VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

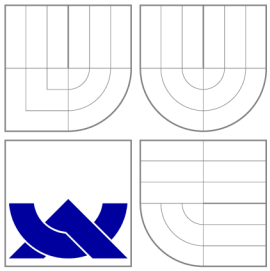
VIZUALIZACE ALGORITMŮ PRO VYHLEDÁNÍ NEJDELŠÍHO SHODNÉHO PREFIXU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

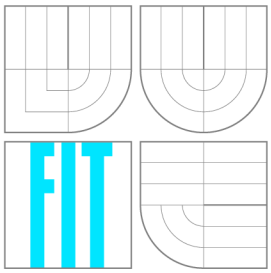
AUTOR PRÁCE
AUTHOR

JIŘÍ FOMICZEW

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VIZUALIZACE ALGORITMŮ PRO VYHLEDÁNÍ NEJDELŠÍHO SHODNÉHO PREFIXU

VISUALIZATION OF LONGEST PREFIX MATCH ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ FOMICZEW

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL KOVÁČIK

BRNO 2013

Abstrakt

Tato práce se zabývá návrhem a implementací programu pro vizualizaci algoritmů pro vyhledání nejdelšího shodného prefixu (LPM), což je jedna z nejdůležitějších operací při klasifikaci a směrování paketů v sítích TCP/IP. Je popsána jak základní teorie, tak vybrané algoritmy – Trie, Tree Bitmap a CPE. Dále je pak popsán návrh a implementace programu pro vizualizaci vyhledávacího procesu těchto algoritmů s důrazem na možné použití pro pedagogické účely. Nakonec jsou probrány možnosti budoucího vývoje a rozšíření programu.

Abstract

This thesis describes the design and implementation of program for vizualization of algorithms for longest prefix match (LPM), which is one of the most important tasks for packet classification and routing in TCP/IP networks. It describes necessary theory and details about selected algorithms – Trie, Tree Bitmap and CPE. Furthermore, it describes the design and implementation of program for vizualization of the search process of these algorithms with emphasis on the potential use for educational purposes. Finally, it describes the possibilities for future development and expansion of the program.

Klíčová slova

LPM, nejdelší shodný prefix, prefix, IP, Trie, Tree Bitmap, CPE, vizualizace

Keywords

LPM, longest prefix match, prefix, IP, Trie, Tree Bitmap, CPE, visualization

Citace

Jiří Fomiczew: Vizualizace algoritmů pro vyhledání nejdelšího shodného prefixu, bakalářská práce, Brno, FIT VUT v Brně, 2013

Vizualizace algoritmů pro vyhledání nejdelšího shodného prefixu

Prohlášení

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

.....
Jiří Fomiczew
5. května 2013

Poděkování

Chtěl bych poděkovat panu Ing. Michalu Kováčikovi za odborné vedení a vstřícnou pomoc při vypracování této práce. Dále bych chtěl poděkovat mé rodině, která mi vytvořila příznivé podmínky ke studiu a potřebné zázemí.

Obsah

Kapitola 1

Úvod

Dnes si bez internetu a počítačových sítí nedovedeme představit každodenní rutinu. Počet zařízení připojovaných do počítačových sítí roste obrovskou rychlostí. Dříve jsme se k internetu připojovali pouze pomocí osobních počítačů a rychlost připojení nebyla nikterak závratná, zatímco v současné době se k internetu připojujeme pomocí mobilních telefonů, tabletů, interaktivních televizí a mnoha dalších zařízení a naše nároky na rychlost připojení rostou. To ovšem znamená, že rostou i nároky na síťová zařízení.

Se zvyšováním rychlosti, a tedy i provozu na síti, se zmenšuje doba, kterou mají zařízení k dispozici pro provedení potřebných akcí nad každým paketem, který jimi prochází. V případě nejzákladnějších akcí jako směrování nebo filtrování paketů se doba jejich provádění musí pohybovat v řádu desítek nanosekund, takže každé urychlení těchto akcí hraje roli.

