VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

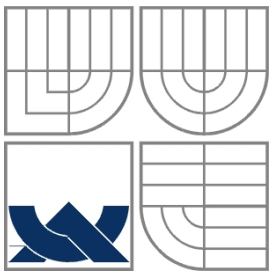
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ALGORITMY PRO VYHLEDÁNÍ NEJDELŠÍHO SHODNÉHO PREFIXU

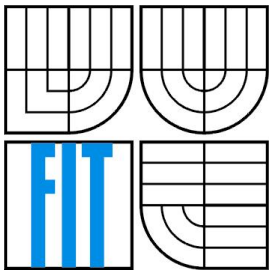
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN SKAČAN
AUTHOR

BRNO 2010



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

MARTIN SKAČAN

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ TOBOLA

BRNO 2010

Abstrakt

Tato práce se zabývá algoritmy pro vyhledání nejdelšího shodného prefixu (longest prefix match – LPM), což je klíčová operace při klasifikaci paketů a směrování v počítačových IP sítích. Je uvedena potřebná teorie a rozbor vybraných algoritmů – Trie, Tree Bitmap, Shape Shifting Tree a Multi-Match. Tyto metody byly detailně popsány a implementovány v programovacím jazyce Python. Nad implementovanými algoritmy byly provedeny testy a simulace pro určení jejich praktických paměťových nároků s cílem identifikovat nejvhodnější metodu pro množiny prefixů o velikosti desítek až tisíců pravidel.

Abstract

This bachelor's thesis deals with the algorithms for longest prefix match (LPM), which is the basic operation of the packet classification and of the routing in the IP computer networks. It is introduced the necessary theory and the analysis of the chosen algorithms – Trie, Tree Bitmap, Shape Shifting Tree and Multi-match. These methods were closely described and they were implemented in the programming language Python. Within the implemented algorithms were performed tests and simulations considering their memory demands with the aim to identify the best method for prefix collections about the size of tens to thousands rules.

Klíčová slova

nejdelší shodný prefix, LPM, algoritmus, trie, tree bitmap, shape-shifting tree, multimatch, IP

Keywords

longest prefix match, LPM, algorithm, trie, tree bitmap, shape-shifting tree, multimatch, IP

Citace

Skačan Martin: Algoritmy pro vyhledání nejdelšího shodného prefixu, bakalářská práce, Brno, FIT VUT v Brně, 2010

Algoritmy pro vyhledání nejdelšího shodného prefixu

Prohlášení

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

……
Martin Skačan
13. května 2010

Poděkování

Děkuji vedoucímu své bakalářské práce Ing. Jiřímu Tobolovi za vedení, trpělivost, připomínky a nápady.

Obsah

1 Úvod

V súčasnosti žijeme v modernej a rýchlej dobe, kedy každodenne prichádzame do kontaktu z najrôznejšími informačnými technológiami. Naše nároky na ich kvalitu, efektivitu a rýchlosť sa samozrejme neustále zvyšujú. Tento trend je celkom prirodzený a je nezastaviteľný. Jednou z napredujúcich IT oblastí sú i počítačové siete a internet.

Internet je veľmi silný pojem, ktorý nie je nutné nikomu prestavovať. Presadil sa už vo všetkých sférach nášho života. Spolu s jeho rozširovaním a rapídnym zvyšovaním rýchlosti však prichádzajú aj problémy. Je potrebné zaistiť bezpečnosť a kvalitu poskytovaných služieb. Na prenosových linkách sa musí riešiť smerovanie, filtrovanie, monitorovanie dátových tokov a mnoho iných výpočtovo náročných úloh. Každej tejto operácii musí predchádzať klasifikácia paketov. Tá prebieha na základe informácií z hlavičky paketu s využitím algoritmu na vyhľadanie najdlhšieho zhodného prefixu. Je to časovo kritická operácia, ktorá musí byť vykonaná v časovom úseku desiatok nanosekúnd. Ďalším problémom je prechod na nový štandard IPv6. Adresný priestor v súčasnosti používanej verzie IPv4 bude už čoskoro vyčerpaný. Preto je prechod na novú verziu nevyhnutný. Tá však prináša jednak vyššiu réžiu a jednak obrovský adresný priestor, ktorý kladie veľké nároky na vyhľadávacie algoritmy.

Bežné procesory osobných počítačov nie sú schopné udržať krok s ohromným rozvojom počítačových sietí. Preto ich úlohu preberajú špecializované hardwarové zariadenia. Ich prednosťou je najmä optimalizácia na konkrétny problém, ako aj možnosť paralelného spracovania dát. Samotné zvyšovanie výkonu hardwarových zariadení však nie je postačujúce na odstránenie spomenutých problémov. Výkonný hardware je samozrejme nepostrádateľný základ, bez ktorého by bola táto situácia neriešiteľná. Na rýchlosť spracovania paketov v sieťach sú však kladené tak vysoké nároky, že je vzácny každý výpočtový krok zariadenia. Preto je nutné zaoberať sa tiež otázkou vhodnosti používaných algoritmov. Neustále musíme hľadať nové algoritmy a optimalizovať ich na dosiahnutie čo najlepších výsledkov.

Táto práca sa zaoberá práve porovnaním jednotlivých algoritmov na vyhľadávanie najdlhšieho zhodného prefixu (Longest prefix match - LPM). Neexistuje univerzálny dokonalý algoritmus, ktorý by spĺňal všetky požiadavky. Samozrejme by sme chceli, aby bol algoritmus čo najrýchlejší, to znamená minimalizovanie počtu krokov potrebných na získanie výsledku. Druhým hlavným cieľom je udržať jeho pamäťové nároky v akceptovateľnej miere. Tieto dve požiadavky stoja v zásade vždy proti sebe, teda zrýchlenie algoritmu býva na úkor vyšších pamäťových nárokov a podobne. Preto nie je jednoduché nájsť optimálnu cestu. Existuje množstvo LPM algoritmov, ktoré sú dobre známe a sú popísané v odborných publikáciách [2, 3, 11]. Väčšina z nich je založená na stromovej štruktúre. Ja som sa zameral na 3 stromové algoritmy – trie, tree bitmap a shape shifting tree. K tomu som pridal jeden vlastný experimentálny algoritmus založený na hashovacích tabuľkách – multi-match. Ten by mohol mať výraznú výhodu hlavne v hardwarovej implementácii pre možnosť paralelizácie výpočtov.

Pred skutočným nasadením a používaním algoritmu v hardwari je potrebné vykonať veľké množstvo testov, simulácií a experimentov. Tie sa vykonávajú spravidla v softwarovej podobe (hlavne v úvodných fázach vývoja) kvôli jednoduchosti a zníženiu nákladov. Neustále však musíme mať v povedomí budúcu hardwarovú implementáciu. Pretože aj niektoré zdanlivo jednoduché úlohy sú hardwarovo ťažko implementovateľné a vo výsledku by stáli množstvo komplikovanej prídavnej

logiky. V mojej práci som vykonal viacero testov vybraných algoritmov zameraných práve na ich rýchlosť a pamäťovú náročnosť.

Práca je hierarchicky usporiadaná do niekoľkých kapitol a jej členenie je nasledovné. V druhej kapitole sa nachádza potrebný teoretický úvod do problematiky, kde je popísaná architektúra počítačových sietí, ako tiež problematika klasifikácie paketov, smerovania a filtrovania dátových tokov. Tretia kapitola mojej práce obsahuje detailný popis vybraných LPM algoritmov. Štvrtá kapitola sa stručne venuje objektovému návrhu a implementácii metód. Výsledky testov a porovnanie algoritmov sa nachádzajú v piatej kapitole. Na záver sú v poslednej kapitole zhrnuté a vyhodnotené dosiahnuté výsledky mojej práce a diskutované možnosti pokračovania práce.

2 Architektúra a operácie v IP sieťach

Problematika spojená s vyhľadávaním najdlhšieho zhodného prefixu sa týka takmer všetkých operácií vykonávaných v počítačových sieťach. Je to základný a prvotný krok, ktorý je súčasťou klasifikácie paketov. Až na základe výsledku tohto kroku sa ďalej rozhoduje, ako bude vykonávaná operácia ďalej pokračovať. Počítačové siete sa rozdeľujú do mnohých kategórií, v závislosti na používanej architektúre, technológii alebo podľa definovaných komunikačných protokolov. V súčasnosti dominantné postavenie držia siete postavené na architektúre TCP/IP.

V nasledujúcich podkapitolách sa stručne venujem počítačovým sieťam založených práve na tejto architektúre. Sú uvedené potrebné základy a kľúčové prvky IP sietí, ktoré sú nevyhnutné k ďalšiemu výkladu. Na začiatku sa nachádza obecný popis architektúry a dátové komunikačné jednotky (IP datagramy), z ktorých získavame vstupné údaje pre klasifikáciu. Ďalej už popisujem konkrétny princíp fungovania klasifikácie a význam LPM v rámci klasifikácie, ako tiež hlavné oblasti jej využitia. Informácie boli čerpané hlavne z [9, 10].

2.1 Architektúra a model TCP/IP

Pre popis počítačových sietí sa väčšinou používajú vrstvové modely, ktoré vo všeobecnosti odlišujú minimálne 3 hlavné vrstvy komunikácie:

- vrstva pre prenos signálu
- zaistenie spoľahlivého prenosu
- aplikačná vrstva

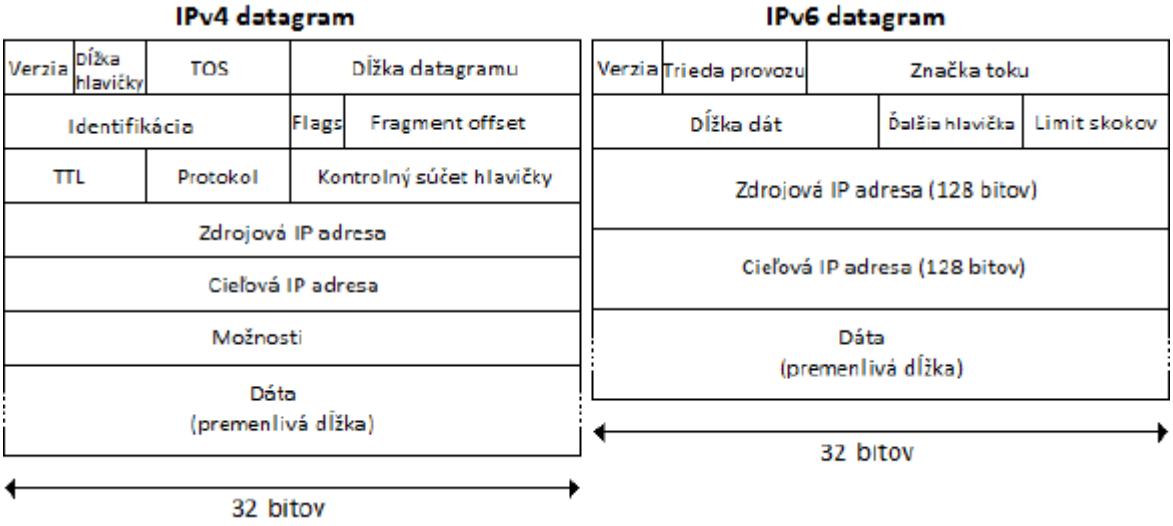
V roku 1987 definovala Medzinárodná štandardizačná organizácia – ISO (International Standards Organization) model počítačových sietí OSI (Open Systems Interconnect Reference Model). Ten sa v praxi síce nepoužíva, no bol prijatý ako referenčný model a vychádzajú z neho takmer všetky ostatné modely. Tento model je dobre známy a často sa objavuje publikáciách zaoberajúcich sa počítačovými sieťami [10]. Najpoužívanejším modelom v súčasnosti, ktorý je zároveň štandardom internetu je model TCP/IP. V tabuľke 2.1 je znázornené porovnanie modelov ISO/OSI a TCP/IP spolu so stručným popisom jednotlivých vrstiev. Vidíme, že model TCP/IP je výrazne jednoduchší, čím odstraňuje hlavné nedostatky komplikovanej štruktúry ISO/OSI, ktorá bránila jej implementácii.

ISO/OSI	TCP/IP	Popis
Aplikačná vrstva	Aplikačná vrstva	Komunikácia medzi jednotlivými aplikáciami
Prezentačná vrstva		
Relačná vrstva		
Transportná vrstva	Transportná vrstva	Logické spojenie medzi procesmi
Sieťová vrstva	Sieťová vrstva	Smerovanie a adresovanie v sieti
Linková vrstva	Vrstva sieťového rozhrania	Prístup k fyzickému médiu, IP datagramy balí na rámce
Fyzická vrstva		

Tabuľka 2.1: Porovnanie modelu ISO/OSI a TCP/IP

2.2 IP datagram

Pri klasifikácii paketov sú potrebné údaje z jeho hlavičky. Jedná sa predovšetkým o zdrojové a cieľové IP adresy a teda nás zaujíma najmä sieťová vrstva internetového modelu. V súčasnosti najpoužívanejším protokolom sieťovej vrstvy je IPv4 protokol [8]. Paralelne s ním sa dnes už používa nová verzia IPv6 [1]. Porovnanie datagramov oboch verzií sa nachádza na obrázku 2.1.



Obrázok 2.1: Porovnanie datagramov IPv4 a IPv6

Hlavným nedostatkom súčasnej verzie je malý adresný priestor. Pomocou IP adresy dokážeme jednoznačne identifikovať a adresovať koncové zariadenie v sieti (obvykle počítač). Vo verzii 4 je IP adresa 32-bitová. To nám poskytuje teoreticky 2^{32} jedinečných adries. Niektoré adresy sú navyše rezervované a nie je možné ich využívať na bežné adresovanie. I s využitím NATu a iných techník budú IP adresy verzie 4 v niekoľkých najbližších rokov úplne vyčerpané. Verzia 6 naproti tomu obsahuje až 128-bitovú IP adresu. To nám dáva nepredstaviteľný adresný priestor – teoreticky 2^{128} adries, čo je približne $5x10^{28}$ IP adries pre každého človeka na Zemi. Napriek tomu sa reálne nasadenie novej verzie presadzuje len veľmi pomaly [7]. IPv6 navyše odstraňuje mnohé nedostatky verzie 4 a poskytuje nové možnosti. Medzi inými len spomeniem niektoré výrazné prednosti:

- zvýšená bezpečnosť
- podpora kvality služieb (QoS)
- zjednodušené prečíslovanie pri zmene ISP
- podpora mobilných zariadení (multihoming)
- automatická konfigurácia
- jednoduchšia hlavička pevnej veľkosti (40 bytov)

IP adresa verzie 4 sa zapisuje ako štyri 8-bitové decimálne čísla oddelené bodkou (napr. 192.168.1.2). IPv6 má formát osem 16-bitových hexadecimálnych čísel oddelených dvojbodkou (napr. 2001:0DB8:0000:0000:0000:0000:1428:57AB). Za sebou nasledujúce nuly môžeme vynechať,

pretože adresu je možné dopočítať na plných 128 bitov. Uvedenú adresu je teda možné skrátene zapísať ako 2001:0DB8::1428:57AB. IP adresy sa často zapisujú vo forme prefixu (192.168.0.0/16, 2001:200::/32). Tento zápis je najčastejšie reprezentovaný ako rozdelenie adresy na adresu siete a adresu koncového zariadenia. Pri vyhľadávaní najdlhšieho zhodného prefixu využívame práve s túto formu zápisu a určuje nám počet platných bitov IP adresy, s ktorými budeme pracovať.

