimnout, jak lze u jednotlivých algoritmů šetřit pamětí použitím nejmenšího možného ukazatele. U většiny metod s bitmapou (Lulea, Tree Bitmap, SST) lze při použití vyšších hodnot parametrů tímto pochopitelně dosáhnout mnohem menší procentuální úspory paměti než při parametrech nižších.

Práci je možné dále rozšiřovat v mnoha směrech. Nabízí se vývoj a implementace dalších LPM algoritmů, případně optimalizace těch současných. Dá se také předpokládat, že během několika let se množiny IPv6 prefixů podstatně rozrostou, proto by bylo vhodné provést jejich aktualizaci a testy z části 4.2.2 provést i na mnohem větších množinách.

Literatura

[1] Accelerated Network Technologies [online]. `http://merlin.fit.vutbr.cz/ant/`, [cit. 2011-05-02].

[2] BGP Link, BGP Routing Info [online]. `http://www.bgp-link.de/`, [cit. 2011-05-02].

[3] BGP Reports [online]. `http://bgp.potaroo.net/index-bgp.html`, [cit. 2011-05-02].

[4] SixXS, IPv6 Deployment and IPv6 Tunnel Broker [online]. `http://www.sixxs.net/`, [cit. 2011-05-02].

[5] 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.

[6] 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.

[7] 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.

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

[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] Suchodol, J.: Algoritmy pro vyhledání nejdelšího shodného prefixu. Bakalářská práce, FIT VUT v Brně, 2010.

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

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

Příloha A

Obsah DVD

Přiložené DVD obsahuje celý repozitář projektu Netbench a elektronickou verzi této zprávy.