VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

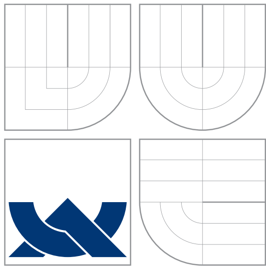
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ALGORITMY KLASIFIKACE PAKETŮ

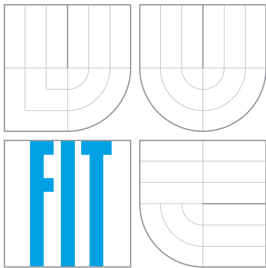
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JIŘÍ MACHALA
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ALGORITMY KLASIFIKACE PAKETŮ

PACKET CLASSIFICATION ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JIŘÍ MACHALA
AUTHOR

VEDOUCÍ PRÁCE — Ing. MICHAL KAJAN
SUPERVISOR

BRNO 2012

Abstrakt

Hlavním cílem této práce je interpretace a implementace algoritmu HyperCuts. Zabývá se však i problémem klasifikace paketů obecně - shrnuje teoretické základy klasifikace a popisuje nejznámější klasifikační metody. Zaměřuje se především na vyhodnocení HyperCuts v porovnání s algoritmem HiCuts a analyzuje přínos optimalizací popsaných v odoborných článcích. Navrhuje nový způsob snížení počtu přístupů do paměti kombinací HyperCuts a HiCuts na jednom klasifikátoru.

Abstract

The main aim of this work is interpretation and implementation of HyperCuts algorithm. It also covers the problematic of packet classification in general - it sums up the basic classification theory and introduces well-known classification methods. It concentrates mostly on evaluation of HyperCuts in comparison to HiCuts algorithm and analyses the assets of optimizations described in professional literature. It proposes a new way to limit the number of memory accesses by combining HyperCuts and HiCuts for single classifier.

Klíčová slova

klasifikace paketů, HyperCuts, HiCuts, implementace, porovnání

Keywords

packet classification, HyperCuts, HiCuts, implementation, comparison

Citace

Jiří Machala: Algoritmy klasifikace paketů, bakalářská práce, Brno, FIT VUT v Brně, 2012

Algoritmy klasifikace paketů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Kajana.

....
Jiří Machala
23. května 2012

Poděkování

Chtěl bych poděkovat vedoucímu práce, Ing. Michalu Kajanovi, za jeho vstřícnost a trpělivost a také své rodině za podporu při práci.

© Jiří Machala, 2012.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **4**

2 Teorie **6**
2.1 Architektura TCP/IP 6
2.2 Klasifikace 6
2.2.1 Klasifikační pravidlo 6
2.2.2 Klasifikátor 7
2.2.3 Geometrická reprezentace 7

3 Způsoby klasifikace podle techniky vyhledávání **9**
3.1 Lineární průchod klasifikátorem 10
3.2 Přímý přístup k pravidlu 10
3.2.1 TCAM 10
3.3 Vyhledávání s využitím bitového součinu 10
3.3.1 BitVector algoritmus 10
3.4 Dekompozice problému 11
3.4.1 Vyhledání nejdelšího shodného prefixu 11
3.4.2 Pseudopravidla 12
3.5 Použití rozhodovacích stromů 12
3.5.1 Algoritmus HiCuts 15
3.5.2 Algoritmus HyperCuts 17
3.5.3 Optimalizace zpracování rozhodovacích stromů 18

4 Návrh implementace **21**
4.1 Problematická interpertace popisu HyperCuts 21
4.2 Navrhovaná interpretace algoritmu 22
4.3 Stručný popis struktury implementace 23
4.3.1 Netbench 23
4.3.2 Problémy v implementaci 24

5 Zhodnocení implementace HyperCuts **25**
5.1 Efekt eliminace překrývaných pravidel 25
5.2 Výhodnost rozdělení klasifikátoru do dvou stromů 26
5.3 Přínos předávání společných pravidel mateřskému uzlu 27
5.4 Vliv zmenšení regionů a jejich spojování na výkonnost algoritmu 28
5.5 Srovnání HyperCuts a HiCuts 29
5.6 Zrychlení kombinací HiCuts a HyperCuts 29
5.7 Srovnání s MSCA a DCFL 30

6 Závěr **33**

A Obsah CD **36**

Seznam obrázků

2.1 Grafické znázornění problému klasifikace 7

3.1 Ilustrační příklad funkce Bitvector algoritmu 11
3.2 Prefixový strom 12
3.3 Dělení uzlu v geometrické reprezentaci 13
3.4 Strom vzniklý dělením regionu 13
3.5 Co je „cut“ 14
3.6 Vliv bucket size na výšku stromu 14
3.7 Srovnání různých bucket size 15
3.8 Dělení uzlu na více dimenzích v jednom kroku 17
3.9 Strom vzniklý dělením dvou dimenzí 17
3.10 Přesunutí replikovaných pravidel do mateřského uzlu 18
3.11 Slučování uzlů 19
3.12 Zmenšení oblasti uzlu podle vnitřních pravidel 19

5.1 Porovnání celkové spotřeby paměti. Přípona sum znamená, že je uveden součet pro dva stromy, all implikuje klasifikátor v jednom stromě. 27
5.2 Porovnání maximálních hloubek listových uzlů. Přípona „sum“znamená, že je v grafu zobrazen součet obou hlubek pro daný algoritmus a „all“že je celý klasifikátor v jednom stromě. 28
5.3 U stromů s přízviskem *pushing* jsou pravidla společná pro všechny potomky vytlačena vzhůru (*pushed up* do rodičovského uzlu. 29
5.4 Porovnání účinků optimalizací spojování a ořezu uzlů na výšku stromů *HyperCuts* a *HiCuts*. U stromů s popiskem merging je proveden optimalizace. Použité parametry byly *spfac*=4 a *bucket size*=10 30
5.5 Merge Velikost 31
5.6 Porovnání celkové výšky HyperCuts a HiCuts pro jednotlivé stromy 31
5.7 Porovnání celkové spotřeby paměti HyperCuts a HiCuts pro jednotlivé stromy 32
5.8 Porovnání rychlosti HyperCuts, HiCuts a kombinace HyperCuts a Hicuts. . 32
5.9 Porovnání spotřeby paměti HyperCuts, HiCuts a kombinace HyperCuts a Hicuts. 32

Kapitola 1

Úvod

Jen před několika měsíci oslavila Česká republika dvacáté výročí prvního oficiálního připojení k Internetu. A zatímco tenkrát bylo u nás připojení ke světové sítí záležitostí jen pár privilegovaných jedinců několika málo vybraných institucí, dnes je nejen díky rozmachu mobilních technologií počet zařízení schopných komunikovat na internetu větší než počet našich obyvatel.

S tím, jak se počet uživatelů masivně zvyšoval, začaly růst také nároky na rychlost a s tím související infrastrukturu. Zatímco rychlost běžných procesorů se za tu dobu zvedla z ani ne stovky Mhz na dnešních, i při uvážení více jádrových procesorů na maximálně desítky GHz, rychlost sítí rostla ještě výrazně rychleji – zatímco v únoru 1992 mělo jediné pevné připojení do zahraničí 19,2 kb/s, dnes je Česko propojeno s celým světem linkami o kapacitě desítek Gb/s a někteří ISP již běžně nabízejí domácnostem agregované stomegabitové připojení.

Pro zajištění kvality a bezpečnosti síťových služeb je třeba provoz na síti sledovat a regulovat. Tuto funkci zastává jednak firewall – služba která podle administrátorem definované sady pravidel určuje, jaký typ provozu povolit a jaký ignorovat, a jednak systém QoS(Quality of Service), který řídí tok paketů podle jejich důležitosti a vybírá pro jednotlivé toky paketů, zda je pozdržet nebo upřednostnit, většinou v závislosti na nutnosti dosáhnout pro danou službu její maximální latenci provozu.

V souvislosti se zmíněným nárůstem rychlosti a rozmanitosti provozu v sítích (a s tím spojeným zvyšováním nároků na komplexnost vytvářených pravidel) a přitom jen zlomkově se zrychlujícími procesory je potřeba nalézat nové způsoby efektivní klasifikace paketů. A právě klasifikace paketů (se zaměřením na zhodnocení algoritmu *HyperCuts*) je hlavním tématem této práce.

V kapitole 2 stručně nastíním strukturu síťového vrstvového modelu TCP/IP a jeho souvislost s klasifikací paketů a celkově uvedu čtenáře do problematiky klasifikace, jejích nároků apod.

Existuje několik základních typů přístupu ke klasifikaci. Z řady naprosto vybočují TCAM (ternární asociativní) paměti, které jsou hardwarovým nealgoritmickým (a hlavně velmi nákladným) řešením, jinak jde obvykle o řešení založené na algoritmech, někdy vhodné spíše pro softwarovou a jindy zase hardwarovou implementaci. Tyto způsoby řešení problému klasifikace bych rád stručně popsal v kapitole 3 s důrazem kladeným na metody využívající rozhodovacích stromů, konkrétně algoritmy *HiCuts* a *HyperCuts*.

V kapitole 4 následně popíšu svoji implementaci algoritmu *HyperCuts*, přiblížím také, jak jsem vyřešil nejasností v originálním popisu algoritmu, a zmíním ještě několik dalších souvislostí.

A v kapitole 5 zhodnotím výsledky testování své implementace *HyperCuts*, vyhodnotím efektivitu jednotlivých optimalizací ze třetí kapitoly, a stejně jako autoři *HyperCuts* srovnám jeho efektivitu s algoritmem *HiCuts*, s tím rozdílem, že se u *HiCuts* pokusím využít optimalizace, které byly navrženy až pro *HyperCuts*.

Kapitola 2

Teorie

2.1 Architektura TCP/IP

Na síťovou architekturu lze pohlížet jako na hierarchii několika vrstev. Kromě referenčního modelu ISO/OSI dělícím síťový provoz na 7 vrstev, existuje jednodušší a s reálným využitím blíže spojený vrstvový model TCP/IP.

1. vrstva se nazývá „vrstva síťového rozhraní“ a zajišťuje přístup přímo k fyzickému médiu. Příkladem takové sítě je Ethernet a poli, které se dají z této vrstvy využít při klasifikaci (ale zpravidla se nevyužívají) je pak zdrojová a cílová MAC adresa.
2. vrstva aneb „síťová vrstva“ se zabývá předáváním datagramů, směrováním a s tím související adresací. Tato vrstva je implementována na všech síťových prvcích a zastřešuje tak často rozdílné implementace první vrstvy. Příkladem protokolů na této vrstvě jsou IP a ICMP a poli používaných ke klasifikaci z této vrstvy jsou zdrojová a cílová adresa a protokol.
3. vrstva, také známá jako transportní vrstva, poskytuje abstrakci nad sítí a umožňuje tak např. zajištění spolehlivého přenosu dat. Protokoly na této vrstvě jsou UDP (nespojovaný, nespolehlivý) a TCP (spojovaný, spolehlivý) a pole z hlavičky této vrstvy využívaná ke klasifikaci jsou zdrojový a cílový port, které adresují službu čtvrté vrstvy a u TCP příznaky spojení. Tato vrstva je implementována jen na zařízeních, která s ní potřebují pracovat, tedy hlavně na koncových bodech.
4. vrstva, zvaná aplikační, pak definuje komunikaci přímo mezi uživatelskými službami. Protokoly této úrovně jsou např HTTP nebo SSH a data z této úrovně se při klasifikaci neužívají.

2.2 Klasifikace

Klasifikace se nazývá proces, kdy je pro vstup, který obvykle sestává z hodnot obsažených v hlavičce příchozího paketu, vybráno jedno nebo více pravidel z klasifkátoru.

2.2.1 Klasifikační pravidlo

Pravidlo je soustava podmínek, kde každá odpovídá některému poli z hlavičky paketu. Matematicky je zapsáno jako: $(n+1)$-tice $R\colon (p, a_1, a_2, \ldots, a_n)$, kde $p \in \mathbb{N}$ je priorita a a_i

jsou příslušné rozsahy. Každé pravidlo tak specifikuje třídu, do které může paket na základě těchto podmínek náležet.