Nie všetky údaje z hlavičky IP datagramu sa využívajú pri klasifikácii, niektoré sa môžu ignorovať, pretože sú pre účely klasifikácie nepodstatné. Naproti tomu často využívame aj hodnoty z transportnej vrstvy (hlavne čísla TCP alebo UDP portov), prípadne z iných vrstiev modelu.

2.3 Princíp klasifikácie paketov

Prvotným a základným krokom pri práci s každým paketom je jeho klasifikácia [9]. Tá je podstatná hlavne pri úlohách ako je smerovanie, filtrovanie, zaistenie bezpečnosti sietí alebo pri sledovaní a manažmente sietí, s čím súvisí tiež zabezpečenie kvality služieb (QoS).
Vstupnými dátami pre klasifikáciu sú:

- množina klasifikačných pravidiel
- údaje z hlavičky paketu

Príkladom klasifikačného pravidla môže byť nasledovný zápis:

```
34 block on 1 proto tcp from 128.0.0.0/1 to any
```

Každé pravidlo teda pracuje s niekoľkými údajmi z hlavičky a všetky pravidlá sú v množine usporiadané podľa priority. Podstatou klasifikácie je vyhľadanie jedného pravidla, ktoré vyhovuje danému paketu. V prípade, že vyhovuje viacero pravidiel, je vybrané to, ktoré má vyššiu prioritu. Po tomto kroku máme jednoznačné klasifikačné pravidlo, na základe ktorého sa rozhodneme ako budeme ďalej pracovať s týmto paketom. Konkrétna činnosť, ktorá bude vykonaná závisí na úlohe, pre ktorú bola vykonaná klasifikácia. Najčastejšie je to zahodenie/prepustenie paketu, preposlanie paketu na ďalší smerovač, zaradenie paketu do dátového toku, logovanie a podobne.

Problém klasifikácie je možné riešiť mnohými spôsobmi. Vznikli dokonca špecializované hardwarové zariadenia (asociatívna pamäť TCAM), ktoré sa snažia ponúknuť vyšší výkon. Na druhej strane existuje snaha neustále zdokonaľovať a nachádzať nové algoritmické metódy riešenia tejto úlohy, ktoré dokážeme implemenetovať na bežne dostupných hardwarových prostriedkoch. Často využívaný algoritmický prístup k riešeniu klasifikácie je dekompozičná metóda [9]. Táto metóda sa skladá z dvoch hlavných krokov:

1. vyhľadanie najdlhšieho zhodného prefixu pre každé pole z hlavičky paketu
2. vyhľadanie klasifikačného pravidla na základe výsledkov prvého kroku

Dekompozičné metódy vykazujú veľký potenciál pre ďalší výskum do budúcnosti. Hlavnou výhodou je, že jednotlivé polia hlavičky paketu na sebe nezávisia, a preto je možné všetky z nich prehľadávať paralelne. To nám poskytuje značnú časovú úsporu. Rýchlosť samotnej klasifikácia teda do veľkej miery závisí na tom, ako rýchlo dokážeme vyhľadať najdlhší zhodný prefix. A práve hľadanie najoptimálnejšej LPM metódy je jeden z cieľov mojej práce.

2.4 Smerovanie a filtrovanie

V predchádzajúcich častiach tejto práce už boli stručne naznačené rôzne oblasti využitia klasifikácie. Dve z nich, ktoré sú snáď najpodstatnejšie a najčastejšie využívané sú smerovanie a filtrovanie dátových paketov v počítačovej sieti [10].

Pri komunikácii dvoch počítačov v sieti je potrebné zabezpečiť prenos dátových paketov medzi nimi. Nie je jednoznačne dané, akou cestou budú putovať a dokonca je možné, že každý paket sa dostane do cieľa úplne odlišnou cestou. Ideálne samozrejme je, aby bola cesta od zdroju k cieľu čo najrýchlejšia. Túto úlohu rieši smerovanie paketov v sieti. Základný princíp smerovania je nasledovný. Zdrojový počítač odošle paket na predvolenú bránu (prvý smerovač). Ak sa cieľový počítač nachádza v rovnakej podsieti ako aktuálny smerovač, je mu paket predaný priamo. Inak smerovač prepošle paket na jeden zo susedných smerovačov a proces sa opakuje. Zostáva otázka, na základe čoho sa smerovač rozhoduje, ktorú z uvedených možností vykoná, prípadne ktorému smerovaču paket prepošle. Každý smerovač obsahuje štruktúru potrebných dát - smerovaciu tabuľku. V nej si uchováva predovšetkým IP adresy vo forme prefixu a ku každej z nich tzv. next-hop adresu. Smerovač pri rozhodovaní vyhľadá cieľovú adresu vo svojej smerovacej tabuľke a následne paket pošle na uvedenú next-hop adresu. V prípade, že cieľovej adrese vyhovuje viac prefixov, vyberie sa ten konkrétnejší. Jedná sa teda o klasické vyhľadanie najdlhšieho zhodného prefixu. Veľkosť smerovacej tabuľky je rôzna na každom smerovači a závisí na veľkosti siete, v ktorej sa nachádza. U veľkých poskytovateľoch sú to obrovské štruktúry, ktoré môžu obsahovať až stotisíce záznamov.

Ďalším využitím klasifikácie je filtrovanie paketov. Je to jedna z najjednoduchších a najstarších foriem zabezpečenia sietí. Túto funkciu plnia takzvané firewally. Firewall môže byť na úrovni softwaru, alebo často ide o hardwarové zariadenie. Je to filter, ktorý ochraňuje konkrétne koncové zariadenie, alebo celú lokálnu sieť. Aby bol firewall schopný poskytnúť ochranu, musí byť správne nakonfigurovaný. Konfigurácia spočíva predovšetkým v nastavení množiny pravidiel, ktorou sa bude riadiť. Firewall teda podobne ako smerovač obsahuje svoju lokálnu tabuľku pravidiel usporiadaných podľa priority. Na rozdiel od smerovačov sa však nepodieľa na smerovaní, ale hrá len úlohu akéhosi medzičlánku, ktorý zachytáva prichádzajúce a odchádzajúce pakety. Pre každý paket vyhľadá v tabuľke prvé pravidlo, ktoré mu odpovedá, a toto pravidlo uplatní. Vyhľadanie pravidla je v podstate klasifikácia paketu – vyhľadanie najdlhšieho zhodného prefixu pre zdrojové a cieľové IP adresy a porty. Oproti smerovačom majú však jednoduchšiu funkciu a rozhodujú sa spravidla medzi dvomi možnosťami. Buď je paket nežiaduci a v tom prípade ho firewall ignoruje – paket je „zahodený“, alebo je povolený a firewall ho prepustí a paket putuje ďalej k cieľu. Tento popis je samozrejme značne zjednodušený a je uvedený len na základnú predstavu fungovania filtrovania paketov v sieti. V skutočnosti sú to omnoho sofistikovanejšie zariadenia, ktoré vykonávajú mnoho ďalších komplexnejších a inteligentnejších úloh. Pretože firewally bývajú nasadené na lokálnych sieťach, je jasné, že počet pravidiel, ktoré využívajú pri klasifikácii je spravidla mnohonásobne nižší ako je to pri smerovačoch.

V mojej práci som sa zameral na problematiku filtrovania. Z toho vychádzajú aj použité testovacie množiny, ktorých veľkosť bola rádovo 100-1000 prefixov.

3 Algoritmy LPM

V predchádzajúcej kapitole bol vysvetlený princíp klasifikácie a význam LPM (Longest prefix match) metód. Teraz sa bližšie pozrieme na spôsob vyhľadávanie prefixov a podrobne popíšem vybrané algoritmy. Vychádzal som prevažne zo zdrojov [2, 3, 11, 12].

Na vstupe každej LPM metódy potrebujeme množinu prefixov, v ktorej sa bude vyhľadávať. Dĺžka prefixov je rôzna, pre IPv4 sa pohybuje v rozsahu 0 – 32 bitov. Algoritmy pracujú s prefixami spravidla v binárnej podobe. V spracovaní IPv4 a IPv6 adresy teda v princípe nie je žiadny rozdiel. Jediný rozdiel sú len vyššie nároky na pamäť a výkon v prípade novej verzie.

Ďalej máme na vstupe konkrétnu hodnotu (IP adresu), ktorú chceme v množine vyhľadať. Výsledkom je jeden prefix z množiny, ktorý odpovedá vstupnej hodnote a je najdlhší. Prípadne môže byť výsledkom množina všetkých prefixov, ktoré odpovedajú vstupnej hodnote. V tomto prípade musia byť prefixy zostupne zoradené podľa ich dĺžky. Najdlhší zhodný prefix je potom prvý prefix z tejto množiny. V mojej práci je použitá druhá varianta, ktorej výhodou je napríklad presnejšie overenie správnosti danej metódy.

Príklad:

- vstupná množina prefixov: *, 1*, 0*, 101*, 1011*, 1010*, 10111*
- vstupná hodnota: 10110
- výsledok: 1011* (resp. 1011*, 101*, 1*, *)

Spôsobov akým je uložená množina prefixov v pamäti a spôsobov vyhľadania prefixu v množine je mnoho a práve na tom záleží celková efektivita algoritmu. Prvým a najtriviálnejším spôsobom, ktorý by nás asi napadol je sekvenčné vyhľadávanie. Tento naivný algoritmus prechádza sekvenčne každý prefix z množiny a porovná ho so zadanou IP adresou. Súčasne si uchováva najdlhší doposiaľ nájdený prefix. Tento spôsob je samozrejme správny, no je neefektívny, pretože na získanie výsledku musíme porovnať každú hodnotu vo vstupnej množine. Čas výpočtu je teda priamo úmerný veľkosti vstupnej množiny. Využíva sa napríklad na overenie správnosti zložitejších metód alebo pri veľmi malých množinách. Ďalej si predstavíme niekoľko sofistikovanejších metód, ktoré sú väčšinou postavené na stromovej štruktúre.

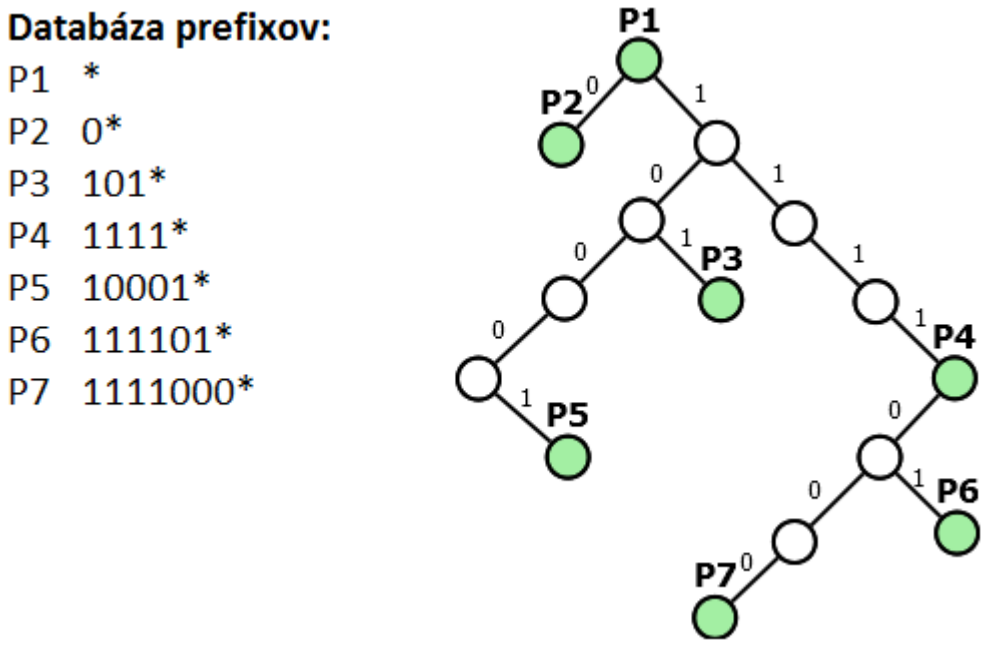
3.1 Trie

Trie je základný stromový LPM algoritmus [2]. Názov tejto metódy pochádza z anglického slova retrieval. Množina prefixov je uložená v podobe špeciálneho binárneho stromu. Je to jednoduchá štruktúra, ktorá kóduje prefixy priamo vo svojej konštrukcii. Jeden uzol stromu reprezentuje jeden bit IP adresy. Koreňový uzol predstavuje prefix * ~ 0.0.0.0/0, ktorý vyhovuje ľubovoľnej IP adrese. Každý uzol stromu obsahuje 2 ukazatele na svoje synovské uzly. Ľavý ukazateľ znamená bit 0, pravý ukazateľ zase bit 1 v IP adrese. Uzol potom už nesie len informáciu o tom, či obsahuje platný prefix a prípadne jeho hodnotu alebo ukazateľ na tento prefix. Príklad jednoduchého prefixového stromu je zobrazený na obrázku 3.1.