Pevnou součástí většiny základních operací je klasifikace paketů, jejíž důležitou součástí je nalezení nejlépe vyhovujícího pravidla v rámci množiny klasifikačních pravidel, a to na základě hodnot přečtených z hlaviček zpracovávaných paketů. Tato činnost je obvykle realizována pomocí tzv. vyhledání nejdelšího shodného prefixu – anglicky *Longest Prefix Match* (zkratka LPM).

Problém vyhledání nejdelšího shodného prefixu je možno řešit na dvou úrovních: na hardwarové a softwarové (algoritmické) úrovni. Na hardwarové úrovni se snažíme především o minimalizaci doby přístupu do paměti, protože při vyhledávání v nějaké množině prefixů je pro čtení jejich hodnot nutno opakovaně přistupovat do paměti, takže s velikostí množiny prefixů počet přístupů do paměti roste. Druhou úrovní je úroveň algoritmická, ve které se snažíme o efektivní uložení množiny prefixů v paměti tak, aby vyhledávání bylo rychlé, ale abychom zároveň minimalizovali počet přístupů do paměti, stejně jako paměťovou náročnost daného řešení (velmi rychlé paměti jsou obvykle také velmi drahé). Tyto požadavky často stojí proti sobě, takže je klíčové mezi nimi najít rozumný kompromis. Na této úrovni se jedná o *Algoritmy pro vyhledání nejdelšího shodného prefixu.*

Cílem této práce je navrhnout a implementovat program pro vizualizaci algoritmů pro vyhledání nejdelších shodných prefixů, jehož úkolem je implementovat a přehledně graficky zobrazit chování některých známých LPM algoritmů tak, aby ho bylo možno využít pro studijní a pedagogické účely. Výsledný program by tedy měl pomoci pochopit principy a chování několika zástupců existujících algoritmů, což je prvním krokem k případnému návrhu nových nebo optimalizaci stávajících řešení. Pro demonstraci byly vybrány různě sofistikované algoritmy tak, aby bylo možné porovnat jednoduchá a složitější řešení, a tedy i lépe pochopit, které cesty vedou k návrhu lepších a výkonnějších algoritmů. V programu budeme pracovat s prefixy ve formě IP adres, protože se jedná o nejtypičtější formu klasifikace a například vyhledání adresy ve směrovací tabulce je naprosto klasickým případem

vyhledání nejdelšího shodného prefixu.

Práce je logicky členěna do kapitol, jejichž obsah je následující: druhá kapitola obsahuje základní teoretické informace o architektuře počítačových sítí a základních operacích, při kterých je využíváno vyhledávání nejdelších shodných prefixů. Třetí kapitola obsahuje fakta o algoritmech pro vyhledání nejdelších shodných prefixů a přehled algoritmů implementovaných v programu s popisem jejich principu. Čtvrtá kapitola obsahuje souhrn požadavků na výsledný program a jeho počáteční návrh. Pátá kapitola pak popisuje implementaci programu, včetně použitých technologií.

Kapitola 2

Architektura počítačových sítí a základní operace

V této kapitole si stručně představíme architekturu počítačových sítí, popíšeme si protokol IP, jehož adresy jsou z našeho pohledu hlavním cílem použití algoritmů pro vyhledání nejdelších shodných prefixů a nakonec si přiblížíme hlavní operace, ve kterých se vyhledávání nejdelších shodných prefixů využívá. Informace obsažené v této kapitole jsem čerpal převážně z [12, 7, 14].

2.1 Architektura počítačových sítí

Architektura počítačových sítí v sobě zahrnuje specifikaci síťových zařízení a jejich fyzické propojení, definuje činnosti jednotlivých zařízení a také formát dat, se kterými se pracuje. Tyto atributy jsou definovány pomocí protokolů nebo protokolových sad. Vzhledem ke komplexnosti sítí se využívá vrstvových modelů, které síťovou architekturu rozdělují do tzv. vrstev.

