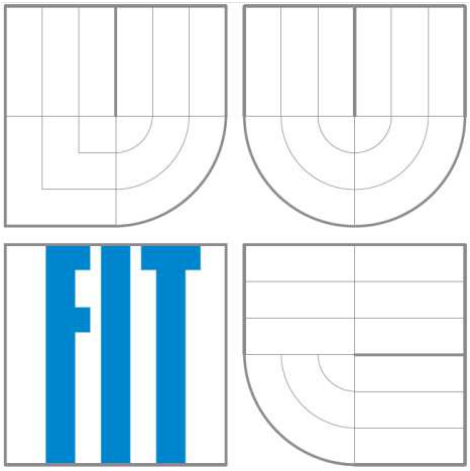

Dokumentace k projektu do předmětu PDS

# Longest-Prefix Match

20. dubna 2013

Autor: Radim Loskot, xlosko01@stud.fit.vutbr.cz
Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

# 1 Úvod a zadání problému

Tento dokument analyzuje vyhledávací algoritmus Longest-Prefix Match a jeho využití ve vyhledávání v oblasti počítačových sítí. Dokument definuje datovou strukturu Trie a zkoumá její vlastnosti, které se dají využít pro vyhledávání klíčů s nejlepší shodou k řetězci hledaného klíče. Dále popisuje implementaci vyhledávání pomocí zmíněného algoritmu pro určení autonomních systému na základě IP adresy. Jelikož vyhledávání je v sítích během směrování velmi frekventovaná operace, byl brán během implementace zřetel na optimalizaci algoritmu, která je rozebrána v závěru dokumentu.

# 2 Analýza problému

## 2.1 Zadání problému

Cílem projektu je nastudovat problematiku Longest-Prefix Match vyhledávání a vytvořit program, který umožní zpracovávat IP adresy ze standardního vstupu a nalézt odpovídající čísla autonomních systémů. Mapování IP adres na korespondující čísla autonomních systému bude definované v souboru, jehož název bude předán programu během spouštění přepínačem `-i`. Postupně nacházená čísla autonomních systémů budou vypisována na standardní výstup.

## 2.2 Longest-Prefix Match a datová struktura Trie

Pojem Longest-Prefix Match odpovídá algoritmu, který se snaží vybírat nejvíce odpovídající záznam k hledané položce. Vyhledání probíhá pomocí klíče, nejčastěji řetězcového. Od toho se odvíjí i název algoritmu, neboť se snažíme položku s odpovídajícím klíčem, nebo položku s nejdelším shodným prefixem klíče. Vyhledávání podle klíče tedy pokaždé nabízí maximálně pouze jednu výslednou položku a to s nejdelším shodným prefixem k hledanému klíči. Z důvodu optimalizace vyhledávání je běžné, že prohledávání probíhá často ve specializovaných datových strukturách, kde jsou jednotlivé položky uchovávány. Jmenovitě se může jednat např. o stromové struktury Trie či Radix-Trie.

Pro implementaci daného vyhledávání použijeme stromovou strukturu Trie. Jedná se o strukturu určenou pro hledání řetězců, kde není zapotřebí testovat každý znak klíče s každým uloženým záznamem (uzlem). Základní myšlenka vyhledávání řetězců v Trie je v procházení znaků hledaného řetězce a v postupném navštívení odpovídajících uzlů ve stromové struktuře – uzlů, které mají shodný prefix. Neexistuje-li daný uzel, přes který bychom mohli projít, hledaný řetězec neexistuje. Maximální počet synovských uzlů, které mohou vést z uzlů, je $k$, kde $k$ je počet symbolů abecedy tvořící řetězce. [2] V případě vyhledávání IP adres budou řetězce tvořeny bity, tudíž se bude jednat o binární Trie. Časová složitost vyhledávání klíče je lineární a odpovídá délce řetězce hledaného klíče.

```
Nastav kořenový uzel jako aktuální uzel
Nastav nalezenou hodnotu ke klíči na NULL
for each ( znak v hledaném klíči ) {
    if exist ( syn aktuálního uzlu odpovídající aktuálnímu znaku klíče ) {
        Nastav uzel tohoto syna jako aktuální uzel
    } else {
        break;            // Konec vyhledávání, nelze se již dále zanořit
    }

    if ( aktuální uzel má nastavenou hodnotu ) {
        Nastav hodnotu aktuálního uzlu jako nalezenou hodnotu ke klíči
    }
}
```

Algoritmus 2.1: Longest-Prefix Match vyhledávání ve struktuře Trie [1]

# 3 Implementace

Implementace aplikace byla provedena v jazyce C++ s využitím standardních knihoven. Aplikace byla optimalizována, zejména pro IO operace, blíže bude popsáno v kapitole 4.