Po vytvorení celého prefixového stromu je princíp vyhľadávania veľmi jednoduchý. Začíname v koreni stromu a postupne prechádzame vstupnú IP adresu po jednotlivých bitoch od

najvýznamnejšieho (MSB) po najmenej významný bit adresy (LSB). V každom kroku sa podľa aktuálneho bitu rozhodujeme, ktorou vetvou stromu budeme pokračovať (0 – vľavo, 1 - vpravo). Postup sa opakuje dokým nespracujeme všetky bity vstupnej adresy alebo dokým neskončíme v jednom z listov stromu a teda nemáme kam pokračovať. V priebehu výpočtu si uchovávame najdlhší doposiaľ nájdený prefix, prípadne si uchovávame všetky platné prefixy na danej ceste a najdlhší je potom posledný nájdený.

Tento algoritmus poskytuje veľmi jednoduchú dátovú štruktúru, ktorá umožňuje jej rýchlu modifikáciu. Nevýhodou je veľký počet ukazateľov, ktoré zbytočne zaberajú užitočný priestor. Časová zložitosť algoritmu je lineárna s dĺžkou používaných prefixov. Pre IPv4 je to v najhoršom prípade 32 prístupov do pamäte. Trie sa vo svojej základnej podobe v praxi takmer nepoužíva. Existuje však mnoho algoritmov, ktoré pracujú na tomto princípe a ďalej ho modifikujú a optimalizujú (CPE, Lulea a iné) [2].



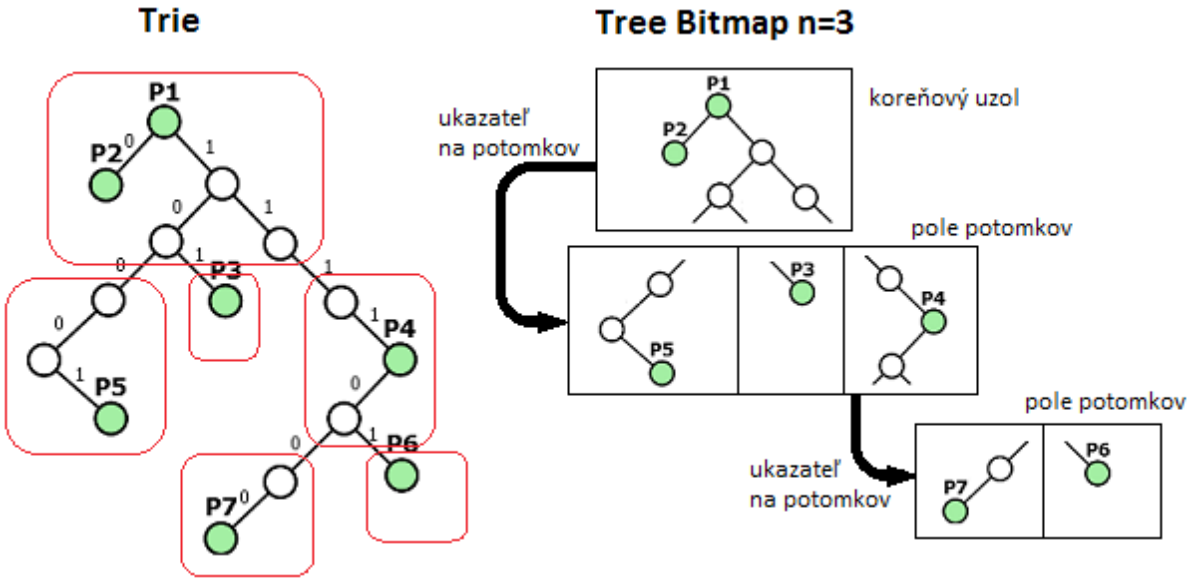
Obrázok 3.1: Príklad jednoduchého stromu Trie

3.2 Tree Bitmap

Tree Bitmap [3] je ďalší príklad LPM algoritmu založeného na stromovej štruktúre. Oproti klasickej trie je zložitejší, avšak je výrazne efektívnejší – má nižšie pamäťové nároky, menej potrebných ukazateľov a vyššiu rýchlosť vyhľadávania.

Na rozdiel od popísaného algoritmu trie, ktorý bol „unibit“, je Tree Bitmap „multibit“ metóda. To znamená, že v každom kroku výpočtu spracujeme viacero bitov vstupnej hodnoty naraz. Jeden uzol tohto stromu teda reprezentuje niekoľko uzlov štruktúry trie. Túto hodnotu môžeme parametrizovať a na základe zvolenej hodnoty má uzol rôznu veľkosť a počet následníkov. Platí však, že pre nastavenú hodnotu *n* (počet spracovaných bitov v jednom kroku) má uzol maximálne 2^n potomkov a jeden uzol reprezentuje *n* úrovní jednoduchej trie.

Jednou z hlavných myšlienok algoritmu Tree Bitmap je, že všetky synovské uzly jedného otcovského uzla sa v pamäti nachádzajú bezprostredne za sebou. Vďaka tomu nám vystačí pre každý uzol maximálne jeden ukazateľ, ktorý ukazuje na prvého potomka. Ukazatele na ďalších jeho potomkov vieme jednoducho dopočítať ako offset od prvého ukazateľa. Táto vlastnosť nám výrazne zníži počet potrebných ukazateľov a tým pádom ušetrí množstvo pamäte. Na obrázku 3.2 vidíme príklad jednoduchého stromu Tree Bitmap s parametrom $n = 3$.

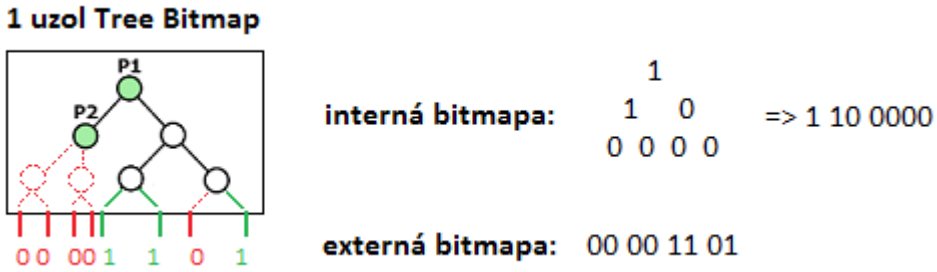


Obrázok 3.2: Príklad stromu Tree Bitmap s parametrom n=3

Už z názvu metódy vyplýva, že tento algorimus bude určitým spôsobom využívať bitmapy. Konkrétne potrebujeme 2 bitmapy pre každý uzol:

- interná bitmapa - pre popis platných prefixov v uzle
- externá bitmapa - pre ukazatele na synovské uzly

Bimapa obsahuje hodnotu 1, ak na danej pozícii podstrom/prefix existuje, inak obsahuje 0. Popis tvorby internej a externej bitmapy je bližšie vysvetlený na obrázku 3.3. Pri zadanom parametri n je veľkosť internej bitmapy (2^n - 1) a veľkosť externej bitmapy 2^n. Platné prefixy uzlu sú podobne ako následníci uložené za sebou v pamäti. Opäť nám teda postačí jeden ukazateľ na prvý platný prefix a ostatné ukazatele je jednoduché dopočítať s pomocou internej bitmapy.



Obrázok 3.3: Ukážka tvorby internej a externej bitmapy

Vyhľadávací algoritmus je pomerne jednoduchý. Začíname samozrejme klasicky v koreni stromu. Podľa zadaného parametru zoberieme prvých *n* bitov adresy. Dostaneme tak n-bitové číslo, ktoré použijeme ako index *P* do externej bitmapy. Ak sa na tejto pozícii nachádza 1, znamená to, že v požadovanom smere sa nachádza ďalší podstrom, v ktorom bude vyhľadávanie pokračovať. Inak sa tu potomok nenachádza a výpočet bude ukončený. Aby sme boli schopní posunúť sa na ďalší uzol (ak existuje) musíme zistiť offset, ktorý jednoducho spočítame ako počet jednotiek v externej bitmape pred pozíciou *P*. Pred presunom na ďalší podstrom ešte musíme skontrolovať platné prefixy na zadanej ceste pomocou internej bitmapy. Týmto spôsobom pokračujeme až spracujeme všetky bity vstupnej adresy, alebo sa dostaneme na koniec stromu.

Tento algoritmus je v súčasnosti často využívaný hlavne pre svoje nízke pamäťové nároky (pri vhodne zvolenom parametri *n*) a vysokú rýchlosť. Navyše je pomerne jednoducho implementovateľný v hardwari. Časová zložitosť je lineárna s dĺžkou vstupnej hodnoty. Táto hodnota je konštantná (32 bitov pre IPv4), a preto aj časová zložitosť algoritmu Tree Bitmap je konštantná. Pre hodnotu $n = 16$ by teoreticky bol výpočet hotový v 2 krokoch. Takto vysoké hodnoty parametru sa však nevyužívajú, pretože pamäťové nároky by sa stali neúnosné. Obvykle sa používajú hodnoty $n = 3$ alebo 4.

3.3 Shape-shifting tree

Táto LPM metóda pracuje na podobnom princípe ako algoritmus Tree Bitmap. Snaží sa však naďalej vylepšovať jeho vlastnosti a efektivitu. Kódovanie jednotlivých uzlov Tree Bitmap by bolo ideálne pre plne zaplnený vyvážený strom. Táto situácia sa však v praxi nevyskytuje a často sa objavujú stromy, ktoré obsahujú dlhé nevetvené cesty a iné nesúmerné štruktúry. Potom sa Tree Bitmap stáva nevýhodný, pretože uzly zostávajú nezaplnené a dochádza k plytvaniu pamäte. Tento problém odstraňuje nové kódovanie, ktoré využíva Shape-shifting tree (SST) [11]. Navyše sa táto metóda ako jedna z mála ukazuje ako vhodná na využitie pri IPv6 adresách. Nevýhodou však je, že SST je pomerne komplexný a náročný na implementáciu. Tiež modifikácia vytvoreného stromu a pridávanie nových prefixov nie je vôbec triviálna záležitosť, pretože sa často mení veľká časť stromu a tvar mnohých uzlov.

3.3.1 Reprezentácia SST

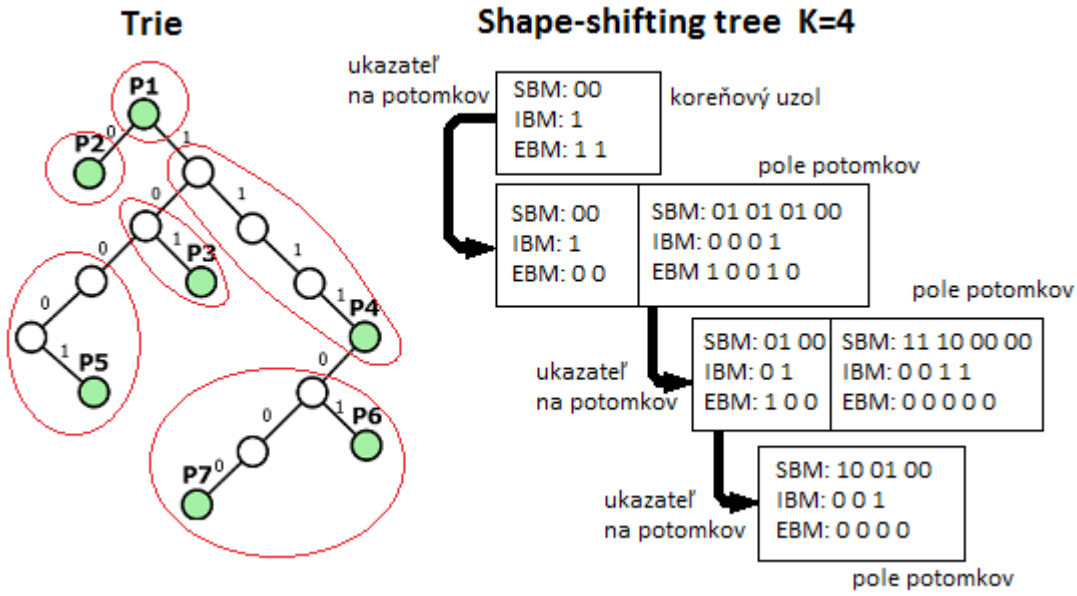
Algoritmus SST má podobne ako Tree Bitmap jeden parameter *K*. Ten však neznamená počet úrovní trie, ktorý zahŕňa jeden uzol (ako v Tree Bitmap), ale udáva maximálny počet uzlov jednoduchej trie, ktoré sa môžu nachádzať v jednom SST uzle. Každý uzol má potom 3 bitmapy:

- tvarová bitmapa (SBM)
- interná bitmapa (IBM)
- externá bitmapa (EBM)

Interná a externá bitmapa majú rovnakú funkciu ako u Tree Bitmap a v princípe ich vytvárame rovnakým spôsobom. SST však má uzly rôznych tvarov, a preto musíme myslieť na to, že bity jednotlivých bitmáp sú vždy v breadth-first poradí. Tretia bitmapa (SBM) slúži práve pre popis tvaru uzlu. SST uzol, ktorý obsahuje *K* uzlov má tvarovú bitmapu veľkosti $2K$ bitov. Pre vysvetlenie výpočtu tvarovej bitmapy najprv SST uzol rozšírime o tzv. fiktívne uzly. Každý originálny uzol trie,

ktorý nemá v rámci SST uzla žiadneho potomka, dostane 2 fiktívne uzly. V prípade, že má jedného potomka, dostane 1 fiktívny uzol. Potom prejdeme originálne uzly v breadth-first poradí a postupne zapisujeme do SBM hodnoty 0 pre fiktívnych potomkov alebo 1 pre originálnych potomkov.

Na obrázku 3.4 vidíme príklad štruktúry trie, ktorá bola rozdelená na SST uzly, spolu s ukážkou bitmáp. Každý uzol opäť obsahuje len 2 ukazatele (na prvého potomka/prefix) s možnosťou dopočítania ostatných ukazateľov ako offset.



Obrázok 3.4: Príklad stromu Shape-shifting tree spolu s ukážkou bitmáp

3.3.2 Vytvorenie SST

Štruktúru trie by sme mohli rozdeliť na SST uzly mnohými spôsobmi. Dve hlavné kritériá, ktoré by sme chceli dodržať sú:

- minimálna výška stromu
- minimálny počet uzlov stromu

Tieto dve podmienky však stoja proti sebe a nie je možné zabezpečiť splnenie oboch z nich. Pre účely vyhľadávania je však podstatný čo najmenší počet prístupov do pamäte. Pri vytváraní optimálneho stromu pre nás preto bude prioritné druhé kritérium.