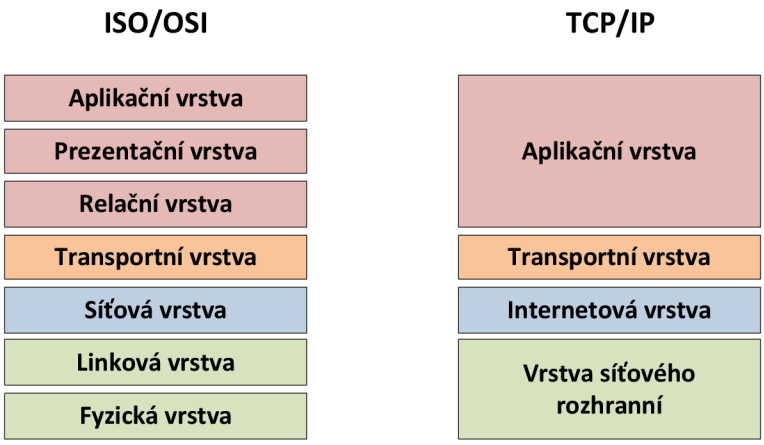
V praxi existují dva nejdůležitější vrstvové modely – referenční model ISO/OSI a model TCP/IP, který je dnes nejrozšířenější (používá se v Internetu).

Model ISO/OSI definuje sedm vrstev, u kterých říká, k čemu daná vrstva slouží a co zajišťuje. Model ISO/OSI tedy nespecifikuje konkrétní aplikační protokoly a služby, ale pouze definuje tyto vrstvy a jejich funkce. Z důvodu složitosti se model ISO/OSI nepoužívá k reálné implementaci, ale spíše jako teoretický model síťové komunikace. [12]

Pro reálnou implementaci se dnes používá model TCP/IP, který se skládá pouze ze čtyř vrstev:

- **Vrstva síťového rozhraní** – přístup k fyzickému médiu a komunikace na něm (Ethernet).
- **Internetová vrstva** – adresace a směrování v rámci globální sítě (protokol IP).
- **Transportní vrstva** – zajištění spolehlivého nebo nespolehlivého přenosu dat (protokoly TCP a UDP).
- **Aplikační vrstva** – komunikace mezi jednotlivými aplikacemi, aplikační protokoly (HTTP, FTP, DNS, ...).

Model TCP/IP je nejrozšířenější zejména z důvodu nezávislosti na použité síťové technologii, ale taky podporou napříč všemi operačními systémy.



Obrázek 2.1: Porovnání modelů ISO/OSI (vlevo) a TCP/IP (vpravo)

Na obrázku 2.1 vidíme porovnání obou modelů, ze kterého je vidět, že model TCP/IP slučuje tři horní vrstvy, stejně jako dvě spodní vrstvy modelu ISO/OSI do jedné vrstvy, čímž se oproti modelu ISO/OSI mění rozhraní mezi jednotlivými vrstvami. Nicméně funkčnost a význam jednotlivých vrstev je zachován.

2.2 Protokol IP

Z pohledu vyhledávání prefixů je pro nás nejdůležitějším protokolem protokol IP. Protokol IP (*Internet Protocol*) se používá na síťové vrstvě modelu ISO/OSI a jeho úkolem je zejména směrování zpráv zvaných *pakety* (IP pakety) napříč sítí. K tomu používá adresaci jednotlivých uzlů v síti pomocí tzv. *IP adres*. IP adresy musí být unikátní v rámci sítě, takže není možné, aby dvě různá zařízení měly v jedné síti stejnou IP adresu. Každý IP paket nese adresu odesílatele a adresu příjemce, pomocí které se provádí směrování. Protokol IP nenavazuje relace spojení, takže každý paket je v síti směrován individuálně. To znamená, že pakety nemusí přijít v pořadí, v jakém byly odeslány, a dokonce nemusí přijít vůbec. Tyto problémy řeší protokoly vyšší vrstvy (TCP). [12] V současné době existují dvě verze protokolu IP – IPv4 a IPv6.

2.2.1 IPv4

Protokol IP verze 4 používá adresy dlouhé 32 bitů, což znamená, že je celkem pro adresování k dispozici 2^{32} unikátních adres. Ve skutečnosti jsou některé adresové rozsahy rezervované či privátní, takže pro směrování v internetu je adres k dispozici méně. V lokálních sítích lze ovšem používat privátní adresy a nečerpat tak adresy z rozsahu globálních adres. K dalšímu snížení adresových nároků přispívá například technika překladu adres (NAT).