Hlavní modul aplikace definovaný v `longest_prefix.cpp` zahrnuje zpracování argumentů z příkazové řádky, načítání mapovacího souboru s autonomními systémy, vkládání těchto systému do struktur Trie, podle verze IP adresy a masky sítě. Po úspěšném naplnění stromu přejde ke čtení IP adres na vstupu a vyhledávání ve stromě jim odpovídajících autonomních systémů.

## 3.1 Třídy aplikace

Struktura Trie je obecně definována jako třída `AddrTrieBase` v souboru `AddrTrieBase.h` a rozšířena o šablonovou definici ve formě třídy `AddrTrie` v souboru `AddrTrie.h`. Šablonová reprezentace Trie implementuje typově specifické metody pro rodinu adres, které jsou do stromu vkládány. Třídy struktury Trie byly napsány obecně a lze je využít jak pro vyhledávání IPv4 adres, tak i pro IPv6 adresy.

Oběma stromovým třídám je zapotřebí předat typovou sémantiku pomocí třídy `FamilyInfo`, která je definována v souboru `AddrFamilies.h`. Stromová třída `AddrTrieBase` přijímá objekt třídy `FamilyInfo` jako parametr v konstruktoru, zatímco třída `AddrTrie` přijímá `FamilyInfo` ve formě šablony, kde proběhne tvorba objektu `FamilyInfo` až v rámci konstrukce objektu `AddrTrie`. Třída `FamilyInfo` uchovává informace o adrese a to její velikost v bytech, rodinu adresy, přesnou velikost adresy v bitech a převodovou funkci pro převod z řetězcové reprezentace adresy na datovou (číselnou) reprezentaci.

Poslední a pro strom nejdůležitější třídou je třída pro uzel stromu `TrieNode`. Třída obsahuje pouze ukazatele na uzel levého a pravého syna a uchovávanou hodnotu, jenž může, ale nemusí uzel obsahovat.

# 4 Profilování a optimalizace implementace

Jelikož aplikace provádí některé operace velmi často, zejména vkládání a hledání ve stromové struktuře Trie, bylo výhodné tyto stěžejní operace (metody) optimalizovat. Z důvodu optimalizace byly často využívané metody definovány jako `inline`.

Během profilování bylo zjištěno, že v rámci parsování vstupního souboru s mapováním autonomních systémů a práce s objekty řetězců docházelo ke značnému zpomalení. Zpomalení vznikalo z důvodu časté realokace paměti pro řetězce a používání knihovních metod pro práci s řetězci. Veškerá práce s objekty řetězců byla zaměněna na práci s řetězci znaků a pro parsování byl vytvořen jednoduchý konečný automat.

Volání knihovní funkce `atoi()` během převodu čísla autonomního systému při parsování mapovacího souboru a funkce `sprintf()` pro převod nalezeného čísla ve stromové struktuře bylo vynecháno. Ve stromové struktuře jsou nyní uloženy přímo znakové řetězce čísel autonomních systémů. Ačkoliv tato optimalizace klade vyšší nároky na paměť, došlo k velkému zrychlení v celé vyhledávací iteraci, jelikož není čísla již potřeba převádět na řetězce znaků. Řetězce autonomních systému jsou nyní alokovány duplicitně, tudíž lze aplikaci dále optimalizovat a redukovat její paměťovou náročnost.

Nejpodstatnější optimalizace byla dosažena lepší prací se vstupem a výstupem a přímým volání jádra pomocí systémových funkcí `write()` a `read()`. Aby došlo ke zmírnění negativního efektu přepnutí kontextu během volání jádra, probíhá načítání vstupu, případně zápis výstupu pro dostatečně velký buffer o velikosti 64 KiB. V aplikaci jsou celkem dva buffery – pro vstup ze souboru `block_rbuffer` a pro výstup do souboru `block_wbuffer`.

## 4.1 Dosažené výsledky

Aplikace byla testována na poskytnutém referenčním stroji, na školních serverech merlin a eva. Z důvodu velmi častého a dlouho trvajícímu blokování aplikace během vstupu a výstupu na referečním stroji, zřejmě z důvodu sdílené složky mezi dvěma systémy, bylo experimentování provedeno na školním stroji merlin.