Typy podmínek:

- hodnota – hodnota v odpovídajícím poli paketu musí být stejná
- rozsah – hodnota v odpovídajícím poli paketu se musí nacházet v tomto rozsahu. Některé rozsahy (jako IP adresy sítě) je ve zvyku zadávat ve formě prefixu. K dané hodnotě (ve dvojkové soustavě) náleží hodnota masky, která říká, kolik bitů zleva je daných a tím pádem naopak kolik bitů zprava má libovolnou hodnotu. Každý prefix lze přímo převést na rozsah tak, že spodní mez vznikne nastavením nemaskovaných bitů na 0 a horní hranice nastavením těchto bitů na 1. Naopak každý rozsah lze převést na soustavu prefixů.[1]

Lze tedy říci, že každá podmínka se dá vyjádřit jako jeden rozsah a z toho také vychází geometrická reprezentace klasifiakce.

Dále je ještě součástí pravidla číslo označující jeho prioritu, protože v případech, kdy jednomu paketu odpovídá více pravidel, je nutné určit, které pravidlo má přednost v případě, kdy musí být vybráno pouze jediné. Podle zvyklosti má přednost vždy pravidlo s nižším číslem priority.

2.2.2 Klasifikátor

Někdy označovaný také jako databáze pravidel, klasifikátor je množina pravidel definovaná administrátorem tak, že každé pravidlo má unikátní prioritu a přiřazenou akci. Ta má vykonat v případě, že paket spadá do třídy definované tímto pravidlem. Matematicky to lze zapsat jako $K = \{R; R_i \cdot p = R_j \cdot p \Rightarrow i = j\}$.

2.2.3 Geometrická reprezentace

Problém klasifikace lze názorně předvést v geometrické reprezentaci (viz diagram 2.1).

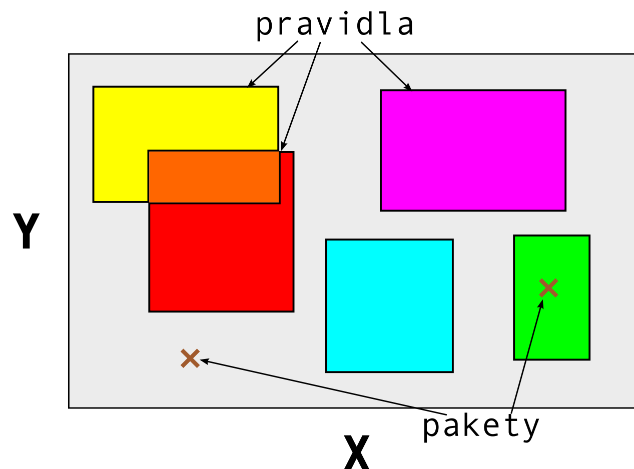


Diagram 2.1: Grafické znázornění problému klasifikace

[1] Reprezentace ve formě několika prefixů může být výhodná při hardwarové implementaci

Každé pole z hlavičky paketu (tedy většinou zdrojová a cílová IP adresa, zdrojový a cílový port a protokol), které je využité v pravidlech v klasifikátoru, definuje jeden diskrétní rozměr rozdělený na počet hodnot. Ten je pro změnu definován počtem možných hodnot nabývaných tímto polem (tzn. např. pro cílový a zdrojový port je to 2^{16} částí). V případě, že pak na tyto rozměry pohlédneme jako na více-dimenzionální prostor, můžeme libovolný paket zobrazit jako bod v tomto prostoru. Pravidla pak budou objekty o stejném počtu dimenzí jako tento prostor (pravidlo samozřejmě může být i bod, tedy objekt, který má ve všech rozměrech délku 1).

Přiřazení paketu do třídy, která je definována klasifikačním pravidlem, pak v geometrické reprezentaci znamená najít objekty znázorňující pravidla, ve kterých se bod znázorňující paket nachází. Zpravidla je pak z těchto objektů nutné vybrat ten, který reprezentuje pravidlo s nejvyšší prioritou.

Kapitola 3

Způsoby klasifikace podle techniky vyhledávání

V předchozí kapitole byly položeny základy pro pochopení problematiky klasifikace, v této kapitole budou představeny základní přístupy, které jednotlivá řešení využívají, a také jejich klady a zápory.

Aby však bylo možné jednotlivá řešení efektivně srovnávat, je nutné nejdříve definovat metriky, které pro klasifikaci budeme využívat:

- **rychlost**

  Je přirozeným požadavkem, aby proces klasifikace paketu vždy probíhal co nejrychleji. Cílem by vždy měl být stav, kdy dané řešení žádným způsobem neomezuje celkovou propustnost sítě.

  Přestože lze jednotlivá řešení porovnat na stejných počítačích za identického síťového provozu a získat tak reálné porovnání rychlosti, některá řešení, jejichž efektivita je závislá na paralelním zpracování a která tedy bývají implementována přímo v hardware, tomu zamezují. Proto bývá často v odborné literatuře pro měření rychlosti použit jako metrika počet přístupů do paměti v nejhorším možném případě.

- **potřebná paměť**

  Dalším kritériem je množství paměti, které je daným řešením vyžadováno. Toto kritérium úzce souvisí s předchozím bodem, protože například v případě firewallu spuštěného na klasickém PC je docela zásadní fakt, zda přístupy do paměti budou směřovat do vyrovnávací paměti procesoru nebo RAM.

- **cena**

  Při praktickém nasazení je cena asi tím nejdůležitějším kritériem, které často vymezuje možné kombinace rychlosti a spotřebované paměti. Například při implelementaci s pomocí technologie FPGA (field-programmable gate array) se může stát, že restriktivním faktorem je počet blokových pamětí na čipu, což povede k nutnosti zakoupení dražší karty s lepším FPGA čipem.

- **počet pravidel a jejich vlastnosti**

  Některá řešení mohou obsahovat omezení maximálního počtu pravidel nebo být efektivní pouze pro klasifikátory s určitými vlastnostmi. Někdy takové algoritmy mohou

být efektivní pouze pro 2 dimenze nebo pro klasifikátory s vysokým, respektive nízkým, počtem unikátních hodnot nebo prefixů.

3.1 Lineární průchod klasifikátorem

Toto je naprosto nejnaivnější způsob klasifikace, který sice nepotřebuje žádnou paměť kromě paměti potřebné na uložení pravidel, nicméně časová složitost nalezení klasifikátoru je $O(N \cdot K)$. To je s výjimkou klasifikátorů obsahujících pouze několik pravidel neakceptovatelné. I tak má tento přístup v klasifikaci stále své místo, protože v některých řešeních (viz dále) se lineární prohledávání úspěšně používá jako součást určitých algoritmů.

3.2 Přímý přístup k pravidlu

Druhým krajním případem by byl přístup, kdy bychom vytvořili blok paměti tak velký, že by znázorňoval vícerozměrný prostor popsaný v oddílu o geometrické reprezentaci klasifikace. Každý by bod obsahoval číslo označující pravidlo s nejvyšší prioritou pro paket adresující tento bod. Nicméně pro běžné klasifikátory (zdrojová a cílová IPv4 adresa, zdrojový a cílový port a protokol) by to vyžadovalo pamět o velikost $2 \cdot 2^{32} \cdot 2 \cdot 2^{16} \cdot 2^{8} = 2^{104}$ bytů i za předpokladu, že by klasifikátor obsahoval pouze 256 pravidel.[7]

Existuje ale i hardwarové řešení které získání daného pravidla v jediném korku umožňuje.

3.2.1 TCAM

Třístavové asociativní paměti jsou typ pamětí, které s použitím komparátorů podporují vyhledávání v paměti podle obsahu tak, že pro slovo uložené v paměti je možné nastavit zda má být daný bit nastaven na 0, 1 nebo ignorován. Tím přímo umožňuje využít reprezentace podmínek ve formě prefixů. Jak ale bylo zmíněno dříve, obecný rozsah bývá často rozdělen na více prefixů, takže vzniká nutnost do TCAM ukládat, místo původního pravidla, více pravidel obsahujících kartézský součin prefixů v jednotlivých polích (za předpokladu, že se pravidla ukládají podle priority, bude nové pravidlo přidáno pouze v případě, že nekoliduje s nějakým předchozím). To v kombinaci s omezenou kapacitou a relativně vysokou cenou TCAM vede k tomu, že často musí být hledána jiná (pomalejší a levnější) řešení.

3.3 Vyhledávání s využitím bitového součinu

Příkladem takovýchto řešení je dále popsaný BitVector algoritmu[5]. Nicméně bitový součin se používá i v dalších algoritmech – například v RFC[3] – algoritmu rekurzivních toků, který využívá bitový součin k určení sousednosti pravidel v ndimenzionálním prostoru a vyhledává pak ve skupinách sousednosti namísto originálních pravidel.

3.3.1 BitVector algoritmus

Tento algoritmus nejprve vytvoří bitové pole takové délky, jako je celkový počet pravidel v klasifikátoru. Dále pak pohlíží na každou dimenzi jako při geometrické reprezentaci, kdy všechny rozsahy v daném poli převede na maximálně $2n+1$[7] nepřekrývajících se rozsahů[1],

[1] rozsahy (0,5), (1,3) a (2,4) se převedou na (0,1), (1,2), (2,3), (3,5)

kde n je počet unikátních rozsahů v dimenzi. Ke každému z těchto rozsahů se přiřadí bitový vektor označující zda sem patří dané pravidlo (viz diagram 3.1).

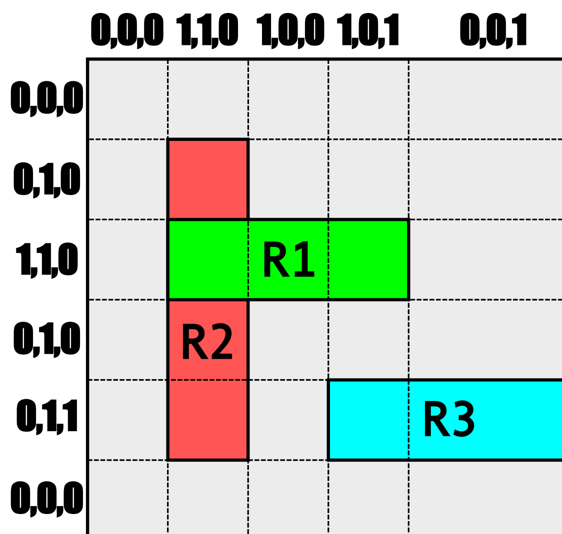


Diagram 3.1: Ilustrační příklad funkce Bitvector algoritmu

Vyhledávání pak probíhá tak, že se paralelně pro každou dimenzi vyhledá bitový vektor (to lze provádět s logaritmickou časovou složitostí například pomocí binárního vyhledávání) z rozsahu, kam spadá paket. Nad těmito vektory se pak provede bitový součin. Výsledný vektor pak bude obsahovat jedničky na místech označujících odpovídající pravidla. Za předpokladu, že byla pravidla seřazena podle priority, bude pozice prvního nenulového bitu shodná s číslem pravidla.

Toto řešení na první pohled vypadá velmi rychle, ale s větším počtem pravidel zde vyvstává problém, že délka bitového vektoru pro klasifikátor vyžaduje několikanásobný přístup do paměti pro jeho celé načtení. To nakonec vede k celkově vyšší časové náročnosti.

3.4 Dekompozice problému

Do této skupiny se řadí algoritmy, které nejdříve pro každou dimenzi vyhledávají prefix s nejdelší shodou (LPM[2]) s hodnotou příslušného pole ve vstupním paketu. Vzhledem k tomu, že takovýto nejspecifičtější prefix nebude pravděpodobně ve všech dimenzích patřit ke stejnému pravidlu, vyvstává problém pseudopravidel (viz níže) a následuje fáze, během které se díky nim vybere příslušné pravidlo.

3.4.1 Vyhledání nejdelšího shodného prefixu

Toho bývá nejčastěji dosaženo s pomocí datové struktury trie (zkratka z anglického *retrieval*; někdy též prefixový strom) nebo jiné struktury pracující na podobném principu. Trie je binární stromová struktura, kde každý uzel označuje jeden bit prefixu, takže cesta od kořene do uzlu tvoří prefix. Klasifikátor ale zpravidla nebude obsahovat všechny možné prefixy a tak v uzlech, kde končí některý prefix, bude uchováno jeho číslo.