V současné době je adresový prostor IPv4 vyčerpán, jsou tedy přiděleny všechny adresovací bloky. Tento problém je hlavním důvodem k přechodu na protokol IPv6.

IPv4 adresy se zapisují ve tvaru `x.x.x.x` [12], kde `x` představuje vyjádření hodnoty daného bytu v desítkové soustavě a jednotlivé byty jsou odděleny tečkou. Příkladem IPv4

adresy je 192.168.10.1 (privátní adresa), nebo 147.229.12.91 (veřejná adresa).

Dalším formátem zápisu IPv4 adresy je tzv. *prefixový zápis*, kdy se za adresu uvádí počet bitů adresové části adresy, které určují tzv. *masku sítě*, pomocí které můžeme z dané adresy určit, která část se použije pro adresování sítě a která část pro adresování koncového zařízení v dané síti. Příkladem prefixového zápisu je adresa 194.16.30.5/24, kdy jako adresu sítě použijeme prvních 24 bitů dané adresy (tedy adresa sítě by byla 194.16.30.0). Prefixový zápis adres se často používá v souvislosti s použitím algoritmů LPM, můžeme ho vidět např. ve směrovacích tabulkách. [14, 7]

2.2.2 IPv6

Protokol IP verze 6 byl navržen jako nástupce protokolu IPv4 a oproti IPv4 přináší řadu změn. Hlavní změnou je rozdílná délka adresy (128 bitů), která nám dává obrovský adresový prostor (2^{128} adres), a tedy řeší problém vyčerpání adres. IPv6 paket má oproti IPv4 odlišnou strukturu hlavičky, která obsahuje podstatně méně položek než u IPv4. Mezi další změny oproti IPv4 patří zvýšení bezpečnosti, možnost automatické konfigurace stanic, podpora kvality služeb (*QoS*) a další. [12]

IPv6 adresa je zapisována ve tvaru `x:x:x:x:x:x:x:x`, kde `x` je jedno až čtyři hexadecimální čísla reprezentující celkem 16 bitů. Každá čtveřice je oddělena dvojtečkou. V zápisu je možno vynechat nuly, které se mohou nacházet na začátku každé čtveřice. Nelze ovšem vynechat nuly uvnitř či na konci čtveřice, jelikož by nebylo možné informaci správně interpretovat. Další možností je, že pokud některá 16 bitová skupina obsahuje samé nuly nebo pokud takovýchto skupin je více za sebou, je možno je vynechat a v zápise adresy je nahradit symbolem „::“. Tento symbol je ovšem možno v adrese použít pouze jednou (jinak by nebylo možno dopočítat kolik čtveřic jsme vlastně vynechali). [12, 5] Zápis prefixu u IPv6 adres je totožný jako u IPv4 adres.

IPv6 adresa	**Úpravy**
2001:0DB8:0000:0000:0008:0800:200C:417A	Bez úprav (plný tvar)
2001:DB8:0000:0000:8:800:200C:417A	Vynechané nuly na začátku čtveřic
2001:DB8::8:800:200C:417A	Vynechání nulových čtveřic

Tabulka 2.1: Různé způsoby zápisu jedné IPv6 adresy

2.3 Základní operace

V této kapitole si popíšeme nejzákladnější a nejdůležitější operace v počítačových sítích. Těmito operacemi jsou klasifikace paketů, směrování paketů a filtrování paketů. Tyto základní operace typicky probíhají nad každým paketem putujícím po síti, a to na zařízení, kterým pakety prochází. Z tohoto důvodu je u nich kritická rychlost provádění těchto operací. Např. na 10 Gbps lince je maximální doba pro zpracování jednoho paketu o velikosti 40 B 32 ns.