Vytvorenie SST stromu je pomerne náročná úloha, ktorá si vyžaduje minimálne 2 priechody štruktúrou trie. V prvom priechode (ideálne post-order) týmto stromom spočítame pre každý uzol x hodnotu $s(x)$, ktorá udáva počet uzlov v podstrome s koreňom x. V druhom priechode stromom „rozsekáme“ trie na jednotlivé SST uzly a to nasledovným spôsobom:

1. Prechádzam nespracované uzly unibit trie v breadth-first poradí
2. Ak narazím na uzol y, pre ktorý platí $s(y) \leq K$, orežem tento uzol a všetkých jeho potomkov a z vzniknutého podstromu vytvorím nový uzol SST
3. U všetkých predkoch uzlu y upravím hodnotu $s(x) = s(x) - s(y)$

Tieto kroky sa opakujú dokým nie sú spracované všetky uzly unibit trie. Výsledkom je Shape-shifting trie, ktorý má minimálnu možnú výšku stromu. Je to výpočtovo vysoko náročná úloha, preto je ideálne, aby sa prefixová množina nemenila a teda bol tento výpočet vykonaný len raz. Vyhľadávanie je ale po vytvorení tohto stromu veľmi rýchle.

3.3.3 Vyhľadávanie prefixov v SST

Posledná otázka, ktorá zostáva je samotný spôsob vyhľadávania prefixov vo vytvorenej štruktúre. Ten je v princípe opäť podobný vyhľadávaniu v Tree Bitmap. Ako pri každom stromovom algoritme aj v prípade SST sa vyhľadávanie začína klasicky v koreňovom uzle. Základný krok pre pohyb v rámci stromu je správne dekódovať tvarovú bitmapu a podľa toho následne určiť smer ďalšieho postupu. Táto úloha nie je úplne triviálna a potrebujeme k tomu definovať niekoľko premenných, ktoré sú súčasťou výpočtu:

- n_i – počet uzlov (vrátane fiktívnych) v SST uzle vo vzdialenosti i od koreňa uzla
- f_i – pozícia bitu v tvarovej bitmape, ktorá označuje prvý uzol vo vzdialenosti i od koreňa, platí, že $f_1 = 0$ a $f_i = f_{i-1} + n_{i-1}$
- ďalej budeme potrebovať 2 funkcie – ones(i, j) a zeros(i, j), ktoré spočítajú počet jednotiek/núl v tvarovej bitmape medzi indexmi i až j
- $n_i = 2 \times ones(f_{i-1}, f_i - 1)$
- a_i – i-tý bit IP adresy
- p_i – index do tvarovej bitmapy, ktorý ukazuje na požadovaný uzol, ktorý sa vzťahuje k i-tému bitu IP adresy, platí, že $p_1 = a_1$ a pre $i > 1$ platí:

$$p_i = f_i + 2 \times ones(f_{i-1}, p_{i-1} - 1) + a_i$$

Najdôležitejší je pre nás práve výpočet premennej p_i, ktorá nám udáva, ktorým smerom v strome sa budeme uberať a ako bude pokračovať výpočet. Ostatné premenné sú viac-menej pomocné a sú potrebné k výpočtu p_i.

Dokým je hodnota tvarovej bitmapy na idexe p_i rovná 1, zostáva výpočet v aktuálnom SST uzle. Akonáhle narazíme na hodnotu 0, znamená to, že opúšťame aktuálny uzol a musíme zistiť, či existuje synovský uzol, v ktorom budeme pokračovať. K tomu potrebujeme externú bitmapu. Ak sa na indexe x = zeros(0, p_i - 1) nachádza hodnota 1, výpočet pokračuje v ďalšom uzle, inak vyhľadávanie končí. Ofset na nasledujúci uzol vypočítame, tak, že spočítame počet jednotiek v externej bitmape pred indexom x.

Zároveň samozrejme musíme v každom kroku kontrolovať internú bitmapu. V prípade, že sa v bitmape na indexe x = ones(0, p_i) nachádza hodnota 1, tak sme našli platný prefix na našej ceste. Ukazateľ na tento prefix vypočítame zase pomocou ofsetu obdobným spôsobom ako v predchádzajúcom prípade. Výpočet sa ukončí po spracovaní všetkých bitov IP adresy alebo po dosiahnutí konca stromu. Výsledok celého vyhľadávania je potom posledný nájdený prefix na zvolenej ceste.

Tento algoritmus za cenu svojej vysokej komplexnosti a náročnosti ponúka veľmi dobré výsledky, nízke pamäťové nároky a vysokú rýchlosť.

3.4 Multi-match

Na rozdiel od predchádzajúcich troch predstavených algoritmov táto metóda nie je založená na stromovej štruktúre. Je to algoritmus, ktorý sa snaží využiť výhody rýchlosti a jednoduchosti hashovacích tabuliek. Hlavnou myšlienkou metódy multi-match je rozdelenie vstupnej množiny prefixov na niekoľko podmnožín. Počet týchto skupín je určený ako parameter *p*. Prefixy, ktoré patria do jednej podmnožiny musia mať rovnakú dĺžku. Potom jednoducho každú množinu uložíme do jednej hashovacej tabuľky.

V princípe je táto metóda veľmi jednoduchá. Musíme však vyriešiť niekoľko problémov. Jedným z nich je požiadavka, že prefixy v podmnožine musia mať rovnakú dĺžku. Obecne majú prefixy rôznu dĺžku a počet skupín je spravidla väčší ako zadaný parameter *p*. Táto situácia sa rieši generovaním pomocných prefixov. V prípade, že máme jeden prefix dĺžky *n* bitov a zadaný prefix chceme zaradiť do podmnožiny, ktorá požaduje *m*-bitovú dĺžku prefixu, pričom $n<m$, musíme z originálneho prefixu expandovať nové prefixy nasledovným spôsobom. Každý vygenerovaný prefix bude mať dĺžku *m* bitov, pričom prvých *n* bitov je zhodných z originálnym prefixom. Zvyšok prefixu je doplnený tak, aby vzniknuté prefixy spolu pokryli všetky možnosti.

Príklad:

- prefix: 101 ($n = 3$)
- veľkosť podmnožiny $m = 6$
- expandované prefixy: 101000, 101001, 101010, 101011, 101100, 101101, 101110, 101111

Z uvedeného vyplýva, že počet expanzií závisí na parametroch *n* a *m* a to podľa vzťahu $2^{(m - n)}$. Tu sa nám vynára ďalší problém a to, ako zvoliť hranice jednotlivých podmnožín (parameter *m* pre každú podmnožinu). Pri nevhodnom nastavení totiž dochádza k zbytočnému generovaniu veľkého počtu prefixov a tým pádom k plytvaniu pamäte. Jednou možnosťou je rovnomerné rozdelenie na celom intervale <0, veľkosť_domény>. Tento interval sa teda rozdelí s krokom k = veľkosť_domény/p. Pre zadaný parameter $p = 4$ a veľkosť_domény = 32 bitov potom vzniknú podmnožiny s veľkosťou prefixov 8, 16, 24 a 32 bitov. Tento spôsob je rýchly a vo väčšine prípadov poskytuje dostatočné výsledky. No pre optimálne využitie pamäte môžeme použiť inú variantu. Úlohou je minimalizovať počet expanzií, teda minimalizovať vzdialenosť hodnôt *m* a *n*. Potrebujeme k tomu histogram, ktorý zobrazuje rozloženie dĺžok prefixov v prefixovej množine. Potom preveríme všetky možné kombinácie nastavenia hraníc podmnožín a pre každé nastavenie vypočítame potrebný počet expanzií. Po dokončení získame hľadané minimum a toto nastavenie uplatníme. Tento spôsob nám ušetrí pamäťové nároky, pretože nájde optimálne riešenie, je však veľmi náročný na výpočtový výkon. Treťou alternatívou je ručné, empirické nastavenie.

Po vytvorení podmnožín a vygenerovaní potrebných prefixov každú množinu uložíme do samostatnej hashovacej tabuľky. Veľkosť tabuľky zistíme ako dvojkový logaritmus počtu prefixov zaokrúhlený smerom nahor. Kľúčom do tabuľky je binárna reprezentácia prefixu a hodnotou môže byť referencia na konkrétny prefix. V tejto chvíli je štruktúra dokončená a pripravená na použitie. Vyhľadávanie je skutočne jednoduché. Vo všetkých vytvorených tabuľkách súčasne vyhľadáme zadanú IP adresu. Kľúčom do každej tabuľky je prvých *m* bitov adresy. Výsledkom je najdlhší nájdený prefix zo všetkých tabuliek.

Túto metódu uvádzam ako vlastnú pokusnú metódu, ktorá zatiaľ nebola bližšie zdokumentovaná a otestovaná. Myslím si, že v hardwarovej implementácii by mohla zaujať hlavne svojou rýchlosťou, predovšetkým pre možnosť paralelného vyhľadávania. Taktiež využitie hashovacích tabuliek, ktoré sú známe svojou rýchlosťou prístupu by mohli prispieť k výkonu tejto metódy. Hlavné úskalie algoritmu vidím vo vyšších pamäťových nárokoch, ktoré sú závislé na rozložení a veľkosti vstupnej množiny a zvolenom parametri p.

4 Návrh a implementácia

V predchádzajúcich častiach tejto práce bola uvedená potrebná teória a boli predstavené 4 vybrané LPM metódy. Táto kapitola obsahuje základné informácie o návrhu výslednej knižnice algoritmov. Ďalej sa stručne venujem popisu implementácie, ktorá vychádza z vytvoreného objektového modelu. Všetky predstavené algoritmy boli implementované vo vyššom programovacom jazyku Python. Pri implementácii som sa pridržiaval vyššie uvedených informácií pre splnenie požadovaných vlastností. Aby bolo možné s metódami pracovať a úspešne simulovať reálne nasadenie, bolo potrebné myslieť aj na budúcu hardwarovú implementáciu. Preto algoritmy okrem základnej funkčnosti obsahujú niektoré dodatočné funkcie, ako napríklad hlásenie o celkovej potrebnej pamäti.

4.1 Výskumná skupina ANT@FIT

Táto bakalárska práca bola vytvorená v spolupráci s výskumnou skupinou ANT@FIT [5], ktorá vznikla v roku 2009 na Fakultě informačních technologií VUT v Brně. Je to skupina akcelerovaných sieťových technológii (Accelerated Network Technologies - ANT), ktorá sa zaoberá rôznymi spôsobmi akcelerovania sieťových aplikácií a zariadení. Okrem urýchľovania časovo kritických operácií sa skupina zaoberá tiež návrhom nových prototypov zariadení pre sieťovú infraštruktúru, monitorovanie a bezpečnosť počítačových sietí, ktoré budú pracovať s prenosovými rýchlosťami 10 – 100 Gbps. V tejto oblasti skupina spolupracuje tiež so združením CESNET a projektom Liberouter [6]. Hlavným kritériom je cena a nízka spotreba systémov, ktoré sú postavené na báze počítača a akceleračných kariet. Pre urýchľovanie primárne využíva technológie MultiCORE alebo FPGA.

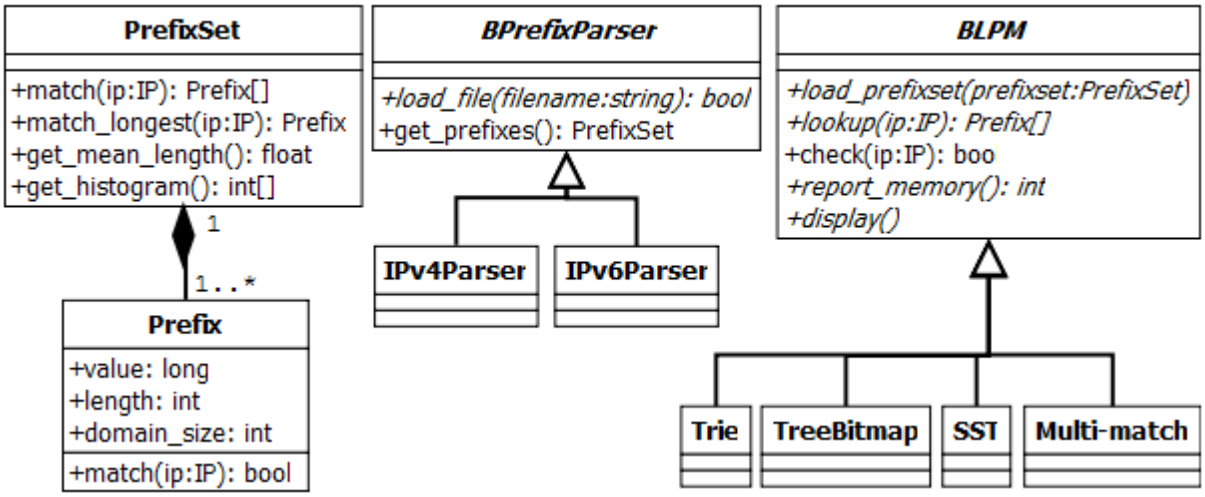
Algoritmy pre vyhľadávanie najdlhšieho zhodného prefixu sú práve jednou z oblastí výskumu. Pri rýchlostiach do 100 Gbps musí byť vyhľadanie prefixu vykonané v časovom úseku desiatok nanosekúnd [4]. Skupina ANT teda potrebuje získať porovnania a simulácie z knižnice rôznych LPM algoritmov. Táto voľne dostupná knižnica experimentov sa potom ďalej môže využívať v rámci výskumu.

4.2 Objektový návrh

Projekt skupiny ANT@FIT je rozsiahly a zaoberá sa napríklad klasifikáciou paketov, správou sieťových tokov, analýzou hlavičky paketu, vyhľadávaním najdlhšieho zhodného prefixu a ďalšími úlohami v tejto oblasti. Na projekte spolupracuje skupina ľudí, ktorí pracujú nezávisle na sebe. Preto musí byť samozrejmosťou dobrá dekompozícia úlohy a využívanie princípov objektového programovania. Pre oblasť LPM bol vytvorený objektový návrh, ktorý vidíme na obrázku 4.1.