Při vyhledávání hodnoty z hlavičky paketu se vždy uloží číslo posledního procházeného prefixu, čímž se získá nejdelší shodný prefix pro danou dimenzi.

[2]Longest Prefix Match

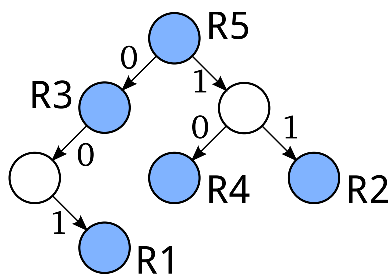


Diagram 3.2: Trie pro prefixy: R1:001*, R2:11*, R3:0*, R4:10* a R5:*

Existují i algoritmy, kdy je nutné vyhledat všechny shodné prefixy, nejen nejdelší, v takových případech se při procházení trie ukládají všechny prošlé prefixy.

3.4.2 Pseudopravidla

Pseudopravidla jsou množina nových pravidel vytvořených tak, že každá platná kombinace LPM na jednotlivých dimenzích je (z pohledu geometrické reprezentace) multidimenzionálním regionem, který je pokryt některým z původní sady pravidel. Jejich generování probíhá tak, že se pro každou dimenzi nejdříve vyhledají všechny existující prefixy a pak se pro každé pravidlo provede kartézský součin nad jeho poli. Každé pole bude obsahovat všechny stejné a specifičtější prefixy (ze všech prefixů v dané dimenzi) jako je původní prefix pravidla[7]. Toto je dále komplikováno nutností převádět rozsahy na více prefixů, viz. 3.2.1.

Potom lze pro všechna pravidla (včetně nových pseudopravidel) zřetězit jejich prefixy, na tuto hodnotu použít hashovací funkci a na dané místo tabulky uložit ukazatel na původní pravidlo[3]. Když je pak vybudována tato hashovací tabulka, lze, po nalezení LPM pro každé pole paketu, přistoupit k odpovídajícímu pravidlu v konstantním čase.

Nevýhodou tohoto řešení je vysoká paměťová náročnost, daná tím, že vytvořených pseudopravidel je průměrně dvěstěkrát víc než původních pravidel[2]. Toto lze částečně eliminovat tím, že se pseudopravidla budou ukládat pouze při budování do pomalejší paměti, nicméně zde zůstává problém s nutností velké hashovací tabulky, aby se minimalizoval počet kolizí a nízké rychlosti v případě kolize pravidel v hashovací tabulce a s tím související nutnost lineárního vyhledávání odpovídajícího pravidla.

Existují však i metody jako například MSCA[2], které dělí pravidla do několika skupin tak, že potom vznikne daleko méně pseudopravidel, to je ale už nad rámec této práce.

3.5 Použití rozhodovacích stromů

Tento způsob přístupu ke klasifikaci vychází z geometrické reprezentace klasifikátoru variant paketů, tzn. jako nějaké hyperkrychle (budu ji takto označovat po vzoru anglických prací, přestože je vzhledem k tomu, že mohutnost jednotlivých rozměrů je různá, slovo „krychle“ tedy zavádějící). Zde každý rozměr představuje jedno pole hlavičky paketu. Tento region, nazvěme ho i se pak dělí N rovnoběžnými rovinami (dá se říci seká, z toho je odvozen i název dalších algoritmů), čímž z něj vznikají menší části, které se opět dělí atd. Počet částí na které je dimenze rozdělena nazveme $n(i)$. Tímto způsobem se pak vytváří strom, v jehož

[3] respektive na zřetězený seznam, vzhledem k možnosti kolizí

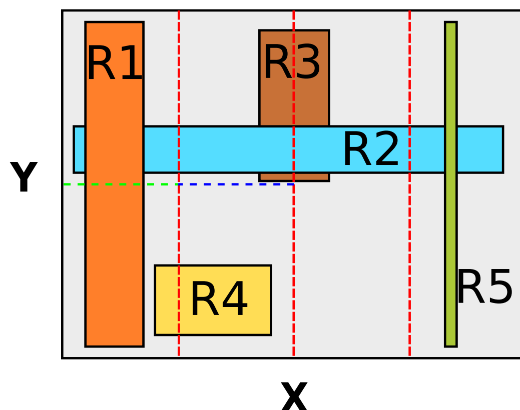


Diagram 3.3: Pro první dělení byla vybrána dimenze X, ve druhé úrovni pak byly dva subregiony rozděleny podle dimenze Y.

listech jsou nakonec uložena pravidla. Pro dvourozměrný klasifikátor je to znázorněno na obrázku 3.3, příslušný strom je pak na diagramu 3.4.

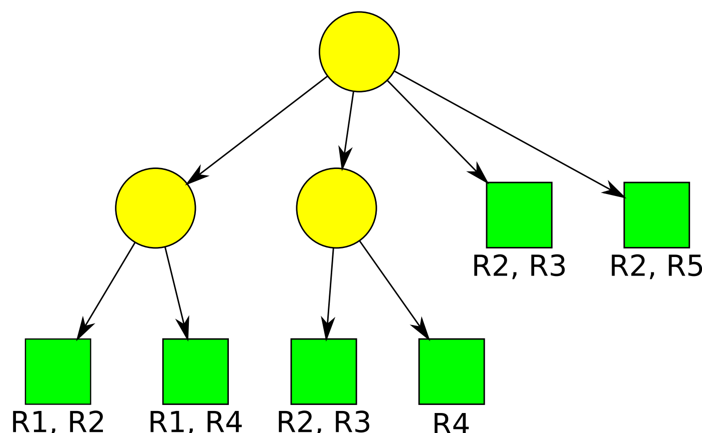


Diagram 3.4: Výsledný strom vzniklý dělením regionu z diagramu 3.3.

V anglické literatuře někdy slovo „cut“ vyjadřuje jednu z částí regionu, na které byl rodičovský uzel rozdělen, a jindy operaci zvýšení počtu poduzlů na dvojnásobek (viz diagram 3.5). To znamená, že N cutů, rozdělí rodičovský uzel na 2^N potomků. Já tento pojem budu používat druhým zmíněným způsobem.

Vyhledávání

V závislosti na příchozím paketu se rozhoduje, ve kterém z podstromů je třeba hledat pravidlo, a to se opakuje tak dlouho, dokud pravidlo není nalezeno. Pro snížení paměťových nároků je pak možné tento přístup zkombinovat s lineárním prohledáváním pravidel způsobem, kdy určíme určitý počet pravidel v uzlu (říkejme tomuto počtu třeba *bucket size*), pro které se stavový prostor už dále nedělí.

Je vhodné poznament, že v mnoha případech tento přístup vede, spolu s heuristikami pro volbu počtu rozdělení regionu, až k takovému snížení hloubky stromu, že je velmi výhodné nejen z hlediska paměti, ale i z hlediska rychosti. Viz diagram 3.6 s klasifikátorem v geoemtrické reprezentaci a diagramy ukazující rozdíl ve výškách stromu při použití *bucket*

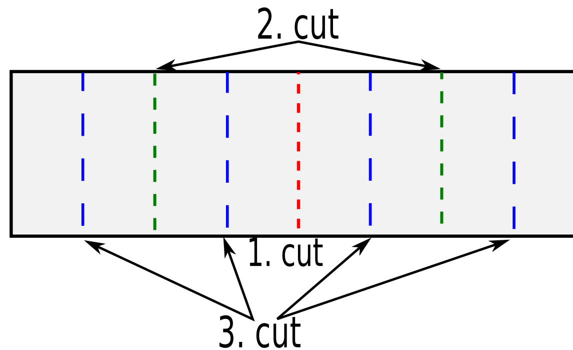


Diagram 3.5: Při dělení regionu v jednom kroku na více částí, se může jeden „cut“ skládat z více rovin.

size = 1 (3.7a) a *bucket size* = 2 (3.7b).

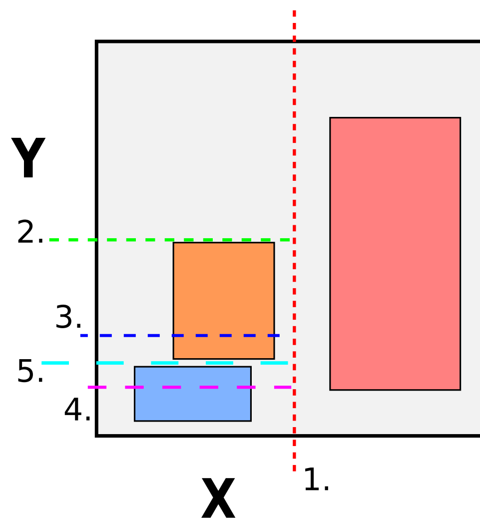


Diagram 3.6: Příklad neefektivního dělení při nízkém *bucket size*. Výsledek viz 3.7

Stavba stromu

Hlavním problém zde však je, jak efektivně vystavět strom tak, aby měl co nejmenší hloubku (kvůli rychosti vyhledávání) při co nejmenším využití paměti. To lze při stavbě ovlivnit dvěma základními rozhodnutími.

Prvním je určení počtu částí na které se hyperregion v dané úrovni dělí. Například je možné v jednom kroku rozdělit strom na 1000 podstromů. Tím se strom s největší pravděpodobností radikálně zmenší, protože množství dalších dělení nutných k separování počtu pravidel menších než požadovaný *bucket size* bude pravděpodobně velmi malé. Taková volba však s největší pravděpodobností radikálně zvýší využití paměti, protože způsobí velké množství duplikací pravidel.

Druhou volbou je výběrem dimenze (nebo dimenzí), na které se uzel bude dělit, tak aby příští dělení bylo opět co nejefektivnější. Zpravidla tedy tak, aby se region rozdělil do částí, pod které spadá co nejnižší počet pravidel - tím dojde k nejméně replikacím pravidel ve

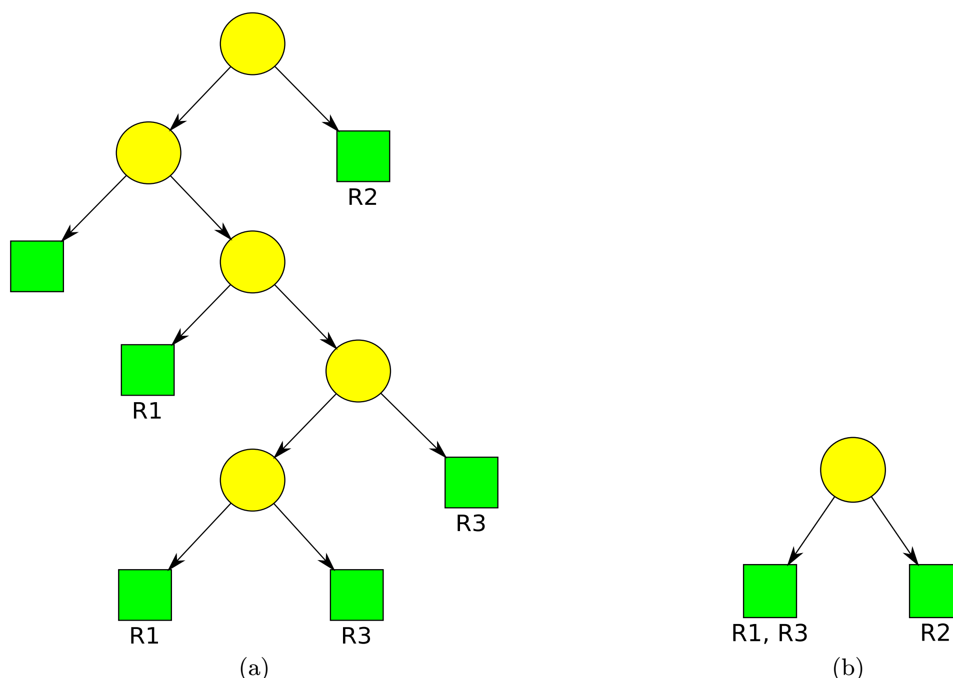


Diagram 3.7: Rozdíl výšky stromu při *bucket size* = 1 a *bucket size* = 2 vytvořeném z regionu z diagramu 3.6

více uzlech.