2.3.1 Klasifikace paketů

Klasifikace paketů patří mezi základní operace na síti. Pro každý paket, který přijde na síťové zařízení, je třeba provést jeho klasifikaci, na jejímž základě je rozhodnuto, co se s paketem bude dít dále. Klasifikace se provádí na základě množiny klasifikačních pravidel a údajů z hlavičky paketu. V závislosti na konkrétním klasifikačním problému se klasifikuje především podle IP adres (zdrojových a cílových), čísel portů (zdrojových a cílových) nebo protokolu (TCP, UDP, IP, ICMP, ...). Klasifikace může probíhat i ve více dimenzích (klasifikace na základě více atributů – např. adres, portů i protokolu najednou).

Výsledkem klasifikace je jedno konkrétní pravidlo z množiny pravidel, které určí, co s paketem. Pokud hodnotám z hlavičky paketu vyhovuje více pravidel, je uplatněna priorita pravidel [14], takže je vybráno vyhovující pravidlo s nejvyšší prioritou.

Klasifikace paketů je základní operací, která je součástí operací, jako je směrování nebo filtrování. Proto je nutné hledat možnosti k jejímu maximálnímu urychlení. Jak již bylo zmíněno v úvodu, nabízejí se dvě hlavní cesty: *hardwarová* (např. využití speciální paměti TCAM) a *softwarová* (algoritmická). [6]

Co se algoritmických metod klasifikace paketů týče, lze je rozdělit na úplné prohledávání, použití rozhodovacích stromů, dekompoziční metody a metody rozdělující stavový prostor. [13]

Pro nás je zajímavá dekompoziční metoda, jejíž činnost se dá rozdělit do dvou hlavních bodů: vyhledání nejdelších shodných prefixů nad položkami z hlaviček paketů a následné určení vyhovujícího pravidla. My se v této práci budeme zabývat prvním z těchto bodů, tedy demonstrací principů algoritmů pro vyhledání nejdelšího shodného prefixu.

2.3.2 Filtrování paketů

Úkolem filtrování paketů je na základě množiny filtrovacích pravidel propouštět či zahazovat pakety putující sítí. Filtrování paketů je jedna z nejzákladnějších operací pro zvýšení bezpečnosti sítě, filtrovací zařízení jsou označovány jako *firewall*.

Filtrovací pravidla obsahují hodnoty atributů nacházející se v hlavičkách paketů (zpravidla se rozhoduje na základě IP adres a čísel portů, případně protokolu) a akci, která specifikuje, co se s paketem má stát (povolit, zahodit), pokud hodnoty z hlaviček aktuálního paketu vyhovují tomuto pravidlu. Během filtrování se tedy nad každým paketem provádí jeho klasifikace (vyhledání nejdelších shodných prefixů daných atributů – IP adres, portů). Filtrovací pravidla jsou seřazeny podle priority, obvykle od nejspecifičtějších po nejméně specifické. [14]

Na následujícím příkladu je ukázka filtrovacích pravidel, které propustí komunikaci ze sítě 192.168.1.0/24 s DNS serverem na adrese 80.20.30.3. Ostatní provoz je zakázán.

```
10 permit udp from 192.168.1.0/24 to host 80.20.30.3 dst-port 53
20 permit udp from host 80.20.30.3 to 192.168.1.0/24 src-port 53
30 deny ip any any
```

2.3.3 Směrování

Aby paket mohl úspěšně doputovat od jednoho počítače k druhému, musíme nějak zajistit, že zařízení budou vědět, kam mají paket poslat, aby úspěšně dorazil ke svému cíli. Toto rozhodování, kudy se má paket vydat, se nazývá *směrování*. Směrování probíhá na základě

IP adresy cíle umístěné v hlavičce každého paketu a mají ho na starosti zařízení zvaná *směrovače* (anglicky „*routery*“).

Každý směrovač má v tzv. *směrovací tabulce* obsaženy informace o sítích, do kterých směruje, tedy ví, kam paket poslat, aby se do daných sítí dostal. Směrovací tabulka pak obsahuje adresy sítí a identifikaci dalšího směrovacího kroku (adresa dalšího směrovače nebo název rozhraní, na které se paket odešle), tzv. *next-hop*. [12, 7] Vyhledání adresy sítě ve směrovací tabulce je vlastně vyhledání nejdelšího shodného prefixu pro danou vstupní adresu (cílová adresa ze směrovaného paketu). Příklad směrovací tabulky je uveden v tabulce 3.1.