Objektový model prehľadne znázorňuje všetky potrebné triedy objektov a závislosti medzi nimi. Triedy, ktorých názov začína písmenom „B“ a je napísaný kurzívou (*BPrefixParse*, *BLPM*), sú takzvané bázové alebo abstraktné triedy. To znamená, že sa z nich nebudú vytvárať žiadne inštancie, ale slúžia ako spoločný základ pre všetky odvodené triedy. Tieto odvodené triedy sú v návrhu zobrazené s prázdnou šípkou, ktorá smeruje k rodičovskej triede. Každá trieda môže mať určité atribúty a metódy, ktoré určujú jej vlastnosti a chovanie. Atribúty sa nachádzajú v prvom poli

a obsahujú dátový typ, metódy sú potom v druhom poli objektovej triedy a majú vstupné parametre a dátový typ návratovej hodnoty. Ďalej si môžeme všimnúť, že názvy niektorých metód v bázovej triede sú vypísané kurzívou. To značí, že tieto metódy budú povinne implementované v každej triede, ktorá dedí z danej bázovej triedy. Ostatné metódy sú implementované priamo v konkrétnej triede. Nakoniec sa v návrhu nachádza agregácia, ktorá sa znázorňuje plnou šípkou. V našom prípade znamená, že PrefixSet sa skladá z jedného až N prefixov.



Obrázok 4.1: Objektový návrh pre oblasť LPM

4.3 Popis implementácie

Implementácia knižnice experimentov prebieha v programovacom jazyku Python verzie 2.6.x. Pri implementácii som vychádzal z uvedeného objektového návrhu. V nasledujúcich podkapitolách uvediem bližšie informácie k jednotlivým triedam návrhu.

4.3.1 Prefix a PrefixSet

Základnou jednotkou, s ktorou pracujú všetky metódy sú triedy Prefix a PrefixSet. Trieda Prefix predstavuje klasickú reprezentáciu prefixu IP adresy obecne ľubovoľnej dĺžky. Hodnota prefixu je zadaná ako celé nezáporné dekadické číslo. Počet platných bitov, s ktorými budeme pracovať potom udáva atribút `length`, prípadne môžeme zadať masku prefixu (napr. 11111100). V práci s IPv4 a IPv6 adresami teda nie je v princípe žiadny rozdiel. Jediný atribút ktorý ich odlišuje je `domain_size`, ktorý určuje maximálnu veľkosť domény prefixu. Pre verziu 4 nadobúda hodnotu 32, pre verziu 6 potom 128. Obecne však môžeme pracovať s prefixami ľubovoľnej dĺžky. Pre nás najdôležitejšia metóda triedy Prefix je metóda `match`. Tá porovná prefix so zadanou hodnotou a v prípade zhody všetkých platných bitov vráti hodnotu `true`.

Všetky prefixy, s ktorými budeme pracovať spolu vytvárajú množinu prefixov, ktorú predstavuje trieda PrefixSet. Táto množina je interne reprezentovaná ako zoznam prefixov usporiadaný zostupne podľa dĺžky prefixu, pričom sú odstránené duplicitné prefixy. Trieda PrefixSet obsahuje metódu `match`, ktorá je implementovaná pomocou triviálneho sekvenčného algoritmu

vyhľadávania najdlhšieho zhodného prefixu a vracia množinu všetkých prefixov, ktoré vyhovujú zadanej vstupnej hodnote. Táto množina je opäť usporiadaná zostupne podľa dĺžky. Implementovaná je tiež metóda `match_longest`, ktorá vráti len najdlhší zhodný prefix alebo hodnotu `None` v prípade, že žiadny prefix nevyhovuje. Ďalej využijeme metódu `get_mean_length`, ktorá spočíta priemernú dĺžku prefixu a metódu `get_histogram`, ktorá vráti histogram rozloženia prefixov podľa dĺžky. Histogram je reprezentovaný ako pole veľkosti domain_size + 1. Index predstavuje dĺžku prefixu a hodnota na tomto indexe je počet prefixov danej dĺžky v množine PrefixSet.

4.3.2 BPrefixParser a odvodené triedy

Pre načítanie prefixovej množiny sa využívajú tzv. parsery, ktoré dokážu spracovať vstupný súbor v zadanom formáte. Základnou úlohou je načítať všetky prefixy zo vstupného súboru a zároveň skontrolovať ich správnosť. Výsledkom je vytvorená množina PrefixSet, ktorá obsahuje všetky platné načítané prefixy. Ako vidíme v objektovom návrhu, bázová trieda BPrefixParser obsahuje len metódu `get_prefixes` na získanie vytvoreného prefixsetu. Celá funkčnosť je implementovaná v jednej z odvodených tried. Ja som vytvoril jednoduchý textový parser pre IP adresy verzie 4. Po zavolaní metódy `load_file`, ktorá dostane ako parameter cestu k vstupnému súboru, prebehne spracovanie a načítanie prefixov. Tento parser očakáva na každom riadku práve jeden prefix zapísaný vo forme masky (napr. 192.168.1.0/24) alebo pomocou hviezdičkovej notácie (napr. 192.168.1.*). V prípade, že sa na riadku nachádzajú ďalšie údaje, sú tieto ignorované a pokračuje sa na ďalšom riadku. Každý prefix je skontrolovaný a potom je pridaný do množiny prefixov. Ak sa na riadku nenachádza platný prefix, je ignorovaný a pokračuje sa v spracovaní. Po dosiahnutí konca súboru máme k dispozícii vytvorený PrefixSet so všetkými validnými načítanými prefixami.

4.3.3 BLPM a odvodené triedy

Najdôležitejšou časťou sú samozrejme jednotlivé LPM metódy. Každá z nich je odvodená od bázovej triedy BLPM. Tá určuje niektoré povinné metódy, ktoré musí každá odvodená trieda implementovať. Sú to metódy:

- load_prefixset
- lookup
- report_memory
- display

Metóda `load_prefixset` načíta prefixy z množiny danej parametrom a vytvorí všetky dátové konštrukcie, ktoré sú potrebné pre vyhľadávanie. U stromových algoritmoch to spravidla znamená vytvorenie správnej stromovej štruktúry. Po tomto kroku je metóda pripravená na použitie a je možné vyhľadávať prefixy. Vyhľadávanie zaistí metóda `lookup`. Výsledkom vyhľadávania je zostupne zoradený zoznam všetkých prefixov, ktoré vyhovujú vstupnej hodnote. Pre potreby experimentov a simulácií musíme poznať pamäťové nároky každej LPM metódy, ktoré nám poskytuje metóda `report_memory`. Tá spočíta a vráti počet potrebných prvkov aktuálne vytvorenej dátovej štruktúry. U stromov to býva hlavne počet uzlov a ukazateľov potrebných pre aktuálne načítanú množinu prefixov. Z týchto údajov potom vieme jednoducho dopočítať skutočné pamäťové požiadavky pre hardwarovú implementáciu. Nakoniec každý algoritmus obsahuje metódu `display`.

Táto metóda zobrazí vytvorený vyhľadávací strom v jednoduchej textovej podobe. Ku kontrole správnosti vyhľadávacieho algoritmu slúži metóda `check`, ktorá je implementovaná už v bázovej triede BLPM. Po jej zavolaní sa skontroluje výsledok metódy `lookup` s výsledkom metódy `match` (implementovanej v PrefixSet). V prípade, že sa výsledky zhodujú, považujeme vyhľadávanie za správne.

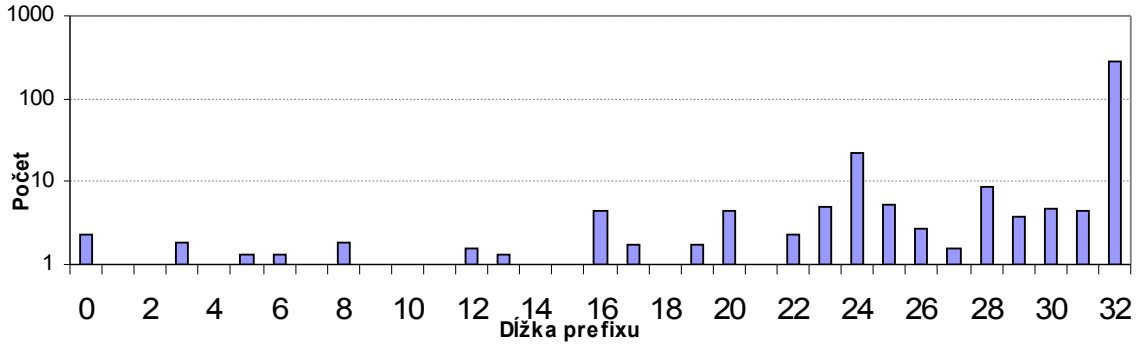
Jednotlivé algoritmy boli implementované podľa rozboru uvedeného v predchádzajúcej kapitole. Strom jednoduchej štruktúry trie predstavuje objekt `Tree`, ktorý sa skladá z objektov `Node`. Každý uzol `Node` obsahuje 2 ukazatele na ďalšie – synovské uzly `Node`, hodnotu prefixu a ukazateľ na platný prefix. Objekt `Tree` má jednu metódu `add_prefix`, ktorá zadaný prefix správne zakóduje do konštrukcie stromu a vytvorí všetky potrebné uzly na danej ceste. Metóda `load_prefixset` teda jednoducho pre každý prefix z množiny zavolá metódu `add_prefix`. Algoritmus Tree Bitmap je z implementačného hľadiska založený na rovnakom princípe ako Trie, len je o niečo zložitejší. Zmena nastala však u algoritmu Shape-shifting tree. Ako vieme z uvedenej teórie, u tohto algoritmu nedokážeme tak jednoducho zistiť tvar výsledného stromu. Preto najprv musíme vytvoriť klasickú štruktúru trie. Až z nej na základe výpočtov môžeme vytvoriť výsledný SST strom. Odlišný je samozrejme algoritmus Multi-match, ktorý využíva hashovacie tabuľky. Tu môžeme s výhodou využiť slovníky, ktoré sú založené na hashovacej tabuľke a sú vstavanou súčasťou jazyka Python.

5 Testovanie a simulácie

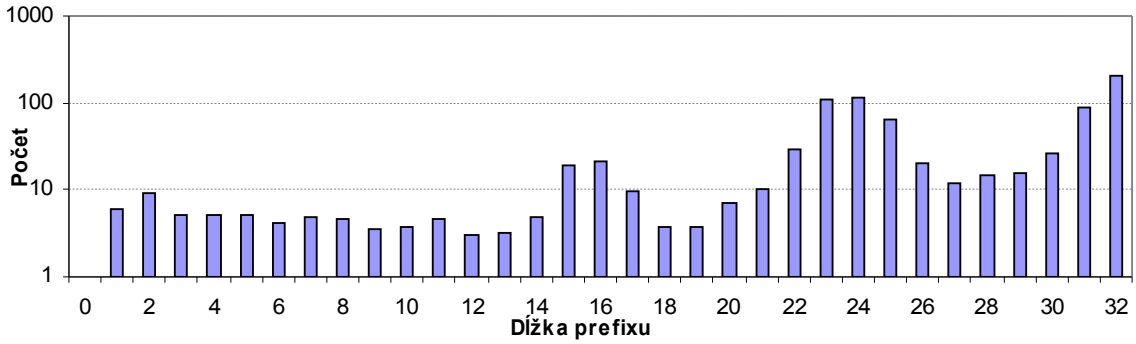
V tejto kapitole sa zameriam na analýzu jednotlivých LPM metód z hľadiska ich pamäťových nárokov a rýchlosti vyhľadávania. Najprv sú uvedené experimenty pre každú metódu zvlášť a na záver kapitoly som sa pokúsil všetky výsledky zhrnúť v prehľadnom porovnaní. Testovanie prebieha softwarovo nad vyššie popísanými algoritmami implementovanými v rámci knižnice. Snažil som sa objektívne zhodnotiť možnosti každého algoritmu a simulovať prípadné využitie v reálnom nasadení. Pri tejto analýze som využil 7 reálnych množín, ktoré sa skutočne využívajú vo firewalloch univerzitných sietí a 8 syntetických množín generovaných pomocou nástroja ClassBench. Pre moje účely som z uvedených testovacích množín použil len zdrojové a cieľové IP adresy. Vlastnosti testovacích množín môžeme vidieť v tabuľke 5.1. V mnohých prípadoch vidíme, že počet unikátnych prefixov je mnohonásobne nižší ako je počet použitých pravidiel. To je spôsobené tým, že v pravidlách sa často opakujú rovnaké IP adresy (veľmi často hlavne univerzálny prefix „any“). Priemerná dĺžka prefixu je spočítaná len medzi univerzálnymi adresami a je podľa očakávania pomerne vysoká. Histogramy rozloženia dĺžky prefixov sú zobrazené na obrázkoch 5.1 a 5.2. Keďže sa jedná o softwarové experimenty, meranie reálneho času výpočtu by pre nás nebolo smerodajné. Pre určenie časovej náročnosti algoritmu preto využívam počet potrebných prístupov do pamäte v najhoršom prípade. Pri stromových algoritmoch je to dané výškou stromu. Pamäťové nároky sú vyjadrené ako veľkosť všetkých dátových štruktúr, ktoré sú nevyhnutné pre výpočet algoritmu. V prípade ukazateľov uvažujem 2 možnosti – ukazateľ pevnej veľkosti (32 bitov) a najmenší možný ukazateľ. Ak nie je uvedené inak, všetky údaje o potrebnej pamäti sú v Bytoch.

Testovacia množina	Počet pravidiel	Počet unikátnych prefixov			Priemerná dĺžka prefixu
		Src IP	Dst IP	Spolu	
real1	70	34	19	45	25,98
real2	173	138	179	179	30,38
real3	104	64	178	178	30,37
real4	340	47	65	112	31,44
real5	1265	157	79	235	31,27
real6	1605	204	144	262	31,23
real7	58	24	39	39	26,21
synth1	882	157	170	319	28,16
synth2	852	121	142	253	28,28
synth3	956	734	273	1004	25,08
synth4	100	74	36	105	19,35
synth5	984	801	441	1209	26,8
synth6	472	102	51	147	18,5
synth7	482	53	81	130	17,72
synth8	481	104	83	177	18,76

Tabuľka 5.1: Vlastnosti použitých testovacích množín



Obrázok 5.1: Histogram rozloženia dĺžky prefixu - reálne množiny



Obrázok 5.2: Histogram rozloženia dĺžky prefixu - syntetické množiny

5.1 Trie

Z uvedeného popisu vieme, že metóda Trie je veľmi jednoduchá a hodnoty prefixov sú zakódované v konštrukcii stromu. V každom uzle ešte potrebujeme informáciu o tom, či obsahuje platný prefix. Ak áno, použijeme navyše ukazateľ na konkrétny prefix do PrefixSetu. Inak je tento ukazateľ nevyužitý. Pri vytvorení každého nového uzla stromu sa potom alokuje miesto pre 2 ukazatele na synovské uzly a 1 ukazateľ na prefix. V prípade využitia 32-bitových ukazateľov potom pamäťové nároky spočítame ako súčet všetkých vytvorených uzlov vynásobené konštantou 3*32 = 96. V prípade optimalizácie pre najnižšie možné nároky predpokladám, že ukazatele na prefixy a ukazatele na uzly budú v oddelených adresných priestoroch. Potom veľkosť potrebných ukazateľov môže byť rôzna. Najmenšia možná veľkosť ukazateľa sa vypočíta ako dvojkový logaritmus počtu uzlov/prefixov zaokrúhlený smerom nahor. Napríklad pre 1000 uzlov nám teda vystačí 10-bitový ukazateľ. Výsledky testovania sa nachádzajú v tabuľkách 5.2 a 5.3.