Vytvoření optimálního rozhodovacího stromu pro daný stavový prostor je NP-úplný problém[3], takže snaha o dosažení nejlepšího existujícího řešení by byla neúnosně náročná. Takže se hledaly cesty, jak se mu co nejvíce přiblížit a byly proto vytvořeny, kromě jiných, dále popsané algoritmy využívající různé heuristiky. Kromě předem určeného dělení, k jehož volbě se došlo analýzou velkého množství reálně používaných databází klasifikátorů, byly navrženy algoritmy, které se pomocí daných pravidel a částečného prohledávání stavového prostoru snaží přizpůsobit použité množině pravidel. Jedním z prvních takto navržených algoritmů byl HiCuts.

3.5.1 Algoritmus HiCuts

Jméno *HiCuts* je zkratkou *Hierarchical Inteligent Cutting* a už název algoritmu přibližuje jeho fungování. Ze slova „hierachical“ lze odvodit, že se pakety klasifikují pomocí stromové struktury, „cutting“ napovídá, že tento strom vytváří „sekáním“ stavového prostoru a „intelligent“, že se snaží chovat chytře s ohledem na množinu vstupních pravidel.

Stručně se dá popsat jeho fungování, jako výběr jedné dimenze a počtu částí na které se strom v daném uzlu rozšíří.

Výběr počtu dělení

Nejdůležitější část *HiCuts* shrnuje pseudokód 1, který znázorňuje heuristiku, jež se v každém uzlu provede pro každou dimenzi (pokud už počet pravidel spadajících do tohoto regionu není menší než *bucket size*), kvůli určení počtu částí na které se by se dal uzel rozdělit[4]:

Algoritmus 1: Volba počtu dělení v uzlu – HiCuts

```
n = numRules(v)
numP = max(4, sqrt(n)) /* starting value of number of partitions to make at this
node */
done = 0
while not done do
    sm(C) = 0
    foreach rule in CollidingRuleSet(v) do
        sm(C)+ = numberofpartitionscollidingwithruler
    end
    sm(C)+ = numP
    if sm(C) < spmf(n) then
        numP = numP * 2/* double the number of partitions */
    end
    else
        done = 1
    end
end
```

Z kódu je zřejmé, že se provádí binární vyhledávání, kdy se při každém cyklu zkouší zvýšit počet navrhovaných částí (potomků) uzlu na dvojnásobek (tzn. přidá se jeden „cut"), dokud součet pravidel pod každým potomkem plus počet potomků nepřekročí hodnotu funkce míry využití *spmf* (space measure function). Tato funkce je definována jako funkce počtu pravidel obsažených v rodiči a volitelného parametru *spfac* (space factor). Matematicky vyjádřeno je to takto[4]:

$$\textit{spmf} \cdot \text{počet pravidel v } r \geq \sum \text{počet pravidel v každém dítěti } r + numP$$

Výběr dimenze

Tím je vyřešena záležitost výběru $nc(i)$, dalším krokem je vybrat, které z těchto i je nejlepší. Autoři *HiCuts* doporučují postupovat podle těchto kriérií:

Měli bychom vybrat dimenzi tak, že:

- Má potomka s nejmenším maximálním počtem pod něj spadajících pravidel ze všech potomků této dimenze.

  Předpokládáme, že počet pravidel v potomkovi dělený součtem pravidel ve všech potomcích je rozdělení pravděpodobnosti s počtem prvků rovným počtu potomků rodičovského uzlu. Snažíme se maximalizovat míru neurčitosti – to v praxi znamená vybrat dimenzi s nejvíce uniformní distribucí pravidel v potomcích[4]. Jak ale poznamenává [10], to může vést k případům, kdy se algoritmus bude do nekonečna snažit rozdělit dimenzi, která obsahuje pouze pravidla, která ji všechna celou pokrývají.

- Bude mít nejnižší součet pravidel ve všech potomcích.

- Pravidla na ní budou vytvářet nejvyšší počet unikátních rozsahů.

3.5.2 Algoritmus HyperCuts

HyperCuts[9] přímo navazuje na *HiCuts* a vychází z idee, že lepších výsledků by bylo možné dosáhnout, nebýt omezení v jednom kroku vytvářet podstromy pouze napříč jednou dimenzí(viz diagram 3.8).

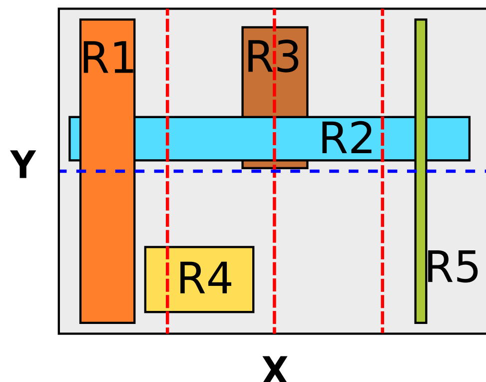


Diagram 3.8: Dělení regionu na dimenzích X (2 cuty) a Y (1 cut) v jediném kroku. Výsledný strom viz 3.9.

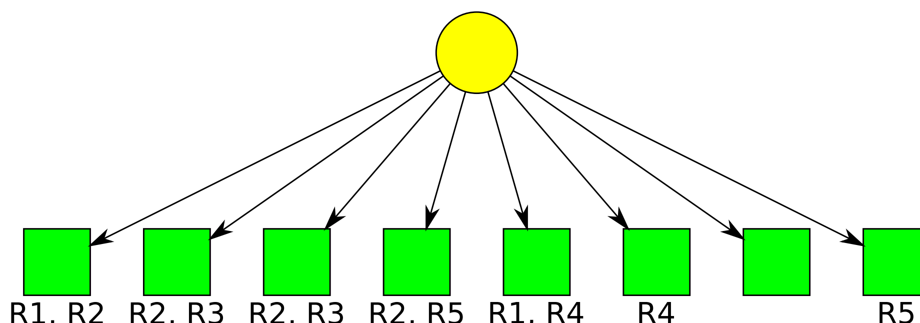


Diagram 3.9: Rozdělením na čtyři části v dimenzi X a dvě v Y vzniklo osm potomků uzlu.

Oproti *HiCuts* je zde jednoduší výběr celkového počtu „cutů“ – uzel se smí rozdělit na maximálně odmocninu z počtu pravidel v daném uzlu krát prostorový parametr *spfac*.

Pro určení pole nebo polí, která se rozdělí, se nejdříve provede předvýběr – pro další rozvažování se berou v úvahu pouze ty, které mají počet unikátních hodnot pravidel na daném rozsahu větší, než je průměr pro všechny dimenze. Následně se zbylé seřadí podle tohoto počtu a v případě stejných hodnot se seřadí od nejmenší.

Další postup je popsán, pravděpodobně z patentových důvodů, pouze v hrubých rysech, které tu shrnu. Jak konkrétně jsem se s implementací těchto instrukcí vyrovnal, pak popíšu v následující kapitole.

Pro každou z předvybraných dimenzí se určuje optimální počet rozdělení iterativním zdvojnásobováním tak dlouho, dokud nepřestane být zlepšení v poměru počtu pravidel na část nebo v počtu pravidel v nejhorší části dostatečné, popřípadě pokud se příliš navýší počet prázdných uzlů.

Následně se z těchto dimenzí vyberou ty, které vyhovují nastavenému prostorovému omezení.

Nakonec je tento výběr dimenzí dále optimalizován zvážením všech jeho možných podmnožin s tím, že se vybere ta s nejlepším poměrem prostor/čas.

3.5.3 Optimalizace zpracování rozhodovacích stromů

Přestože byly tyto optimalizace navržené v článcích autory *HiCuts*[4] a *HyperCuts*[9], měly by být zpravidla použitelné pro jakýkoliv podobný algoritmus.

Eliminace překrytých pravidel

V průběhu vytváření stromu se může stát, že hyperregion, který daný uzel pokrývá, obsahuje pravidlo, respektive pravidla, které je v dané oblasti kompletně překryto jiným pravidlem s vyšší prioritou. Je proto možné ho zde vynechat. Přestože je tento přístup popsaný jako optimalizace, dá se říct, že pro správné fungování *HiCuts* a *HyperCuts* je nezbytný, protože bez něj bude často jednomu paketu odpovídat více pravidel, než je *bucket size*, i když region odpovídající uzlu nemusí být dále dělitelný.

Přenesení společných pravidel do rodičovského uzlu

Autoři algoritmu *HyperCuts* si všimli, že se často stává, že pravidlo je replikováno, ve všech potomcích uzlu. Proto navrhli řešení, kdy při zpětném průchodu stromem zdola nahoru tato pravidla přesouvají z listů do rodičovských uzlů 3.10. Neplatí to však jen pro poslední úroveň stromu – v obvyklém případě kdy v klasifikátoru existuje výchozí pravidlo pokrývající celý region, bude toto postuponě přesunuto až do kořene stromu.

Stojí za povšimnutí, že tento postup, vzhledem k tomu že se postupuje zdola nahoru, nikdy nezmění výšku stromu, může pouze ušetřit paměť snížením počtu replikací v uzlech.

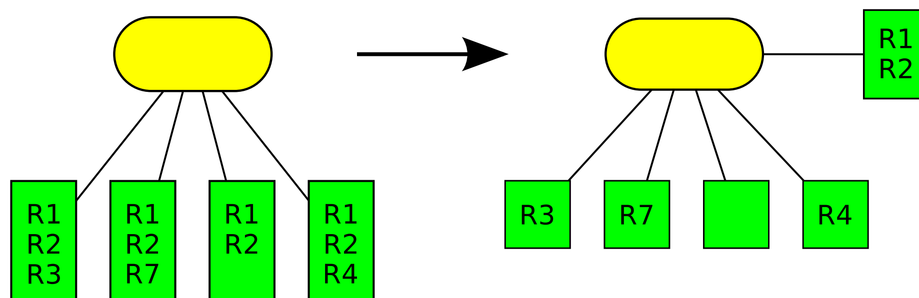


Diagram 3.10: Nachází-li se uzel ve všech potomcích rodičovského uzlu, lze jej přesunout o úroveň výše.

Existuje ještě obdoba tohoto přístupu, kde se používá dopředné přesouvání pravidel do uzlů. Místo postupného přesouvání pravidel z listů do rodičovských prvků se ukládají ihned, jakmile se při dělení v dané dimenzi pravidlo replikuje ve všech potomcích (i nelistech) – to vede k podstatně většímu množství takto přesunutých prvků a při vyhledávání odpovídajícího pravidla nese nebezpečí výrazně větší vyhledávací doby v lineárních seznamech po cestě do listového uzlu. Song proto doporučuje omezit se pouze na několik prvních úrovní stromu, kde tento postup, díky eliminaci pravidel s wildcards na daných dimenzích, může výrazně snížit celkovou hloubku.[10]

Spojování uzlů

Zatímco předchozí dvě optimalizace se zabývaly pouze ušetřením paměti listů, tato se snaží paměť uspořit jiným způsobem.

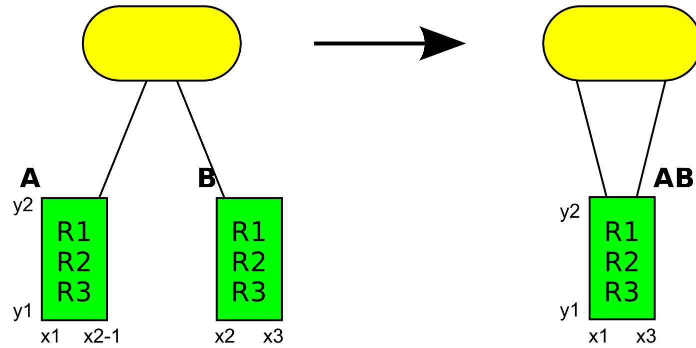


Diagram 3.11: Při budování stromu došlo k vytvoření uzlů A a B se kterými jsou asociována stejná pravidla. Proto mohou být sloučeny do uzlu AB, který pokrývá oblast obou původních uzlů.