Pro výměnu směrovacích tabulek mezi směrovači slouží tzv. *dynamické směrovací protokoly*, které zajišťují, že všechny směrovače mají ve svých směrovacích tabulkách stejné informace tak, aby nevznikaly chyby nebo směrovací smyčky. Příklady dynamických směrovacích protokolů jsou protokoly RIP, OSPF, EIGRP a další.

V Internetu existuje síť směrovačů, které mezi sebou směrují veškerý provoz. Proto je nutné, aby směrování byla velice rychlá, ale taky přesná operace, tedy aby pakety putovaly pokud možno co nejkratší (nejrychlejší) cestou ke svému cíli. Směrovací tabulky na takovýchto směrovačích mohou mít i tisíce položek, což klade důraz na použití rychlých algoritmů LPM.

Kapitola 3

Algoritmy pro vyhledání nejdelšího shodného prefixu

V této kapitole si představíme zejména obecný princip rodiny algoritmů pro vyhledávání nejdelšího shodného prefixu (dále jen LPM – *longest prefix match*) a dále několik jejich zástupců, jenž jsou implementovány ve vizualizačním programu. U každého algoritmu si vysvětlíme princip jeho fungování a zhodnotíme jeho klíčové parametry. Hlavními zdroji informací pro tuto kapitolu byly [7] a [14].

3.1 Obecný princip vyhledávání nejdelšího shodného prefixu

Úkolem LPM algoritmů je v množině známých prefixů (např. směrovací tabulka) vyhledat ten, který má nejdelší shodu se vstupní hodnotou (typicky IP adresou).

Prefix	**Rozhraní**
192.168.1.0/24	Fa0/0
172.16.0.0/16	Fa0/1
172.16.1.0/24	Fa0/2

Tabulka 3.1: Příklad směrovací tabulky

V tabulce 3.1 vidíme směrovací tabulku, která obsahuje prefixy ve formě adres sítí, do kterých router směruje pakety. U každé takové adresy je název rozhraní, přes které se budou pakety určené pro danou síť posílat. Pokud přijde paket s cílovou adresou 192.168.1.1, odpovídá této adrese první prefix a paket se pošle na rozhraní Fa0/0. Pokud ovšem přijde paket s cílovou adresou 172.16.1.1, vyhovuje této adrese jak druhý, tak třetí prefix. Jelikož ale bude pro tuto adresu nejdelší shodný prefix síť s maskou /24, tak se paket odešle přes rozhraní Fa0/2.

Číslo prefixu	**Prefix**
1	0101*
2	01100*
3	01*
4	01010*

Tabulka 3.2: Binární reprezentace prefixů

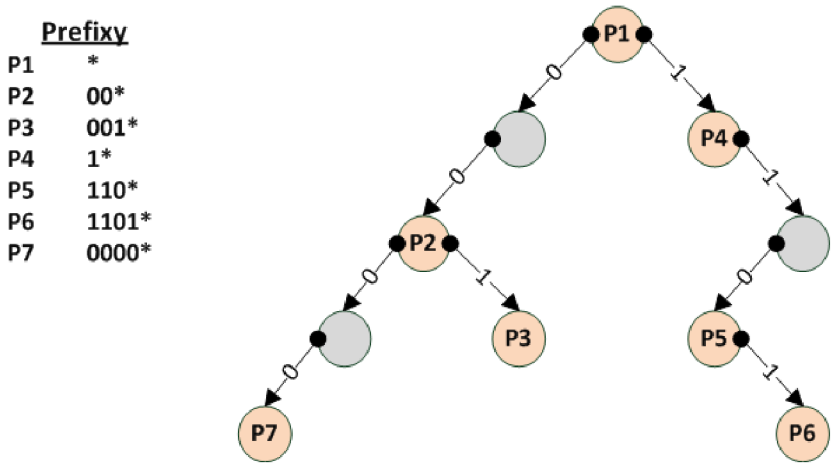
V tabulce 3.2 je ukázka prefixů v binární reprezentaci. Znak ∗ znamená, že další bity mohou být libovolné. Je-li vstupní hodnotou např. hodnota 0110, je nejdelším shodným prefixem třetí prefix (01*) (druhý prefix je delší než hodnota, tedy ten to být nemůže). Naopak pokud bude vstupní hodnotou např. 0101001, tak nejdelším shodným prefixem bude čtvrtý prefix (odpovídá mu prvních 5 bitů).