Pri každej testovacej množine som si všímal výsledky zvlášť pre prefixy zdrojovej IP adresy (scrIP) a cieľovej adresy (dstIP). Poslednou možnosťou je prípad, že sú obe prefixové množiny uložené v jednej spoločnej množine (both). Toto zjednotenie množín nám ušetrí v priemere 6% pamäte v prípade syntetických množín a vyše 25% pamäte pri reálnych množinách. Je to spôsobené tým, že obe množiny (srcIP a dstIP) majú veľký počet spoločných prefixov a v niektorých prípadoch je dokonca jedna podmnožina druhej. Preto som sa rozhodol v ďalších experimentoch pre prehľadnosť uvádzať len výsledky zo zjednotenej množiny „both“.

Ďalším zaujímavým údajom je optimalizácia ukazateľa. Výsledky sú približne rovnaké vo všetkých prípadoch a môže nám až ušetriť v priemere až 70% pamäte. Avšak aj napriek tomu, že s využitím oboch optimalizácií ušetríme priemerne 80% pamäte, stále má táto metóda vysoké pamäťové nároky. Ďalším negatívom algoritmu je rýchlosť vyhľadávania, ktoré potrebuje vo všetkých prípadoch 32 prístupov do pamäte.

		real1	real2	real3	real4	real5	real6	real7
Počet uzlov	srcIP	315	969	401	190	1147	1297	150
	dstIP	237	1190	1186	554	283	786	193
	both	413	1190	1186	698	1378	1523	193
Pamäť [B] (32-bit ptr)	srcIP	3780	11628	4812	2280	13764	15564	1800
	dstIP	2844	14280	14232	6648	3396	9432	2316
	both	4956	14280	14232	8376	16536	18276	2316
Pamäť [B] (min. ptr)	srcIP	945	3391,5	1203	522,5	4301,25	4863,75	393,75
	dstIP	622,13	4462,5	4447,5	1869,75	884,38	2751	530,75
	both	1239	4462,5	4447,5	2355,75	5167,5	5901,63	530,75
Ušetrená pamäť pri both		25,18%	44,88%	25,27%	6,18%	3,64%	26,88%	43,73%
Ušetrená pamäť pri ptr		75,00%	68,75%	68,75%	71,88%	68,75%	67,71%	77,08%
Celkom ušetrených		84,12%	84,38%	84,38%	84,32%	74,07%	78,70%	88,54%

Tabuľka 5.2: Pamäťové nároky metódy Trie - reálne množiny

		synth1	synth2	synth3	synth4	synth5	synth6	synth7	synth8
Počet uzlov	srcIP	670	718	7433	393	10521	233	114	253
	dstIP	1243	907	5392	117	7221	112	204	200
	both	1653	1408	12446	492	16944	330	307	442
Pamäť [B] (32-bit ptr)	srcIP	8040	8616	89196	4716	126252	2796	1368	3036
	dstIP	14916	10884	64704	1404	86652	1344	2448	2400
	both	19836	16896	149352	5904	203328	3960	3684	5304
Pamäť [B] (min. ptr)	srcIP	2345	2423,25	33448,5	1228,13	49974,75	669,86	285	727,36
	dstIP	4661,25	3174,5	23590	292,5	31591,86	280	586,5	575
	both	6405,38	5280	59118,5	1537,5	86838	1072,5	997,75	1436,5
Ušetrená pamäť pri both		13,59%	13,35%	2,96%	3,53%	4,50%	4,35%	3,46%	2,43%
Ušetrená pamäť pri ptr		67,71%	68,75%	60,42%	73,96%	57,29%	72,92%	72,92%	72,92%
Celkom ušetrených		81,57%	80,99%	72,38%	78,96%	70,05%	79,78%	83,73%	81,3

Tabuľka 5.3: Pamäťové nároky metódy Trie - syntetické množiny

5.2 Tree Bitmap

Dátové štruktúry potrebné pre algoritmus Tree Bitmap sú už o niečo zložitejšie. Vďaka tomu však dosiahneme úsporu pamäte. Ako vyplýva z popisu, pamäťové nároky závisia na parametri *n*. Veľkosť internej bitmapy je $2^n - 1$ bitov a veľkosť externej bitmapy 2^n bitov. Ďalej každý uzol musí mať 2 ukazatele, ktorých veľkosť môžeme opäť uvažovať 32 bitov, alebo minimálnu možnú veľkosť. Týmto spôsobom spočítame veľkosť jedného uzla a potom jednoducho dopočítame pamäťové požiadavky

podľa celkového počtu potrebných uzlov. V testovaní som použil parametre n = 3, 4, 5 a 8. Výsledky sú v tabuľkách 5.4 a 5.5.

Vidíme, že so zvyšovaním parametru *n* sa výrazne znižuje počet potrebných uzlov a tým pádom i rýchlosť metódy. Počet prístupov do pamäti bol vo všetkých prípadoch 10, 8, 6 a 4 (pre parametre 3, 4, 5, 8 v tomto poradí). Parameter $n = 8$ nám dáva veľmi slušnú rýchlosť vyhľadávania, no za cenu obrovského zvýšenia pamäťových nárokov. Parametre $n = 3, 4, 5$ sú pomerne vyrovnané čo sa týka potrebnej pamäte. No s každým zvýšením parametru o 1 sa zvýši veľkosť potrebných bitmáp dvojnásobne. To spôsobuje, ako vidíme v tabuľkách, že množstvo ušetrenej pamäte optimalizovaním veľkosti ukazateľa nám ušetrí len malé percento pamäte, pretože väčšinu veľkosti uzla zaberajú bitmapy.

Ďalším podstatným údajom je efektivita využitia alokovanej pamäte. Uzol Tree Bitmap alokuje priestor pre podstrom Trie, ktorý je vyvážený a má výšku *n*, nezávisle na skutočnom tvare štruktúry Trie. So zvyšujúcim sa parametrom preto dochádza k čoraz väčšiemu plytvaniu, pretože len zlomok alokovanej pamäte v skutočnosti potrebujeme a využijeme. Pri $n=8$ to bola táto efektivita priemere len 2,2% pri reálnych množinách a približne 3% pri syntetických.

Hoci vysoké hodnoty parametra nám poskytujú veľmi dobrú rýchlosť vyhľadávania, z uvedených skutočností vyplýva že parametre $n=8$ a vyššie sú absolútne neprijateľné pre reálne využite.

	n	real1	real2	real3	real4	real5	real6	real7
Počet uzlov	3	133	347	346	197	379	420	61
	4	118	351	350	220	439	486	51
	5	79	239	238	136	266	293	39
	8	69	211	210	145	291	325	24
Pamäť [B] (32-bit ptr)	3	1313,38	3426,63	3416,75	1945,38	3742,63	4147,5	602,38
	4	1401,25	4168,13	4156,25	2612,5	5213,13	5771,25	605,63
	5	1254,13	3794,13	3778,25	2159	4333,75	4651,38	619,125
	8	4959,38	15165,63	15093,75	10421	20915,63	23359,38	1725
Pamäť [B] (min. ptr)	3	482,13	1388	1384	738,75	1516	1732,5	205,88
	4	649	2106	2100	1265	2634	2976,75	274,13
	5	750,5	2360,13	2350,25	1326	2660	2966,63	365,63
	8	4519,5	13899,63	13833,75	9533,75	19206	21490,63	1566
Ušetrená pamäť	3	63,29%	59,49%	59,49%	62,03%	59,49%	58,23%	65,82%
	4	53,68%	49,47%	49,47%	51,58%	49,47%	48,42%	54,74%
	5	40,16%	37,80%	37,80%	38,58%	38,62%	36,22%	40,94%
	8	8,87%	8,35%	8,35%	8,51%	8,17%	8,00%	9,22%
Efektívne využitie pamäte v uzloch	3	44,36%	48,99%	48,97%	50,62%	51,94%	51,80%	45,20%
	4	23,33%	22,60%	22,59%	21,15%	20,93%	20,89%	25,23%
	5	16,86%	16,06%	16,07%	16,56%	16,71%	16,77%	15,96%
	8	2,35%	2,21%	2,21%	1,89%	1,86%	1,84%	3,15%

Tabuľka 5.4: Pamäťové nároky metódy Tree Bitmap - reálne množiny

	n	synth1	synth2	synth3	synth4	synth5	synth6	synth7	synth8
Počet uzlov	3	528	447	4088	164	5538	107	103	141
	4	406	364	3184	120	4403	85	78	115
	5	348	296	2462	94	3379	70	62	93
	8	208	189	1652	62	2294	39	35	54
Pamäť [B] (32-bit ptr)	3	5214	4414,13	40369	1619,5	54687,75	1056,63	1017,13	1392,38
	4	4821,25	4322,5	37810	1425	52285,63	1009,38	926,25	1365,63
	5	5524,5	4699	39084,25	1492,25	53641,63	1111,25	984,25	1476,38
	8	14950	13584,18	118737,5	4456,25	164881,3	2803,13	2515,63	3881,25
Pamäť [B] (min. ptr)	3	2244	1788	18907	615	26997,75	401,25	386,25	546,38
	4	2486,75	2184	21094	675	30270,63	488,75	448,5	661,25
	5	3523,5	2860	26158,75	904,75	36324,25	682,5	596,75	906,75
	8	13728	12450,38	109858	4061	153124,5	2559,36	2296,86	3543,75
Ušetrená pamäť	3	56,96%	59,49%	53,16%	62,03%	50,63%	62,03%	62,03%	60,76%
	4	48,42%	49,47%	44,21%	52,63%	42,11%	51,58%	51,58%	51,58%
	5	36,22%	39,14%	33,07%	39,37%	32,28%	38,58%	39,37%	38,58%
	8	8,17%	8,35%	7,48%	8,87%	7,13%	8,70%	8,70%	8,70%
Efektívne využitie pamäte v uzloch	3	44,72%	45,00%	43,49%	42,86%	43,71%	44,06%	42,58%	44,78%
	4	27,14%	25,79%	26,06%	27,33%	25,66%	25,88%	26,24%	25,62%
	5	15,32%	15,34%	16,31%	16,88%	16,18%	15,21%	15,97%	15,33%
	8	3,12%	2,92%	2,95%	3,11%	2,90%	3,32%	3,44%	3,21%

Tabuľka 5.5: Pamäťové nároky metódy Tree Bitmap - syntetické množiny

5.3 Shape-shifting tree

Dátová štruktúra, ktorá predstavuje jeden uzol stromu Shape-shifting tree je takmer identická s Tree Bitmap. Obsahuje navyše len tvarovú bitmapu. Vďaka tomu však dokáže algoritmus oveľa lepšie hospodáriť s pamäťovým priestorom a prispôsobiť sa konkrétnej prefixovej množine. Opäť platí, že veľkosť jedného uzla je závislá na zadanom parametre *K*. Potrebnú pamäť zase spočítame podobným spôsobom ako pri predchádzajúcom algoritme: každý uzol potrebuje *K* bitov pre internú bitmapu, *K+1* bitov pre externú bitmapu, *2K* bitov pre tvarovú bitmapu a 32 bitov pre každý ukazateľ (resp. menej pri optimalizácii). Pre účely testovania som zvolil hodnoty parametru K = 4, 8, 16 a 32. Prehľad výsledkov zobrazuje tabuľka 5.6 a 5.7.

Z popisu metódy a zo získaných výsledkov vidíme, že táto metóda odstraňuje problémy a nedostatky algoritmu Tree Bitmap. So zvyšovaním parametru *K* sa samozrejme opäť znižuje percento ušetrenej pamäte, no nie tak dramaticky ako v prípade Tree Bitmap. Preto pre prehľadnosť tento údaj v tabuľke neuvádzam. V priemere sa množstvo ušetrenej pamäte pohybovalo okolo 60% pre *K*=4 a 25% pre *K*=32.

Vďaka tvarovej bitmape je alokovaná pamäť využitá takmer na 100%. Ďalšou pozitívnou správou je, že so zvyšovaním parametru *K* nedochádza k tak drastickému zvyšovaniu pamäte, no práve naopak. S každým zvýšením parametru o 1 sa celková veľkosť jedného SST uzla zvýši len o 4 bity. Celková veľkosť uzlov teda bude o niečo vyššia, zato však počet potrebných uzlov sa zníži. To spôsobí aj zníženie celkovej potrebnej pamäte, ako vidíme v tabuľkách.

Výsledky experimentov potvrdili, že metóda SST má veľmi priaznivé vlastnosti, či už sa jedná o rýchlosť alebo pamäťové nároky.