Jakmile se region rozdělí na části, zkontroluje se pro sousední(z hlediska rozměrů dimenzí) poduzly, zda neobsahují přesně stejná pravidla. Jestliže ano, spojí se do jednoho uzlu (respektive bude v poli odkazů mateřského uzlu odkaz na stejný prvek). Nicméně, vzhledem k původnímu stavu, kdy je region pokrytý uzlem definovaný svým umístěním ve stromu, zde vzniká nutnost ukládat v uzlu jeho nové hranice.

Odstranění redundance v listových uzlech

Často se stává, že při rozdělení regionu vznikne několik listových uzlů, které obsahují stejná pravidla. Vzhledem k tomu, že listové uzly se už dále nedělí, je možné se bez jakékoliv újmy vyhnout jejich replikaci a nechat je všechny odkazovat na stejné místo v paměti.

K demonstraci tohoto se dá opět použít diagram 3.11, odmyslíme-li si z něj nový rozsah regionů (u listových uzlů lze rozměry regionu ignorovat).

Oříznutí regionu

V případě, že celý prostor regionu není pokryt pravidly, je vhodné region zmenšit a tak pravděpodobně zvýšit efektivitu příštích „cutů“. Stejně jako předchozí optimalizace to však s sebou nese nutnost uchovat v uzlu nové hranice hyperregionu.

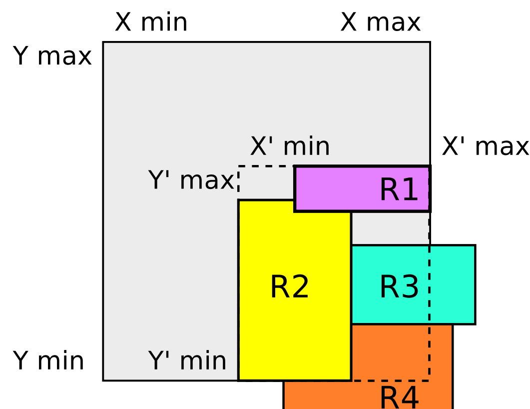


Diagram 3.12: Zatímco před oříznutím by jeden řez v kterékoliv z dimenzí pouze zvýšil hloubku stromu, teď bude následné dělení efektivnější.

Zvláštní strom pro pravidla s oběma IP adresami obsahujícími wildcards

Toto řešení bylo navrženo autory *HyperCuts* jako plnohodnotná součást tohoto algoritmu. Vychází z (někdy nesprávného) předpokladu, že ve zpracovávaných pravidlech bude pouze malé množství takovýchto IP adres a tím výrazně sníží paměťové nároky a následně i hloubku umožněním efektivnějšího dělení hlavního stromu bez těchto adres. Vzhledem k této skutečnosti, se budu tímto návrhem zabývat pouze jako optimalizací.

Kapitola 4

Návrh implementace

4.1 Problematická interpertace popisu HyperCuts

Jak bylo zmíněno v předešlé kapitole, článek popisující algoritmus je často relativně neurčitý a přestože říká, co se má udělat a často i jaké metriky se mají využít, v zásadních otázkách postrádá informace o tom, jak se to udělat – bylo tedy nutné si v některých případech domýšlet.

První krok, výběr dimenzí s nadprůměrným počtem unikátních prvků je jasný – nicméně ani na toto, jak prokázala jedna z prvních fází mého testování, nesmí být pohlíženo nekriticky. Sice je ta šance při stovkách až tisících pravidel na první pohled velmi nízká, ale při vytváření mnoha uzlů se může snadno stát, že všechny dimenze daného hyperregionu budou mít stejný počet unikátních prvků.

Následně má probíhat iterativně zjišťování optimálního počtu dělení pro jednotlivé dimenze, podobně jako u *HiCuts*. V případě nedostatečného zlepšení se vrátí k hodnotě z předchozí iterace, ale dimenze se vybere pouze pokud vyhovuje nastavenému prostorovému omezení.

Zde však vyvstává otázka, co znamená „podobně jako u *HiCuts*“. Použití stejného algoritmu jako u *HiCuts* a *spfac* parametru nastaveném na jedna, bude znamenat, že výchozí výběr počtu dělení bude nastaven na odmocninu z počtu pravidel v uzlu. To je také maximální počet dělení pro všechny dimenze dohromady, což může způsobit, že ve výsledku mohou být vybrány i 4 z 5 dimenzí. Z toho každá s $nc(i)$ v nejhorším případě (v závislosti na umístění kontroly celkového počtu dělení pro *HyperCuts*) buď větším než celkové povolené maximum pro *HyperCuts* (případ, kdy se celkové omezení aplikuje až později), nebo stejné jako maximum (omezení se aplikuje při iteraci).

Z tohoto výběru by se pak měly každopádně zvážit všechny možné podmnožiny kombinací, což je tatáž věc, kterou autoři označili za výpočetně příliš náročnou.

Modifikace, kdy se z *HiCuts* algoritmu odstraní výchozí výběr počtu dělení v závislosti na počtu pravidel, bude při $spfac = 1$ znamenat, že se vybere několik dimenzí. Každá bude obsahovat právě jeden „cut“ (podmínka z minulého odstavce nebude ani zde uspokojena a tak nedojde k dalším iteracím), nebude muset proběhnout zvážení žádných podmnožin, protože počet dělení pro všechny dimenze bude zpravidla dodržen, a celkový strom bude velmi rychle růst do hloubky. V případě $spfac > 1$ zase bude součin počtu rozdělení na jednotlivých dimenzích často vyšší než povolené celkové maximum, ale tento problém je shodný pro všechna řešení, která nejdříve přidělují „cuty“ na samostatné dimenze.

Až dodnes existovaly v zásadě dvě veřejné implementace *HyperCuts*, přičemž každá z

nich přistupuje k původnímu popisu z úplně jiného směru, takže, než popíšu své vlastní řešení, pokusím se je stručně srovnat.

Implementace V. Mašíčka z Univerzity Karlovy

Tato implementace[6] se původním popisem algoritmu řídí jen velmi volně. Kromě očividné chyby, kdy vybírá dimenze pro dělení podle jejich rozsahu (tzn. na začátku vždy vybere IP adresy), místo počtu unikátních prvků, také určuje počet dělení regionu na konstantní počet částí, místo podle množství pravidel v uzlu. Na této implementaci je zajímavé, že se očividně zaměřila na část původního popisu, která říká:

> At the end of this procedure, the pruned set of dimensions is optimized further by considering every possible subset of the pruned set, and choosing the best subset in terms of a storage/time tradeoff.[9]

A tak u vybraných dimenzí zkouší všechny kombinace dělení uzlu na těchto rozměrech a vybírá z nich právě tu nejlepší.

Implementace H. Songa z Washington University in St. Louis

Toto řešení vychází z algoritmu pro určení optimálního počtu dělení dimenze v *HiCuts*, nicméně odstraňuje počáteční nastavení počtu rozdělení dimenze, takže iteruje od začátku. Jak bylo zmíněno výše, to velmi často vede k celkovému počtu rozdělení vyššímu než je maximum povolené *HyperCuts*.

To řeší tak, že následně snižuje počet rozdělení v dimezi (nebo dimenzích) s minimálním *nc(i)*, tak dlouho dokud součin *nc(i)* nevyhovuje omezení. To by mělo vést k výsledku, kdy nejvyšší *nc(i)* bude vybráno pro nejefektivněji rozdělenou dimenzi, nicméně staví na, často mylném, předpokladu, že iterativní zvyšování *nc(i)* přináší stejnoměrné zlepšení poměru prostor/čas. I tak je to velmi rychlý způsob jak určit výsledné *nc(i)* pro jednotlivé dimenze a, jak ukazují testy, na reálných datech má velmi slušné výsledky.[10]

4.2 Navrhovaná interpretace algoritmu

Já sám jsem se rozhodl k problému přistupovat takto:

Mé východisko bylo, že problém rozhodování o optimálním počtu dělení na jednotlivé dimenze lze rozdělit na několik variant, kdy na každou lze efektivně nahlížet jiným způsobem. Varianty dělím podle počtu předvybraných dimenzí (počet různých hodnot pravidel v dimenzi je nadprůměrný) v kombinaci s maximálním počtem rozdělení uzlu.

První varianta je stav, kdy existuje pouze jedna předvybraná dimenze. Zde je zbytečné testovat optimální počet dělení pro tuto dimenzi a lze ihned rozhodnout o přidělení maximálního počtu „cutů“.

Za druhý případ považuji situaci, kdy existuje malý počet způsobů dělení jednotlivých dimenzí. Za „malý“ pokládám přibližně tři dimenze a 8 částí (až 10 možností) nebo 4 dimenze a 4 části (také 10 možností) a nebo dvě dimenze a až 64 částí (tzn. až 7 možností, ale vzhledem k omezující podmínce na počet dělení v *HyperCuts*, $NC \leq spfac \cdot \sqrt{\text{num_rules}}$, relativně větší počet zpracovávaných pravidel). Zde lze díky relativně malému množství možných kombinací omluvit řešení hrubou silou a to i z důvodu, že zde odpadá iterativní

určování počtu dělení pro jednotlivé dimenze, čímž se čas naopak ušetří. Nevýhodou tohoto řešení je, že vždy volí maximální povolený počet dělení uzlu a tyto rozdělení distribuje mezi dimenze, což znamená, že u špatně dělitelných regionů může docházet k příliš agresivnímu „cutování" vedoucímu k přehnané spotřebě paměti.

Třetí varianta jsou případy, kdy je počet dimenzí a maximálního počtu dělení už příliš vysoký pro hledání ideálního řešení hrubou silou. Zde jsem se proto rozhodl napodobit Songovo řešení, tzn. použít část algoritmu *HiCuts* pro nalezení kandidátního počtu dělení pro jednotlivé dimenze a tyto pak iterativně snižovat, dokud jejich součin nevyhovuje celkové podmínce. Navíc jsem se však při vyhledávání počtu dělení na jednotlivých dimenzích pokusil respektovat původní návrh algoritmu a implementovat kontrolní funkci s funkčností odpovídající této části původního návrhu *HyperCuts*.

> If after a number of subsequent steps there is no significant change in the mean or the maximum number of rules in the child nodes, or there is a significant increase in the number of empty child nodes, then we backtrack and use the last known best value as the chosen number of splits to be made along the dimension under consideration.[9]

Zde ovšem vyvstává otázka jak přesně interpretovat pojem *„significant increase"* .

Já jsem se po pečlivém zvažování a testování rozhodl pokládat novou iteraci za lepší v případě, že se průměrný počet pravidel v poduzlech zmenší alespoň o 20%, nebo dojde ke snížení maximálního počtu nejméně o 10%. Přestože si nekladu nároky na to, že je to optimální řešení, připadá mi, že výsledky toto řešení ospravedlňují.

4.3 Stručný popis struktury implementace

Stejně jako u většiny dalších algoritmů implementovaných v Netbench (viz 4.3.1), je funkčnost algoritmu rozdělena do dvou hlavních tříd. *HyperCuts* zajišťuje úvodní předzpracování pravidel (viz dále), rozdělení na stromy s jednou (nebo žádnou) IP adresou obsahující wildcard a ty ostatní. Pro účely srovnání výhodnosti vytvoření dvou stromů místo jednoho společného se ale vytváří i tento společný strom.

Stavba stromu samotného je pak implementována ve třídě HyperCube, která probíhá tak, že se vytvoří instance této třídy, které se předají všechna pravidla pro daný strom, a která se následně rekurzivně dělí na jednotlivé uzly potomků, až je vybudována celá stromová struktura.

Instance HyperCube přijímají parametry, které určují, jak se mají dále větvit (*bucket size* a *spfac*) a jaké optimalizace vykonávat. Navíc zde existuje parametr, který nahradí výše popsané určení dimenzí, na kterých se bude provádět dělení a počet částí, algoritmem z *HiCuts*, čímž dosáhne toho, že v každém uzlu bude dělena pouze jedna dimenze a výsledek tedy bude totožný se stromem vytvořeným algoritmem *HiCuts*.

4.3.1 Netbench

Netbench[8] je Python framework pro testování a vyhodnocování klasifikačních, RE matching a IP lookup algoritmů vyvíjený výzkumnou skupinou ANT@FIT[1] na naší fakultě. Snaží se o maximální zjednodušení implementace těchto algoritmů vytvořením souboru modulů pro parsování a analýzu databází pravidel, zpracování prefixů atd.