V našem případě, kdy uvažujeme vyhledávání nad IP adresami, mají tyto prefixy délku 0–32 bitů (IPv4 adresy) nebo 0–128 bitů (IPv6 adresy). Je zřejmé, že v případě IPv6 adres je vzhledem k jejich délce vyhledání nejdelšího shodného prefixu několikrát náročnější než u IPv4 adres. Liší se samozřejmě také paměťová náročnost algoritmů, která je u IPv6 adres mnohonásobně větší. Algoritmy LPM většinou pracují nad binární podobou adres [10], takže algoritmy pro práci s IPv4 adresami jsou po drobných úpravách většinou schopny pracovat i s IPv6 adresami. Je ovšem nutné podotknout, že zatímco při práci s IPv4 adresami mohou být některé algoritmy velmi rychlé a hojně používané, tak při práci s adresami IPv6 mohou být naopak nevyhovující (ať už rychlostně nebo třeba kvůli obrovské paměťové náročnosti) a je třeba použít jiné. S postupným rozmachem protokolu IPv6 je tedy třeba dále hledat nové, vhodnější cesty pro co nejrychlejší vyhledávání prefixů.

3.2 Algoritmus Unibit Trie

Unibit Trie představuje základní a nejjednodušší algoritmus pro LPM založený na stromové struktuře *Trie*[1]. Slovo *Unibit* znamená, že v každém uzlu stromu je uložena hodnota jednoho bitu, tedy výška stromu odpovídá délce nejdelšího prefixu (v bitech).

V každém uzlu je obsažena bitová hodnota (1 nebo 0), která představuje hodnotu na dané pozici v prefixu. Spolu s touto hodnotou je také uložena informace, zdali uzel nese hodnotu nějakého prefixu nebo ne. Každý uzel také obsahuje dva ukazatele na následující uzly: *levý* s hodnotou 1, *pravý* s hodnotou 0. Kořenový uzel stromové struktury nese libovolnou hodnotu (*).



Obrázek 3.1: Ukázka stromu trie s uloženými prefixy

Během vyhledávání čteme bity vstupní adresy postupně od nejvýznamnějšího (MSB) po nejméně významný (LSB) a podle jejich hodnot procházíme stromem (podle hodnoty

[1] Z anglického slova *retrieval* – získávání, vyhledávání.

aktuálního bitu se rozhodujeme mezi levým (0) a pravým (1) ukazatelem). Pokud uzel obsahuje prefix, zapamatujeme si jeho hodnotu jako hodnotu nejdelšího prefixu. Tím zajistíme to, že při neshodě dalších bitů adresy se nebudeme muset ve stromě vracet kvůli zjištění hodnoty prefixu. Algoritmus končí, pokud jsme přečetli všechny bity adresy nebo pokud už v průchodu stromem nelze pokračovat (nacházíme se v listu stromu). Hodnota posledního zapamatovaného uzlu je zároveň hodnotou nejdelšího shodného prefixu pro danou adresu. Hodnotu prefixu reprezentuje cesta (hodnoty hran) od kořene stromu do posledního zapamatovaného uzlu. [14]

Algoritmus v základní podobě dosahuje lineární asymptotické složitosti $O(n)$, kde n je délka vstupní adresy. Paměťová náročnost je přímo úměrná počtu uložených prefixů a jejich maximální délce. Počet přístupů do paměti je maximálně roven délce vstupní adresy. [10, 6]

V praxi se algoritmus v této základní podobě moc nepoužívá, ale je základem pro další algoritmy, které jej dále modifikují.