	K	real1	real2	real3	real4	real5	real6	real7
Počet uzlov	4	117	349	348	209	413	453	58
	8	67	202	201	113	225	247	32
	16	39	110	110	62	126	136	16
	32	21	60	59	35	69	75	12
Počet prístupov do pamäte (v najhoršom prípade)	4	12	13	13	15	17	17	10
	8	9	9	9	8	9	10	7
	16	8	7	7	5	6	6	3
	32	4	6	6	5	5	5	3
Pamäť [B] (32-bit ptr)	4	1184,63	3533,63	3523,5	2116,13	4181,63	4586,63	587,25
	8	812,38	2449,25	2437,13	1370,13	2728,13	2994,88	388
	16	628,88	1773,75	1773,75	999,75	2031,75	2193	258
	32	506,63	1447,5	1423,38	844,38	1664,63	1809,38	289
Pamäť [B] (min. ptr)	4	438,75	1483,25	1479	836	1755,25	1981,88	210,25
	8	385,25	1237,25	1231,13	663,88	1378,13	1543,75	176
	16	375,375	1100	1100	604,5	1260	1394	150
	32	367,5	1072,5	1054,63	621,25	1242	1359,38	208,5

Tabuľka 5.6: Pamäťové nároky metódy SST - reálne množiny

	K	synth1	synth2	synth3	synth4	synth5	synth6	synth7	synth8
Počet uzlov	4	531	450	3544	184	4739	172	158	221
	8	303	254	1994	124	2597	137	126	167
	16	170	147	1065	81	1466	102	92	121
	32	94	82	581	37	795	65	52	75
Počet prístupov do pamäte	4	14	13	16	11	15	10	10	12
	8	10	10	11	8	11	7	6	7
	16	7	7	11	6	10	5	5	5
	32	6	6	9	4	9	3	3	4
Pamäť [B] (32-bit ptr)	4	5376,38	4556,25	35883	1863	47982,38	1741,5	1599,75	2237,63
	8	3673,88	3079,75	24177,25	1503,5	31488,63	1661,13	1527,75	2024,88
	16	2741,25	2370,38	17173,13	1306,13	23639,25	1644,75	1483,5	1951,13
	32	2267,75	1978,25	14016,63	892,63	19179,38	1568,13	1254,5	1809,38
Pamäť [B] (min. ptr)	4	2389,5	1912,5	17277	736	24287,38	709,5	651,75	911,63
	8	1931,63	1555,75	13459,5	728,5	18179	839,13	756	1022,88
	16	1742,5	1488,38	11448,75	799,875	15942,75	1020	920	1210
	32	1703,75	1476	10821,13	656,75	14906,25	1170	929	1350

Tabuľka 5.7: Pamäťové nároky metódy SST - syntetické množiny

5.4 Multi-match

Dátové štruktúry potrebné pre tento algoritmu sú veľmi nenáročné a jednoduché. Ide vlastne o niekoľko (počet závisí na zvolenom parametri) tabuliek – jednorozmerné polia. Každá položka poľa predstavuje odkaz na konkrétny prefix. Opäť uvažujem 2 varianty – 32-bitový ukazateľ a najmenší možný ukazateľ pre danú množinu prefixov. Časová zložitosť algoritmu je konštantná. Pri využití perfektného hashovania odstránime problém vzniku možných kolízií. Tým pádom dostaneme v jednom kroku požadované prefixy zo všetkých tabuliek. V druhom výpočtovom kroku potom z nich vyberieme ten najdlhší. Rýchlosťou vyhľadávania teda táto metóda výrazne prekonáva všetky predchádzajúce algoritmy. Výsledky experimentov vidíme v tabuľkách 5.8 a 5.9.

V testoch som použil parameter P = 4, 6 a 8, ktorý udáva počet použitých hashovacích tabuliek. Pri každom som sledoval výsledky klasickej a optimalizovanej metódy. Ako vidíme, výsledky sú

veľmi dobré, porovnateľné s metódou SST. Použitím metódy pre optimálne rozloženie prefixov do skupín ušetríme v priemere približne 40% pamäte. Ďalej vidíme, že výsledky sú vyššie hodnoty parametru P=6 a 8 mnohonásobne lepšie ako je to pri parametri 4. Tento fakt je pochopiteľný, pretože dochádza k menšiemu počtu potrebných expanzií. Vyššie hodnoty parametru už však nie sú vhodné, pretože použitie optimalizovanej metódy by stálo neúmerné množstvo potrebného času a výpočtového výkonu na vytvorenie tabuliek a komplikovala by sa hardwarová implementácia.

	P	real1	real2	real3	real4	real5	real6	real7
Počet prefixov v tabuľkách (vrátane expandovaných)	4	233	938	937	321	601	800	679
	4 opt.	109	314	313	112	327	369	176
	6	136	396	395	177	341	430	215
	6 opt.	65	213	212	112	262	289	72
	8	77	239	238	126	251	292	96
	8 opt.	51	185	184	112	237	266	46
Pamäť [B] (32-bit ptr)	4	1408	6148	6148	2072	4496	4512	3332
	4 opt.	576	2372	2372	540	1664	2450	1220
	6	832	2324	2324	808	1744	2272	1108
	6 opt.	324	1388	1388	548	1288	1288	332
	8	368	1420	1420	612	1196	1340	532
	8 opt.	69	1268	1268	556	1108	1156	216
Pamäť [B] (min. ptr)	4	264	1537	1537	453,25	1124	1269	624,75
	4 opt.	108	593	593	118,13	416	720	228,75
	6	156	581	581	176,75	436	639	207,75
	6 opt.	60,75	347	347	119,88	322	362,25	62,25
	8	276	355	355	133,86	299	376,88	99,75
	8 opt.	51,75	317	317	121,63	277	325,13	40,5
Ušetrená pamäť pri použití optimal	4	59,09%	61,42%	61,42%	73,94%	62,99%	45,70%	63,39%
	6	61,06%	40,28%	40,28%	32,18%	26,15%	43,31%	70,04%
	8	81,25%	10,70%	10,70%	9,15%	7,36%	13,73%	59,40%

Tabuľka 5.8: Pamäťové nároky metódy multimatch - reálne množiny

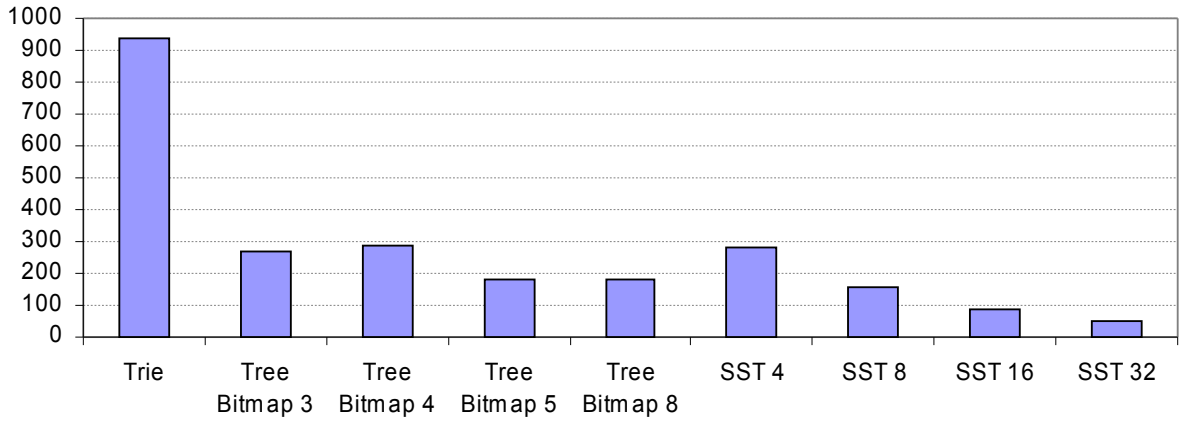
	P	synth1	synth2	synth3	synth4	synth5	synth6	synth7	synth8
Počet prefixov v tabuľkách	4	3049	2642	15742	3052	22790	3296	3156	3730
	4 opt.	2827	2342	3623	2126	4208	3251	3439	3476
	6	1187	940	5041	772	6644	1182	1109	1303
	6 opt.	915	677	1480	518	1716	818	804	971
	8	817	652	2302	408	2810	499	440	588
	8 opt.	565	393	1214	317	1434	454	432	521
Pamäť [B] (32-bit ptr)	4	20480	11520	83968	17408	148992	17408	17408	17408
	4 opt.	15360	14352	18448	12288	22544	17408	20992	15360
	6	6144	4544	26884	4416	43268	6528	6464	6528
	6 opt.	5264	3696	8208	3200	10256	4352	4736	5376
	8	5376	3228	11848	2368	17480	2880	2624	3264
	8 opt.	3216	2032	6864	1952	7952	2688	2624	2688
Pamäť [B] (min. ptr)	4	5760	2880	26240	3808	51216	4352	4352	4352
	4 opt.	4320	3588	5765	2688	7748,5	4352	5248	3840
	6	1728	1136	8401,3	966	14873,4	1632	1616	1632
	6 opt.	1480,5	924	2565	700	3525,5	1088	1184	1344
	8	1512	807	3702,5	518	6008,75	720	656	816
	8 opt.	904,5	508	2145	427	2733,5	672	656	672
Ušetrená pamäť pri použití optimal	4	25,00%	-24,6%	78,03%	29,41%	84,87%	0,00%	-20,6%	11,76%
	6	14,32%	18,66%	69,47%	27,54%	76,30%	33,33%	26,73%	17,65%
	8	40,18%	37,05%	42,07%	17,57%	54,51%	6,67%	0,00%	17,65%

Tabuľka 5.9: Pamäťové nároky metódy multimatch - syntetické množiny

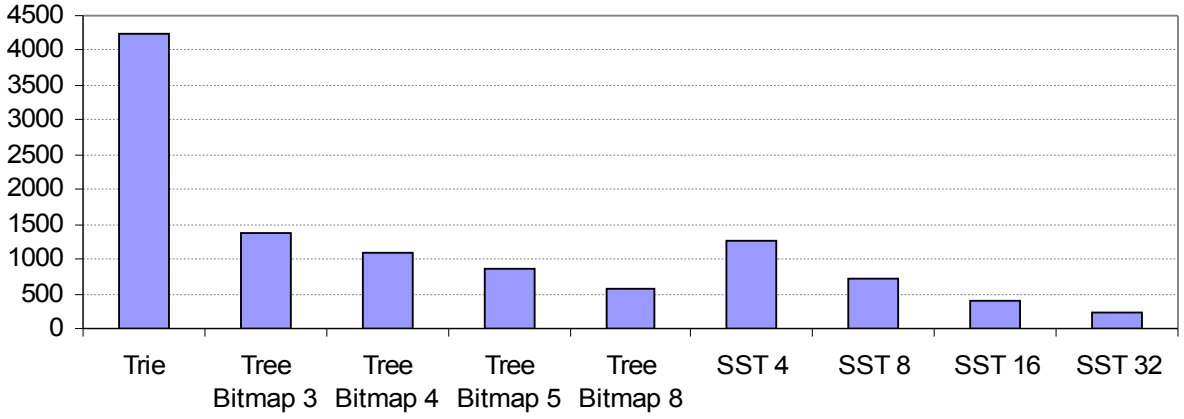
5.5 Porovnanie výsledkov metód

V tejto časti sa pokúsim súhrnne vyhodnotiť výsledky všetkých použitých metód a poukázať na ich kladné a záporné vlastnosti. Na základe výsledkov mojich experimentov je možné vytvoriť si predstavu o nárokoch jednotlivých algoritmoch a o možnostiach reálneho využitia.

Na obrázkoch 5.3 a 5.4 vidíme priemerný počet uzlov, ktoré potrebujú algoritmy založené na stromovej štruktúre. Tento údaj je však pomerne všeobecný, pretože nižší počet uzlov ešte nemusí znamenať nižšie pamäťové nároky. So znižovaním počtu potrebných uzlov by sa mala zvyšovať rýchlosť vyhľadávania, no tá záleží predovšetkým na výške výsledného stromu. Preto nám viac o vlastnostiach algoritmov napovedia nasledujúce grafy. No už na základe uvedených obrázkov je jasne viditeľné zaostávanie Trie oproti ostatným metódam.

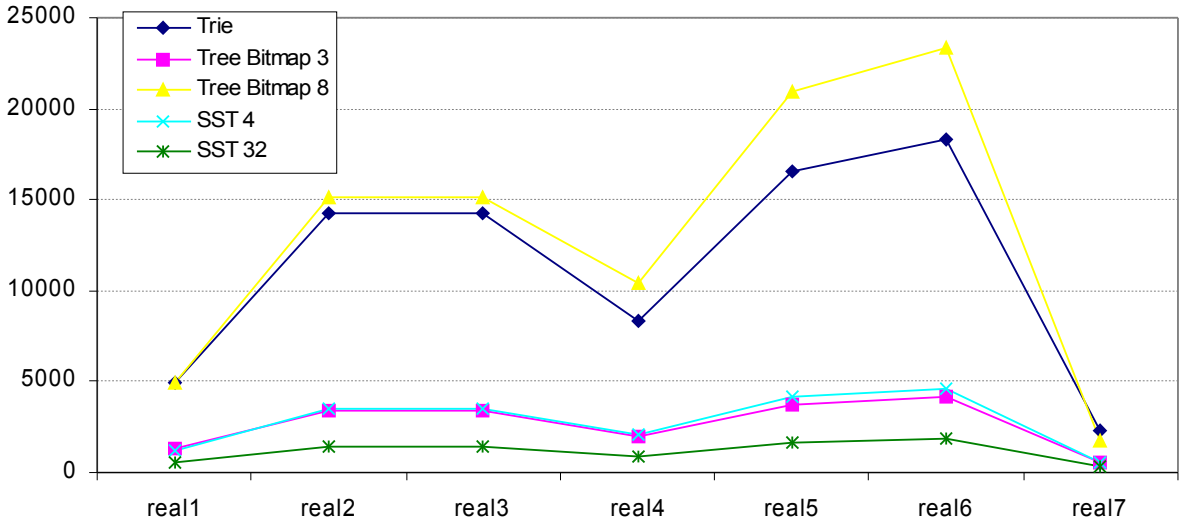


Obrázok 5.3: Priemerný počet potrebných uzlov stromových algoritmov - reálne množiny

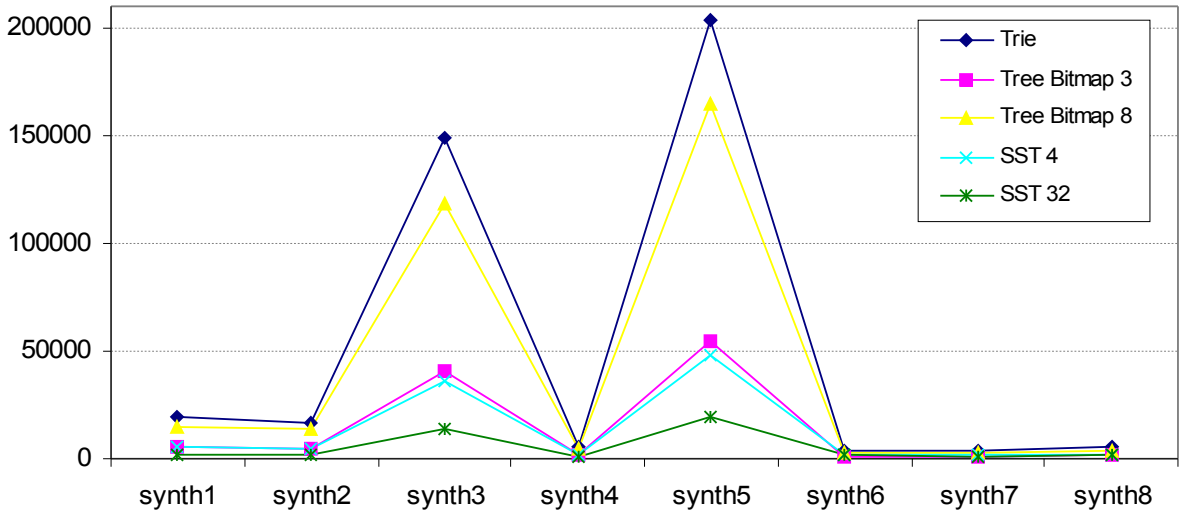


Obrázok 5.4: Priemerný počet potrebných uzlov stromových algoritmov - syntetické množiny

Na nasledujúcich obrázkoch 5.5 a 5.6 som vykreslil pamäťové nároky vybraných stromových algoritmov. Pre prehľadnosť som zvolil len krajné hodnoty použitých parametrov pri metódach Tree Bitmap a Shape-shifting tree. Z uvedeného vyplýva, že jednoznačne najlepšie výsledky dosiahla metóda SST s parametrom K=32. Veľmi pekne je badateľná tiež neefektivita algoritmu Tree Bitmap n=8, ktorý pri reálnych testovacích množinách potreboval dokonca viac pamäte ako štruktúra Trie.