[1] Accelerated Network Technologies at FIT

V rámci tohoto frameworku je vytvářena rozsáhlá databáze algoritmů pro zpracování paketů a má tak do budoucna ambice se stát uznávanou platformou pro jejich efektivní porovnávání. V příští verzi bude do Netbench zařazena i implementace *HyperCuts*, která je předmětem této bakalářské práce.

4.3.2 Problémy v implementaci

Paměťová náročnost

Přestože množství paměti zabrané rozhodovacími stromy se v testech zpravidla pohybovalo pod hranicí jednoho megabyte, paměť vyžadovaná pro jeho výstavbu byla často o několik řádů vyšší. Toto je způsobeno tím, že při výstavbě je nutné přiřazovat náležící pravidla nejen k listům stromu, ale i uzlům, a to jak z důvodu celkového zrychlení výstavby, tak i z nezbytnosti implementace optimalizací pro eliminaci překrytých pravidel a slučování uzlů. Tento problém je dále umocněn použitím vysokoúrovňového jazyka implementace, jazyka Python, který neumožňuje tak přesnou správu paměti.

Reprezentace pravidel

V Netbench jsou pravidla reprezentována jako objekt, který zapouzdřuje množinu prefixů pro každé pole hlavičky paketu. Tato reprezentace ale není vhodná pro algoritmy rozhodovacích stromů, takže po spuštění se pravidla musejí konvertovat na mnou vytvořený objekt RangeRule. Toto řešení sice jde proti idei Netbench, nicméně při vytváření stromů pro obsáhlejší klasifikátory se ukázalo být nezbytné pro jejich vybudování v rozumném čase.

Kapitola 5

Zhodnocení implementace HyperCuts

V této části se nejdříve zaměřím na vyhodnocení dříve popsaných možností optimalizace, následně srovnám výsledky algoritmů *HyperCuts* a *HiCuts*, vezmu v úvahu výsledky výše zmíněných optimalizací a navrhnu řešení, které by vedlo k nejrychlejší klasifikaci. Nakonec pak ještě srovnám paměťovou náročnost mnou navrhnutého řešení s implementacemi MSCA[2] a DCFL[1][12] v Netbench.

Použité klasifikátory

Aby byla možnost srovnání s jinými implementacemi, pro testování jsem použil klasifikátory použité H. Songem[11]. Tyto klasifikátory jsou syntetické, generované nástrojem Classbench[1], nicméně by měly imitovat strukturu reálných klasifikátorů.

5.1 Efekt eliminace překrývaných pravidel

Jak bylo zmíněno už v části popisující tuto optimalizaci, eliminace překrývaných pravidel je pro správné fungování *HyperCuts* i *HiCuts* téměř nezbytná. Jak je vidět z tabulky 5.1, její důležitost se odvíjí především od struktury klasifikátoru.

	s překrýváním		*bez překrývání*	
klasifikátor	*hloubka*	*velikost*	*hloubka*	*velikost*
`acl1_100`	26	2932	26	2944
`acl1_1K`	29	92068	30	114148
`acl1_5K`	29	1533824	nelze postavit	

Tabulka 5.1: V obou případech byl vybudován strom s *bucket size* = 8 a *spfac* = 2, kromě eliminace překrytí nebyla použita žádná optimalizace. Velikost je uváděna bytech. V tabulce 5.2 je příklad překrývajících se pravidel v regionu, který už nelze dál rozdělit

Pro malé počty pravidel je šance, že na zpracovávaném hyperregionu jedno pravidlo kompletně překryje jiné díky malému počtu úrovní stromu a z toho plynoucích regionů o

[1]Distributed Crossproducting of Field Labels

velkých rozsazích, relativně malá. Naopak u klasifikátorů s velkým počtem pravidel nejen že nemožností eliminovat pravidlo jakmile je překryté zbytečně zvyšujeme výšku stromu, ale také můžeme dosáhnout situace, kdy dostaneme region o velikosti jedna ve všech dimenzích a přitom máme stále počet pravidel vyšší než *bucket*.

src IP	dst IP	src port	dst port	protocol
9.152.73.194/32	239.253.176.22/32	0 - 65535	1708 - 1708	0x06/0xFF
9.152.73.194/32	239.253.176.22/32	0 - 65535	1700 - 1720	0x06/0xFF
9.152.73.194/32	239.253.176.22/32	0 - 65535	1700 - 1750	0x06/0xFF
9.152.73.194/32	239.253.176.22/32	0 - 65535	0 - 65535	0x06/0xFF
9.152.73.194/32	239.253.176.22/32	0 - 65535	0 - 65535	0x00/0x00
9.152.72. 0/23	239.253.176.22/32	0 - 65535	0 - 65535	0x06/0xFF
9.152.72. 0/23	239.253.176.22/32	0 - 65535	0 - 65535	0x00/0x00
9.152.72. 0/23	192. 0. 0. 0/ 2	0 - 65535	0 - 65535	0x06/0xFF
9.152.72. 0/23	0. 0. 0. 0/ 0	0 - 65535	0 - 65535	0x06/0xFF
9. 0. 0. 0/ 8	0. 0. 0. 0/ 0	0 - 65535	0 - 65535	0x06/0xFF
0. 0. 0. 0/ 0	0. 0. 0. 0/ 0	0 - 65535	0 - 65535	0x06/0xFF
0. 0. 0. 0/ 0	0. 0. 0. 0/ 0	0 - 65535	0 - 65535	0x00/0x00

Tabulka 5.2: Neseparovatelná pravidla číslo 1233, 2620, 2726, 3223, 3742, 3975, 4205, 4371, 4373, 4374, 4402, 4408 a 4414 z klasifikátoru `acl1_5K`.

Je samozřejmě možné namítnout, že u nesyntetických klasifikátorů by se takové množství překrývajících se pravidel nevyskytlo, nicméně, jak poznamenávají[9], pro reálné firewall klasifikátory je běžných až 7 takto se překrývajících pravidel. Přitom je nutné ještě vzít v úvahu možnost chyby administrátora, kdy víckrát vloží stejné pravidlo do klasifikátoru.

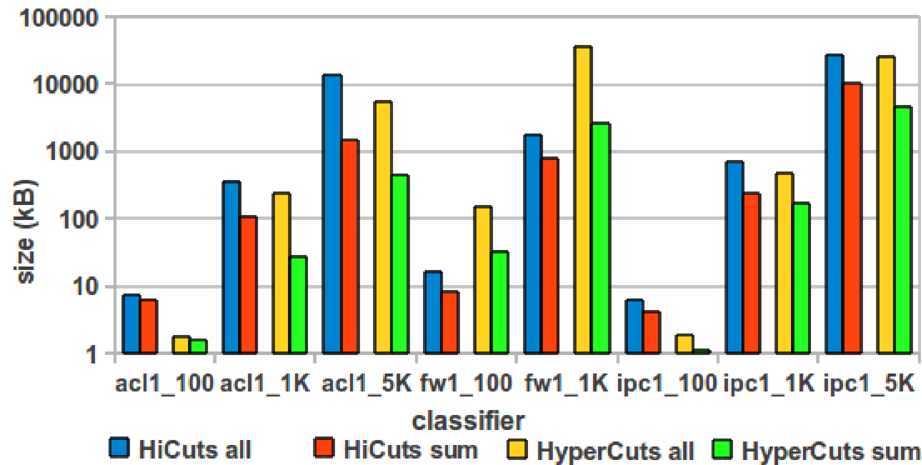
5.2 Výhodnost rozdělení klasifikátoru do dvou stromů

Využití zvláštního stromu pro pravidla s oběmi IP adresami obsahujícími wildcards (dále budu někdy označovat *wild tree*) a druhého stromu pro ostatní pravidla (*base tree*), bylo navrženo tvůrci *HyperCuts* jako regulérní součást algoritmu kvůli ušetření paměti u rozsáhlých klasifikátorů. Proto jsem se snažil zjistit jak moc je to výhodné a zdalipak by toto řešení bylo výhodné využít i pro další algoritmy rozhodovacích stromů, konkrétně pak *HiCuts*.

Jak je vidět z grafu 5.1, ve zkoumaných situacích, kromě klasifikátorů `acl1_100` a `ipc1_100`, bylo ve všech případech využití paměti při použití dvou stromů nejméně poloviční, v některých případech však i méně než desetinové. Dále si můžeme všimnout, že ve všech případech je u *HyperCuts* paměťová úspora vyšší, to souvisí s tím, že *HiCuts* a *HyperCuts* interpretují *spfac* parametr jiným způsobem, což způsobuje, že při stejném parametru dělí *HiCuts* regiony zpravidla agresivněji.

To je vidět i na grafu 5.2, který ukazuje, že hloubka společného stromu *HiCuts* je vždy stejná nebo menší než u *HyperCuts*, stejně jako u součtu hloubek *base* a *wild tree*, kde výjimku tvoří klasifikátor `ipc1_100`, v tomto případě je rozdíl způsobený extrémně nízkým počtem pravidel v kombinaci s možností provádět dělení na více dimenzích v jednom uzlu.

Čeho je také třeba si všimnout, je fakt, že vyhledání paketu bude ve všech případech rychlejši u kombinovaných stromů. I když je v případě dvou klasifikátorů součet výšek stromů nižší než výška společného stromu, je nutné vzít v úvahu ještě *bucket size*, která se



Graf 5.1: Porovnání celkové spotřeby paměti. Přípona sum znamená, že je uveden součet pro dva stromy, all implikuje klasifikátor v jednom stromě.

bude u dvou stromů započítávat dvakrát.

Z toho plyne, že se dělení klasifikátoru na dva stromy vyplatí v případech, kdy je nějakým způsobem omezená využitelná paměť – tedy zpravidla tehdy, když by se celý společný strom nevešel do (určité úrovně) vyrovnávací paměti procesoru.

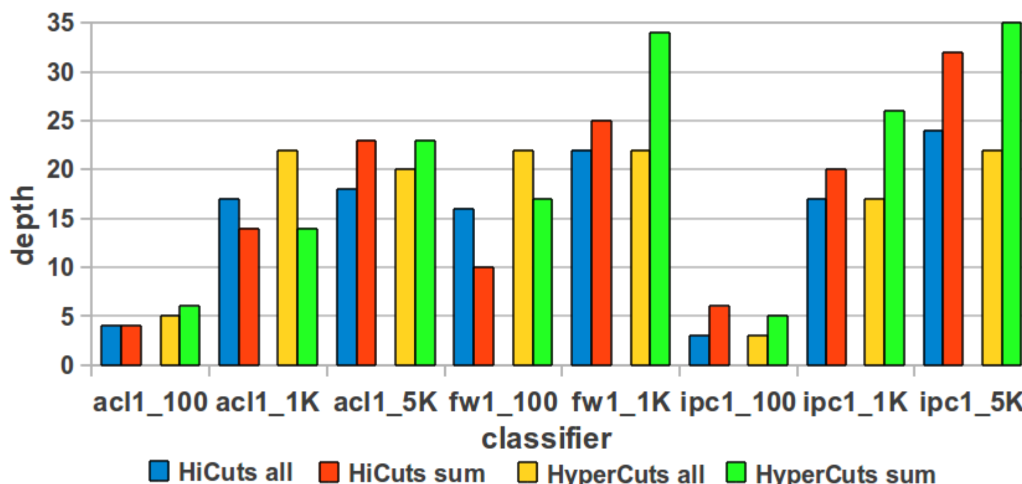
Nicméně, protože pro klasifikátory zabírající velmi málo paměti je nutný jen malý počet přístupů do paměti i při procházení dvou stromů a rychlost je tedy méně limitujícím faktorem, budu se nadále zabývat pouze možností, kdy jsou jak pro *HyperCuts*, tak pro *HiCuts* vytvářeny dva oddělené stromy.