Během implementace byl brán ohled na optimalizaci kódu. Rychlost prvotního kódu dosahovala téměř 1 milionu zpracovaných IP adres za sekundu. Tento kód se ovšem podařilo zoptimalizovat ještě jednou takovým poměrem. Výsledná aplikace zpracuje 20 mil. adres za 10 sekund, rychlost je 2 000 000 adres/s, z toho 9,6 s stráví aplikace v uživatelském režimu a 0,4 s v režimu jádra.

Vstupní soubor s mapováním autonomních systému není nijak předzpracováván a jeho načítání trvá po optimalizaci pro vyhledávání 0,3 s. Před provedením optimalizací – uchováváním ve stromu čísel autonomních systému a voláním knihovních funkcí `atoi()` a `sprintf()` trvalo načítání pouhých 0,1 s. Danou optimalizací ovšem došlo k redukci času při vyhledávání o 3 s na 20 milionů adres, tudíž celkové zrychlení převážilo zpomalení procesu konstrukce stromu.

Odstraněním volání knihovních funkcí pro vstup a výstup a přímým voláním jádra byly ušetřeny celkem 4 s na 20 milionů adres. Zlepšením parsování mapovacího souboru a vyhnutí se práci s objekty řetězců se ušetřila téměř 1 s výpočetního času. Zbylé úspory na čase byly dosaženy v efektivnější práci s řetězci pro vstup/výstup.

### 4.2 Omezení z důvodu optimalizací

V důsledku implementování optimalizací rychlosti aplikace byla odstraněna kontrola správnosti vstupních dat. Aplikace neprovádí testy na validitu adres, nesnaží se je ani zotavovat a dokonce ani neinformuje o této skutečnosti uživatele na chybový výstup. Chybný vstup se vždy přečte a je naivně předpokládán jako správný vstup, tudíž proběhne buď vložení do stromu špatné adresy, nebo vyhledávání se špatnou adresou. Při detekci nového řádku by však měl vstup být čten opět očekávaně a nemělo by dojít k neočekáváným stavům běhu programu.

Vstup v aplikaci není čten po řádcích, ale bufferovaně s bufferem, který pojme až několik tisíc IP adres, to může vést k dalšímu neobvyklému chování programu. Při bufferování nedochází totiž k okamžitému převodu a vyhledání odpovídající adresy při načtení nového řádku, ale až poté, co je buffer zaplněn. Po zaplnění bufferu a navrácení jeho obsahu jádrem je vyhledávání spuštěno nárazově nad celou sadou adres, obsažených v bufferu.

## 5 Překlad a spuštění programu

Pro překlad aplikace je zapotřebí mít nainstalovaný GNU make. GNU makefile pro překlad aplikace je umístěn v souboru `Makefile.am`. Pokud je GNU make asociovaný s příkazem `make`, lze překlad provést následujícím způsobem:

```
$ make -f Makefile.am
```

Pro zjednodušení překladu byl vytvořen univerzální makefile, který dokáže zpracovat jak GNU make, tak i FreeBSD make. V tomto univerzálním makefilu je volán skript `run_make.sh`, jenž v závislosti na operačním systému spouští GNU make. Předpokládá se, že v distribucích Linuxu je GNU make asociovaný přímo s příkazem `make` a na FreeBSD s příkazem `gmake`. Ve výsledku je tedy překladu dosaženo pouhým příkazem `make`.

Pro spuštění programu `lpm` je vždy zapotřebí zadat soubor s autonomními systémy přepínačem `-i`. Pokud se jmenuje soubor s autonomními systémy `asns.txt` a soubor s vyhledávanými IP adresami `ip.txt`, lze spustit program pouhým příkazem `make run`.

## 6 Závěr

Během řešení projektu jsem se snažil splnit veškeré požadavky plynoucí ze specifikace projektu. Při vypracování se nevyskytlo příliš mnoho závažnějších problémů, které by bylo nutné řešit. Aplikace byla vyvíjena pro platformu Linux, nicméně byla testována i na platformě FreeBSD.

Byl implementován algoritmus Longest-Prefix Match, který pracuje nad strukturou Trie, ve které vyhledává odpovídající čísla autonomních systému k předaným IPv4 a IPv6 adresám na standardním výstupu. Aplikace byla optimalizována a v současné době je schopna na testované stanici merlin zpracovat průměrně 2 miliony IP adres za sekundu.

## Literatura


[1] Knuth, Donald (1997). "6.3: Digital Searching". *The Art of Computer Programming Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley. p. 492. ISBN 0-201-89685-0.

[2] *Lecture 25: Tries and Digital Search Trees* [online]. [cit. 2013-04-20]. Dostupný z WWW: <http://www.cs.umd.edu/users/meesh/420/Notes/MountNotes/lecture25-tries.pdf>