Obrázok 5.5: Pamäťové nároky stromových algoritmov - reálne množiny



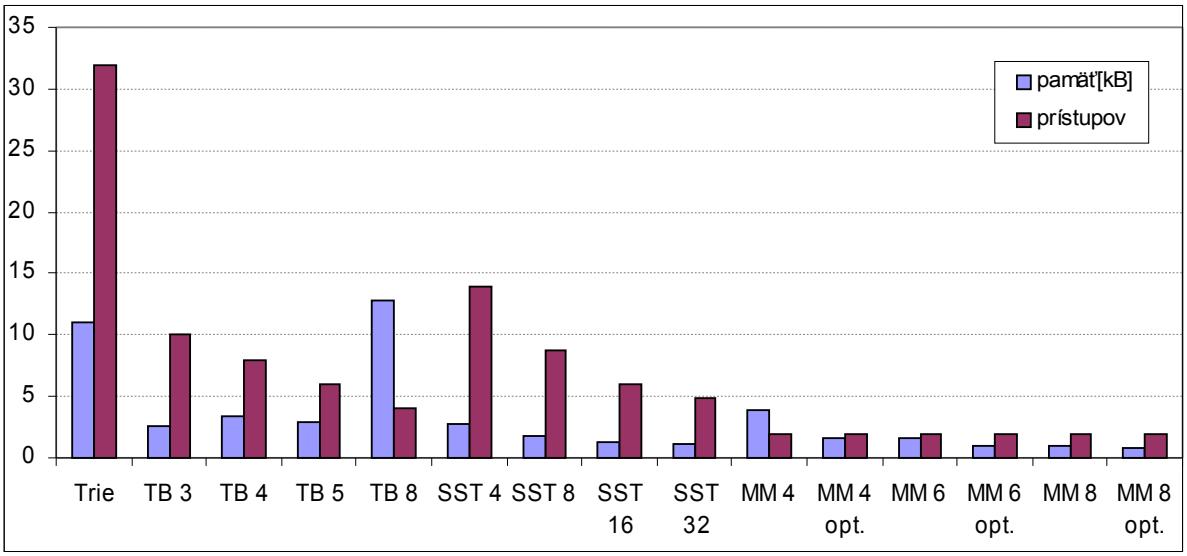
Obrázok 5.6: Pamäťové nároky stromových algoritmov - syntetické množiny

Na posledných dvoch obrázkoch – 5.7 a 5.8, sú uvedené pamäťové nároky a počet potrebných prístupov do pamäti implementovaných LPM metód. Jedná sa o priemerné výsledky nad všetkými testovacími množinami. Tieto dva grafy názorne reflektujú vlastnosti použitých algoritmov. Opäť vidíme nevýhody metód Trie a Tree Bitmap 8, ktoré preto zjavne nie sú vhodné na reálne využitie.

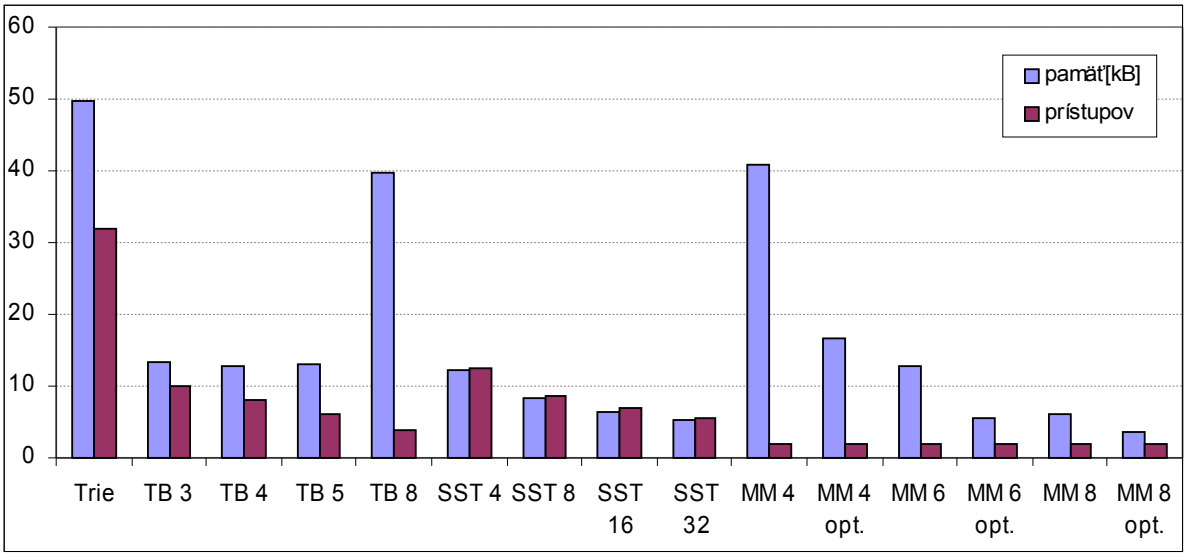
Zaujímavé výsledky môžeme pozorovať v prípade algoritmu Multi-match. Pri reálnych množinách, ktoré majú priemerne 150 prefixov, dosahuje výborné výsledky. No v prípade syntetických množín sa algoritmus zreteľne prepadol. Syntetické množiny obsahujú vyšší počet prefixov, a preto potom dochádza k veľkému počtu potrebných expanzií prefixov. Podobne ako väčšina ostatných algoritmov nie je ani tento optimálny pre verziu IPv6, pretože by dochádzalo k ešte

vyššiemu počtu zbytočných generovaní pomocných prefixov. Pre dobré výsledky by sme teda potrebovali viac hashovacích tabuliek, čo je zase implementačne náročné.

Asi najlepším algoritmom, ktorý poskytuje vyvážené a veľmi dobré výsledky vo všetkých prípadoch je Shape-shifting tree. Vidíme, že efektívne pracuje s pamäťou a rýchlosť algoritmu so zvyšujúcim sa parametrom *K* veľmi dobrá.



Obrázok 5.7: Porovnanie rýchlosti a pamäťových nárokov LPM metód – reálne množiny



Obrázok 5.8: Porovnanie rýchlosti a pamäťových nárokov LPM metód – syntetické množiny

6 Záver

V tejto bakalárskej práci som sa venoval problematike vyhľadávania najdlhšieho zhodného prefixu. Je to pomerne rozsiahla problematika, ktorá je súčasťou klasifikácie paketov v počítačových sieťach. Zameral som sa na 4 LPM alogoritmy, ktoré som detailne teoreticky rozobral a následne prakticky implementoval. Prvé tri algoritmy – trie, tree bitmap a shape-shifting tree sú založenené na stromovej štruktúre a boli predstavené v odbornej literatúre. Posledný algoritmus – multimatch, doteraz nebol bližšie predstavený. Je to vlastný experimentálny algoritmus založený na práci s hashovacími tabuľkami. Tento algoritmus som zaradil s cieľom otestovať jeho možnosti a zhodnotiť jeho efektivitu v porovnaní so známymi a používanými metódami.

Samotnej práci a implementácii predchádzalo zbieranie a dôkladné naštudovanie dostupných materiálov. Zoznámil som sa s architektúrou počítačových sietí založených na TCP/IP a s používanými IP protokolmi verzie 4 a 6. Vyhľadávanie najdlhšieho zhodného prefixu je najdôležitejšou elementárnou úlohou klasifikácie paketov, ktorá sa využíva hlavne v oblasti smerovania a filtrovania dátových paketov. Všetky uvedené prerekvizity som preštudoval, spracoval a uviedol v tejto práci. Následne som sa už venoval výlučne spomenutým LPM algoritmom.

Hlavným cieľom mojej práce bola analýza rýchlosti a pamäťových nárokov LPM metód. Nad implementovanými algoritmami som vykonal množstvo testov a experimentov s ohľadom na uvedené kritériá. Pri testovaní som zvolil dostatočné množstvo testovacích množín, ktoré simulujú reálne podmienky. Boli využité tiež vzorky dát z reálneho nasadenia v rámci univerzitnej siete. Vďaka tomu som získal relevantné výsledky, ktoré som následne analyzoval. Po spracovaní som ich uviedol v práci v podobe prehľadných tabuliek a grafov. Výsledky potvrdili očakávané vlastnosti algoritmov založených na štruktúre trie. Jednoznačne najefektívnejší algoritmus bol SST, ktorý vynikal svojimi nízkymi pamäťovými nárokmi a vysokou rýchlosťou. Jeho veľkú výhodu vidím v schopnosti efektívne využívať pamäťový priestor a hlavne je ako jeden z mála vhodný na použitie s protokolom IPv6. Metóda trie, podobne ako Tree Bitmap s parametrom $n \geq 8$, sa ukazujú ako absolútne nepoužiteľné pre svoje obrovské pamäťové nároky. Tree Bitmap pre $n < 8$ dosahuje dobrú rýchlosť a priemerné pamäťové nároky, čo však kompenzuje jednoduchá hardwarová implementácia. Preto je tento algoritmu stále veľmi populárny v reálnom nasadení. Veľmi dobré výsledky dosiahol i môj algoritmus multi-match, ktorý je efektívny hlavne pri použití s malými množinami a v mnohých prípadoch prekonáva dokonca metódu SST. Výsledky ukazujú, že by mohol úspešne konkurovať používaným metódam hlavne vďaka svojej bezkonkurenčnej rýchlosti vyhľadávania.

Túto bakalársku prácu som vytvoril v rámci výskumnej skupiny ANT@FIT, ktorá sa zaoberá akceleráciou časovo kritický operácií sieťových technológií. Táto skupina aktuálne pracuje na knižnici experimentov, ktorá sa venuje napríklad klasifikácii paketov, správou sieťových tokov, analýzou hlavičky paketu a jednou z oblastí výskumu je tiež vyhľadávanie najdlhšieho zhodného prefixu. Všetky metódy ktoré som implementoval sa stanú súčasťou tejto knižnice. Výsledky mojej práce preto majú svoj význam a nájdu uplatnenie v ďalšom rozvoji a výskume. Keďže sa jedná o reálny projekt, sú tu mnohé cesty, ktorými by táto práca mohla pokračovať. Predovšetkým je potrebné neustále optimalizovať a vyvíjať nové LPM metódy. V tejto práci som sa zameral len na 4 metódy. Existuje však mnoho ďalších, ktoré by bolo vhodné zahrnúť do experimentov a porovnať ich vlastnosti. Vybrané metódy by sa tiež mali podrobne otestovať s použitím protokolu IPv6. A pri tom všetkom musíme samozrejme myslieť na budúcu hardwarovú implementáciu.

Literatúra

[1] Deering, S., Hinden, R.: *Internet Protocol, Version 6 (IPv6) Specification* [online]. Aktualizované 1998-12-01 [cit. 2010-04-18]. Dostupné na URL: <http://www.ietf.org/rfc/rfc2460.txt>

[2] Eatherton, W. N.: *Hardware-based internet protocol prefix lookups*. Dissertation thesis, Washington University – Department of Electrical Engineering, Saint Louis, Missouri, 1999.

[3] Eatherton, W., Dittia, Z., Varghese, G.: *Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates*. SISGCOMM Computer Communication Review, 2004.

[4] Hauger, S., Mutter, A, et al.: *Packet Processing at 100 Gbps and Beyond: Challenges and Perspectives*. Technische Universität München, Arcisstraße 21, 80290 Munich, Germany, 2009.

[5] Kořenek, J.: *Accelerated Network Technologies* [online]. [cit. 2010-04-25]. Dostupné na URL: <http://merlin.fit.vutbr.cz/ant/home/index.html>

[6] Lhotka, L., Novotný, J.: *Liberouter: Programmable hardware* [online]. Aktualizované 2010-03-31 [cit. 2010-04-25]. Dostupné na URL: <http://www.liberouter.org/>

[7] Piscitello, D., Stephen C., et al.: *Survey of IPv6 Support in Commercial Firewalls*. ICANN Security and Stability Advisory Committee, 2007.

[8] Postel, J.: *Internet Protocol* [Online]. Aktualizované 1981-09-01 [cit. 2010-04-18]. Dostupné na URL: <http://www.ietf.org/rfc/rfc791.txt>

[9] Puš, V.: *Klasifikace paketů s využitím technologie FPGA*. Diplomová práce, Vysoké učení technické v Brně – Fakulta informačních technologií, Brno, 2008, vedoucí práce Jan Kořenek.

[10] Shinder, D. L.: *Počítačové sítě*. SoftPress, Praha, 2003. ISBN 80-86497-55-0, 752 s.

[11] Song, H., Turner, J., Lockwood, J.: *Shape Shifting Tries for Faster IP Route Lookup*. Proceedings of the IEEE International Conference on Network Protocols (ICNP), Boston, MA, Nov. 6, 2005, pp. 358-367.

[12] Tobola, J.: *Vyhledání nejdelšího prefixu*. Pojednání k tématu disertační práce, Vysoké učení technické v Brně – Fakulta informačních technologií, Brno, 2009, vedoucí práce Vladimír Drábek.