5.3 Přínos předávání společných pravidel mateřskému uzlu

U této optimalizace, ve formě navržené autory *HyperCuts*(3.5.3), se na první pohled může zdát, že přestože ušetří paměť, nutnost kontroly u každého uzlu (nejen listů), zda s ním nejsou asociována nějaká pravidla, natolik zvýší počet přístupů do paměti, že úspora paměti toto zpomalení nevynahradí. Lze to však obejít tak, že se do hlavičky uzlu (která obsahuje počet „cutů“ na jednotlivých dimenzích) přidá bit značící, zda je s uzlem asociováno nějaké pravidlo a tím se tyto zbytečné přístupy do paměti eliminují[9], což znamená, že je to možnost jak ušetřit paměť „zadarmo“.

Jak je vidět na grafu 5.3, výsledná úspora paměti je výrazná u klasifikátorů, kde je velké množství obecných a překrývajících se pravidel, u `acl1_10K` je naproti tomu naprosto zanedbatelná. Přestože byla tato optimalizace navržena pro *HyperCuts*, je vidět, že pro všechny klasifikátory je procentuální úspora paměti *HiCuts* srovnatelná, pro `fw1_5K` dokonce výrazně vyšší (to je způsobeno velkým množstvím překrývajících se pravidel v kombinaci s agresivnějším dělením regionů a tím pádem větší úspory duplikovaných pravidel).

V dalších testech se bude automaticky předpokládat, že je tato optimalizace povolena.



Graf 5.2: Porovnání maximálních hloubek listových uzlů. Přípona „sum“ znamená, že je v grafu zobrazen součet obou hlubek pro daný algoritmus a „all“ že je celý klasifikátor v jednom stromě.

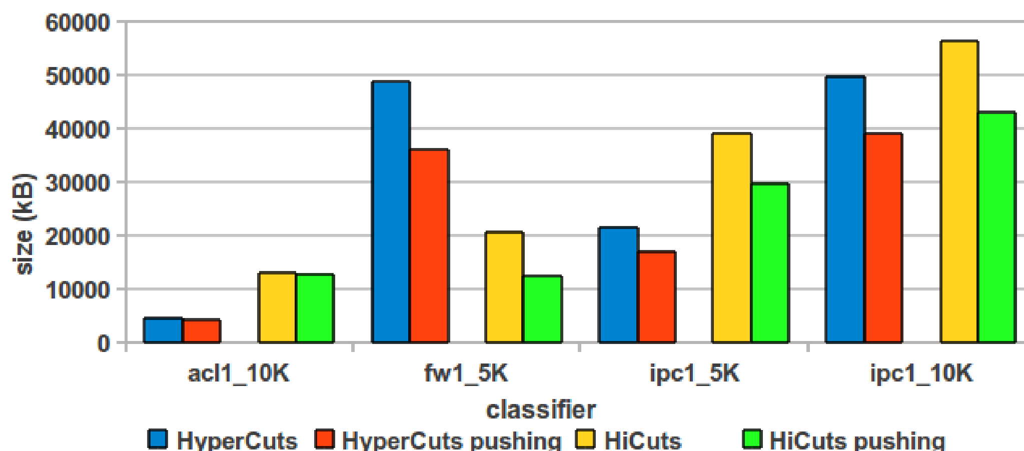
5.4 Vliv zmenšení regionů a jejich spojování na výkonnost algoritmu

Přestože jsou to optimalizace dvě, vzhledem k tomu, že obě vyžadují, aby v uzlech (které jsou touto optimalizací ovlivněny) byly uloženy nové rozměry regionu, nemá smysl je hodnotit odděleně. To vede k tomu, že při vyhledávání jsou nutné přístupy do paměti navíc, ale na druhou stranu zmenšování regionů podle obsažených pravidel může vést ke zvýšení efektivity dělení regionů (nižší hloubce stromu) a spojování uzlů zase k redukci potřebné paměti.

Jak je vidět z grafu 5.5, u `acl1_10K` klasifkátoru, kde jsou pravidla nejspecifičtější, vedly tyto optimalizace k naprosto zásadní úspoře paměti, u ostatních klasifikátorů vyjma `fw1_5K` je úspora paměti přibližně třípětinová. Na první pohled může být překvapivé, že u `fw1_5K`, kde bychom čekali velké množství obecných pravidel, je užitek ze slučování tak malý. Nicméně, vrátíme-li se ke grafu 5.3, uvidíme, že tento klasifikátor nejvíce profitoval z optimalizace předáváním pravidel rodičovskému uzlu, a pochopíme, že obě tyto optimalizace řeší různým způsobem stejný problém redundance pravidel v uzlech. Takže, byla-li jedna optimalizace velmi efektivní, u druhé už nebude takový prostor ke zlepšení.

V grafu 5.4 je vidět především efekt oříznutí regionů podle vnitřních pravidel. U ACL klasifikátorů je redukce hloubky výrazná, zatímco u ostatních klasifikátorů, kde pravidla pokrývají velkou část regionů (a ořezávání regionu se tedy neprovádí) je změna zanedbatelná. U *HyperCuts* dokonce v jednom případě hloubka vzrostla. Pravděpodobně se v nějakém místě po provedení optimalizace změnily metriky výběru dimenzí pro provedení dělení (v konečném výsledku k horšímu).

Z těchto výsledků je patrné, že, vzhledem k nárustu přístupů do paměti při vyhledávání(v krajních případech až na dvojnásobek), je přínos a užitečnost této optimalizace krajně diskutabilní. Závěr je tedy podobný jako u rozhodování se o použití dvou stromů – využití je vhodné jen v případech, kdy je paměť omezujícím faktorem. Vzhledem k mírám úspory paměti je v případě výběru z těchto dvou možností zpravidla výhodnější použít dva stromy.



Graf 5.3: U stromů s přízviskem *pushing* jsou pravidla společná pro všechny potomky vytlačena vzhůru (*pushed up* do rodičovského uzlu.

5.5 Srovnání HyperCuts a HiCuts

Z předchozích výsledků bylo zřejmé, že přestože se výsledky lišily u jednotlivých klasifikátorů, všeobecně se dá říci, že *HyperCuts* spotřebovalo výrazně méně paměti, zatímco *HiCuts* dosahovalo o něco menších součtů výšek stromů. Nicméně, abychom mohli analyzovat zdroje rozdílů v závislosti na klasifikátorech, pouze součty hodnot obou stromů mohou být zavádějící. Proto si prohlédňeme výsledky jednotlivých stromů odděleně.

Jak je vidno z grafu 5.6, pro všechny klasifikátory platí, že pro *base tree* dosahuje *HiCuts* výrazného snížení výšky stromu, zatímco *HyperCuts* dosahuje lepších výsledků u *wild tree*. To lze vysvětlit tak, že agresivní dělení stromů je výhodné jen pro specifická pravidla, zatímco u obecných pravidel se více projeví větší variabilita dělení u *HyperCuts*.

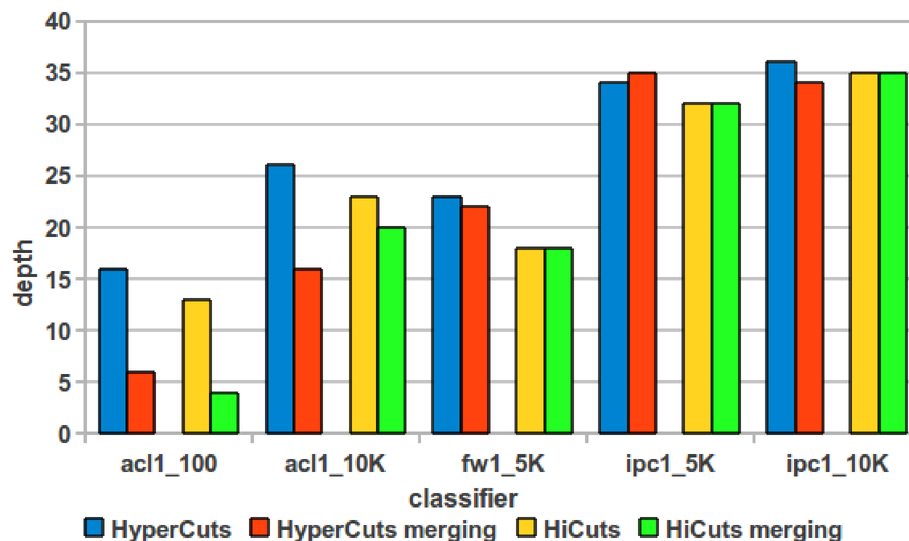
V grafu srovnávajícím spotřebu paměti jednotlivých stromů (graf 5.7) je však situace složitější. Pro *base tree* klasifikátoru `acl1_10K`, kde dosáhlo *HiCuts* poloviční výšky oproti *HyperCuts*, je spotřeba paměti naopak trojnásobná. U `fw1_5K` se zase chová atypicky pro *wild tree HyperCuts* – zatímco ve všech ostatních připadech oproti *HiCuts* dosáhl lepších výsledků jak v případě hloubky, tak v případě velkosti paměti, zde je spotřeba paměti dokonce šestkrát vyšší.

Pravděpodobné řešení této otázky leží v nepoužití (z výše uvedených důvodů) optimalizací spojování a ořezávání uzlů.

U `acl_10K` se agresivní dělení regionů (umožněné relativně prázdnými regiony, viz 3.5.1) projeví velkou mírou duplikace pravidel, zatímco u `fw1_5K` se s největší pravděpodobností projevila výše zmíněná nevýhoda mého řešení, která dělila region na maximální (ale stále omezený podmínkou navrhnutou autory *HyperCuts*) počet částí, když je malý počet možností způsobů distribuce cutů (viz 4.2).

5.6 Zrychlení kombinací HiCuts a HyperCuts

Protože při dělení klasifikátoru do dvou stromů pravidelně HiCuts dosahuje nižšího *base tree* a *HyperCuts* zase *wild tree*, lze získat to lepší s obou dvou jejich kombinací na jednom klasifikátoru. Srovnání tohoto řešení s původními *HyperCuts* a *Hicuts* potom bude vypadat jako na diagramech 5.8 a 5.9.



Graf 5.4: Porovnání účinků optimalizací spojování a ořezu uzlů na výšku stromů *HyperCuts* a *HiCuts*. U stromů s popiskem merging je proveden optimalizace. Použité parametry byly *spfac*=4 a *bucket size*=10

Jak lze vidět, přestože se výška stromů této kombinace snížila, spotřebovaná paměť řešení zůstala taková, že nikdy není nejhorší a v jednom případě je dokonce nejlepší.

5.7 Srovnání s MSCA a DCFL

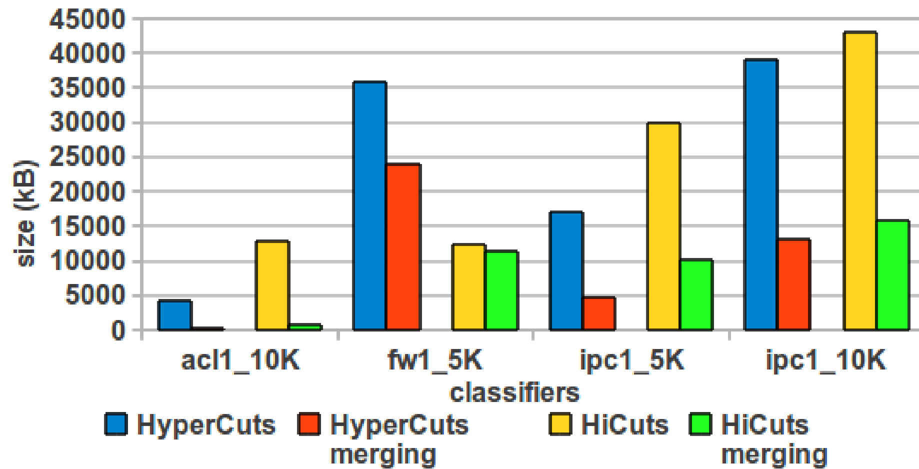
Pro srovnání jsem vybral klasifikátory o tisíci pravidlech, u větších už bylo při zpracovávání, zvlášť u MSCA, limitujícím faktorem množství paměti testovaného stroje.

Jak je vidět v tabulce 5.3, s přehledem nejnižši spotřeby paměti dosahuje algoritmus DCFL, kombinace *HyperCuts* a *HiCuts*, kromě tradičně nejhoršího klasifikátoru `fw1`, dosáhla slušné úrovně přibližně 350 bytů na pravidlo(to je podobný výsledek jako dosáhl Song, vezmeme-li v úvahu nižší *bucket size* a absenci dopředného přesouvání pravidel do rodičovského uzlu[10]). Naproti tomu, MSCA dopadlo velmi špatně, došlo zde k situaci zvané *„memory blowup“*.

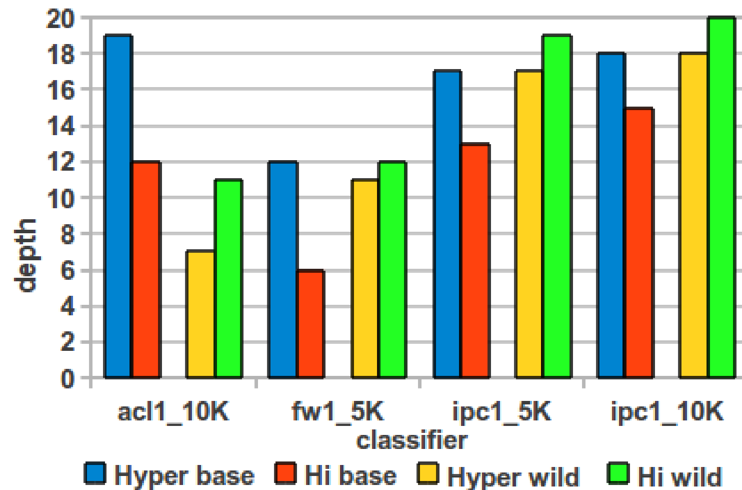
algorithm	acl1_1K	fw1_1K	ipc1_1K
MSCA	8917.6	85014	291030
DCFL	10.8	15	12.1
CombinedCuts	330.8	2616.7	320.5

Tabulka 5.3: Spotřeba paměti v kB u MSCA, DCFL a kombinace HyperCuts a Hicuts

Rychlost je obtížnější posoudit, protože MSCA a DCFL mají, jako hardwarová řešení, výhodu v možnosti přistupovat k většímu množství paměti naráz (například DCFL v tomto případě po 72 bitech). Pro DCFL je nicméně maximální nutný počet přístupů do paměti k nalezení pravidla 20 (tzn. 180 bytů pro klasifikátor `ipc1_1K`). Kombinace *HiCuts* a *HyperCuts* v nejhorším případě při vyhledávání paketu přečte 380 bytů $((10+10)*13byt$ průchod listy, hloubka 30 – krát 4 byty za hlavičku uzlu) pro klasifikátor `fw1_1K`. U MSCA není



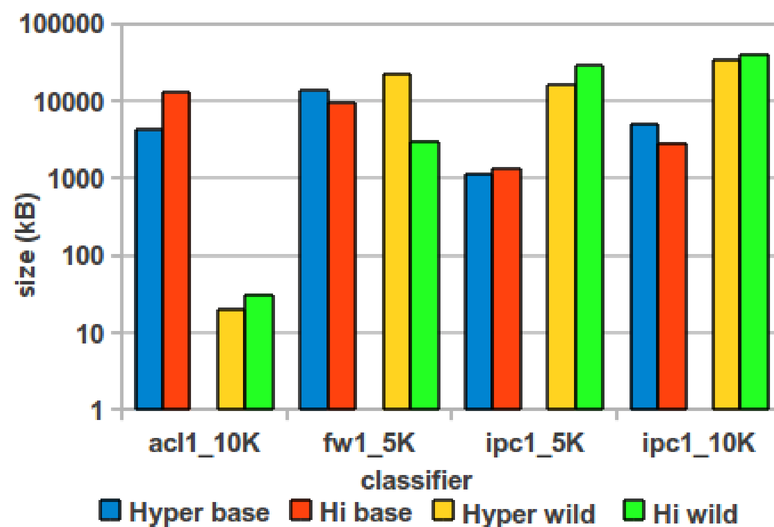
Graf 5.5: Merge Velikost



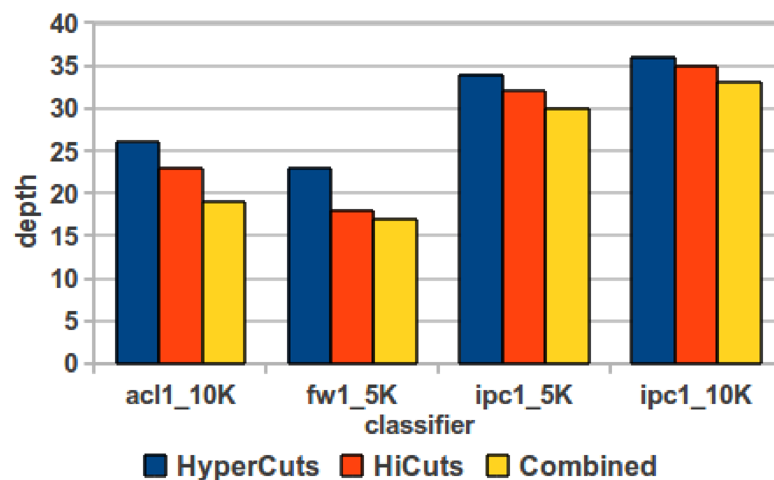
Graf 5.6: Porovnání celkové výšky HyperCuts a HiCuts pro jednotlivé stromy

možné určit počet přístupů do paměti – základní počet je sice čtyři, nicméně vzhledem k tomu, že MSCA vrací všechna příslušná pravidla (nejen to s nevyšší prioritou) a počet překrytých pravidel je u těchto klasifikátorů přinejmenším 12 (viz tabulku5.2), pravděpodobně však více. Lze tedy předpokládat, že výsledek bude horší než u DCFL.

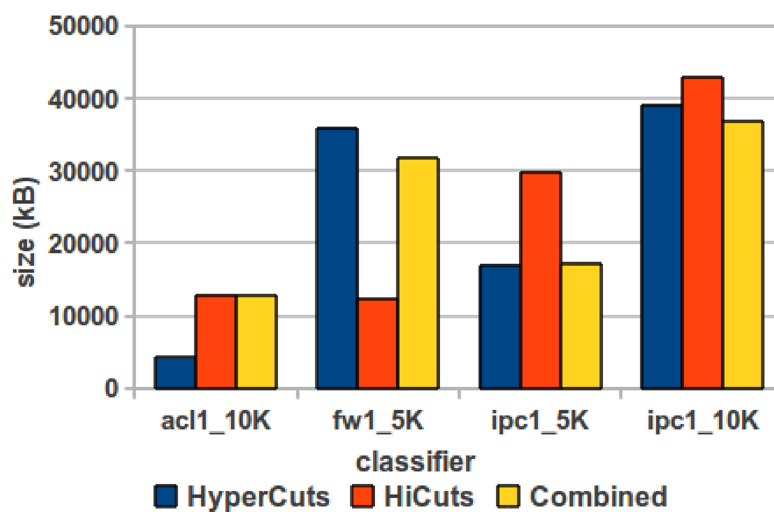
Přestože se v tomto porovnání ukázalo, že ve srovnání s algoritmy určenými pro zpracování v hardware není *HyperCuts* (respektive *HyperCuts* v kombinaci s *HiCuts*) příliš efektivní, stále má své uplatnění jako řešení, které může běžet na libovolném PC bez investice do specializovaného hardware.



Graf 5.7: Porovnání celkové spotřeby paměti HyperCuts a HiCuts pro jednotlivé stromy



Graf 5.8: Porovnání rychlosti HyperCuts, HiCuts a kombinace HyperCuts a Hicuts.



Graf 5.9: Porovnání spotřeby paměti HyperCuts, HiCuts a kombinace HyperCuts a Hicuts.

Kapitola 6

Závěr

V rámci této bakalářské práce jsem analyzoval a implementoval algoritmy *HyperCuts* a *HiCuts* a optimalizace navržené pro klasifkační algoritmy rozhodovacích stromů. Ještě předtím jsem se však podrobně seznámil se základními principy klasifikace paketů a také s jinými přístupy ke klasifikaci, než je použití rozhodovacích stromů. Některé z nich jsem popsal v kapitole 3.

I přes velmi nejasný popis *HyperCuts* jeho autory se mi podařilo algoritmus implementovat a vytvořit tak jedinou veřejnou verzi implementující všechny původně navržené optimalizace.

Dále jsem zkoumal výkonnost algoritmu na různých klasifikátorech, srovnával jeho výkonnost s *HiCuts* a analyzoval přínos jednotlivých optimalizací.

Díky výsledkům této analýzy se mi podařilo najít způsob, jak dosáhnout snížení počtu přístupů do paměti na všech zkoumaných klasifikátorech použitím algoritmu *HyperCuts* na jedné části pravidel a *HiCuts* na zbylých.

Přestože jsem s výsledky, kterých jsem v této práci dosáhl spokojen, napadá mě ještě jeden způsob jak by bylo možné implementační část této práce rozšířit.

Tímto rozšířením je vypracování metody analýzy klasifikátoru tak, aby ho bylo možné rozdělovat do dvou stromů efektivněji, než jen pomocí jednoduchého *„jsou obě IP adresy definovány jako rozsah?“*, podobně jako Netebench umožňuje analyzovat prefixy obsažené v klasifiátoru. To by bylo možné ještě kombinovat s doporučením vhodných parametrů *spfac* a *bucket size* tak, aby bylo dosaženo co nejlepšího výsledku, v současné podobě jsou totiž tyto parametry závislé čistě na uvážení uživatele. Nicméně toto bude vyžadovat ještě velké množství testování a studia odborné literatury a je to, alespoň prozatím, zcela mimo mé schopnosti.

Literatura

[1] David E. Taylor, J. S. T.: ClassBench: A Packet Classification Benchmark. [online], 2011-18-01 [cit. 2012-15-05].
URL http://www.arl.wustl.edu/classbench/index.htm

[2] Dharmapurikar, S.; Song, H.; Turner, J.; aj.: Fast packet classification using bloom filters. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, ANCS '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-580-0, s. 61–70, doi:10.1145/1185347.1185356.
URL http://doi.acm.org/10.1145/1185347.1185356

[3] Gupta, P.: *Algorithms for Routing Lookups and Packet Classification*. Dizertační práce, Stanford University, 2000.

[4] Gupta, P.; Mckeown, N.: Packet Classification using Hierarchical Intelligent Cuttings. *Work*, 1999: s. 34–41.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.7809

[5] Lakshman, T. V.; Stiliadis, D.: High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *In ACM SIGCOMM*, 1998, s. 203–214.

[6] Mašíček, V.: *HyperCuts pro filtrování Linuxových paketů*. Diplomová práce, Univerzita Karlova v Praze, 2006.

[7] Puš, V.: *Klasifikace paketů s využitím technologie FPGA*. Diplomová práce, Vysoké učení technické v Brně, 2008.

[8] Puš, V.; Tobola, J.; Kosař, V.; aj.: Netbench Framework. [online], 2012-10-04 [cit. 2012-15-05].
URL http://merlin.fit.vutbr.cz/ant/netbench/index.html

[9] Singh, S.; Baboescu, F.; Varghese, G.; aj.: Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, New York, NY, USA: ACM, 2003, ISBN 1-58113-735-4, s. 213–224, doi:10.1145/863955.863980.
URL http://doi.acm.org/10.1145/863955.863980

[10] Song, H.: *Design and Evaluation of Packet Classification Systems*. Dizertační práce, Washington University in Saint Louis, 2006.

[11] Song, H.: Evaluation of Packet Classification Algorithms. [online], 2007-13-02 [cit. 2012-15-05].
URL http://www.arl.wustl.edu/~hs1/PClassEval.html#3._Filter_Sets

[12] Taylor, D. E.; Turner, J. S.: Scalable Packet Classification Using Distributed Crossproducting of Field Labels. 2004.

Příloha A

Obsah CD

- **netbench/** - Netbench klasifikační framework
- **logs/** - výstupy z testování implementace s různými parametry
- **rules/** - použité klasifikátory
- **latex/** - zdrojové kódy tohoto textu
- **BP.pdf** - elektronická verze této práce
- **REAMDE** - info ke spuštění HyperCuts
- **test.py** - spouštěč HyperCuts
- **setup.sh** - přidá Netbench do PATH