3.3 Algoritmus Controlled Prefix Expansion

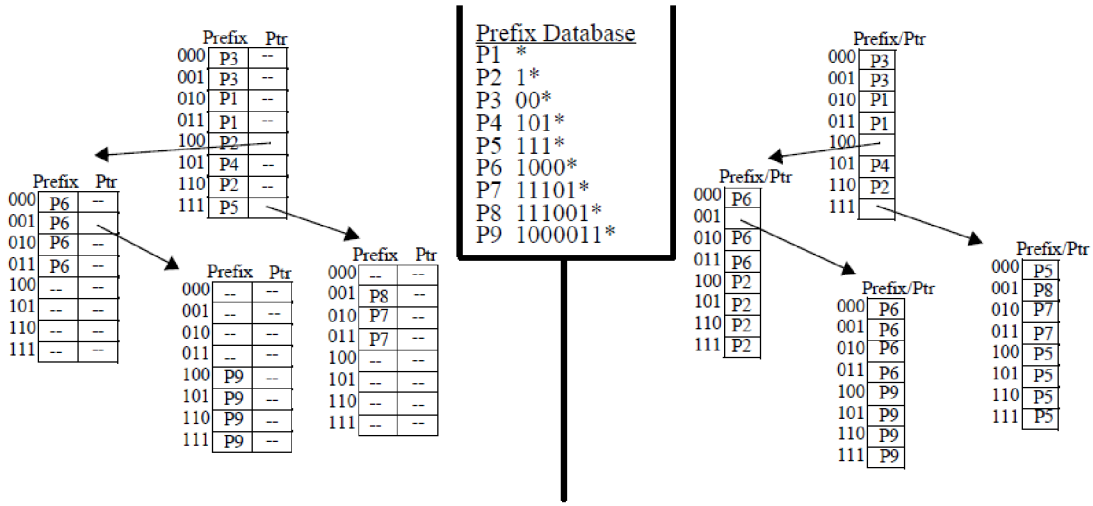
Algoritmus Controlled Prefix Expansion (značen také CPE) je modifikací algoritmu Unibit Trie. Také používá stromovou strukturu *Trie* pro uložení prefixů, samotné její uzly však mají jinou strukturu.

Základní myšlenkou algoritmu CPE je sjednotit délky uložených prefixů na násobky n, kde n je volitelná délka kroku – tzv. *stride length*. K tomu se používá techniky zvané expanze prefixů. Prefixy, které mají délku m, která je menší než n, musíme expandovat na požadovanou délku n. Musíme tedy k prefixu doplnit dalších $n-m$ bitů, které tedy ponesou 2^{n-m} dalších kombinací. Například prefix 11* můžeme expandovat na čtyři (2^{4-2}) prefixy délky $n = 4$: (1100*, 1101*, 1110*, 1111*). Při expanzi se berou v potaz ostatní existující prefixy, takže pokud se expandovaná hodnota shoduje s jiným existujícím prefixem, je použit přímo tento prefix. [11]

Parametr *stride length* (n) je volitelný a udává kolik bitů bude kódovat každý uzel stromu. S ohledem na tuto informaci a také s nutností expanze každého prefixu na požadovanou délku je třeba tento parametr volit velmi obezřetně – ovlivňuje jak rychlost algoritmu, tak jeho paměťové nároky. S vyšší hodnotou parametru se zvyšují paměťové nároky algoritmu, ale i jeho rychlost. Pro reálné použití je vhodné volit délku střídy 3 až 8, jinak se ztrácejí výhody algoritmu.

Každý uzel stromu obsahuje pole prefixů (prefixy dlouhé n bitů), které má kvůli expanzi uzlů velikost 2^n položek. Dále uzel obsahuje pole ukazatelů na následující bity prefixů (další uzly), také o velikosti 2^n položek. Počet následovníků každého uzlu je tedy 2^n. V případě, že některé prefixy nemají následovníky, není prostor v poli ukazatelů smysluplně využit, což zbytečně zvyšuje paměťovou náročnost. Kvůli tomuto byla zavedena optimalizace zvaná *leaf pushing*. Smyslem optimalizace je neplýtvat s pamětí. Místo dvou polí se v každém uzlu nachází pouze jedno pole o 2^n položkách, jehož položky obsahují buď hodnotu prefixu nebo ukazatel na následující uzel. Toto ovšem s sebou přináší i nevýhodu, že prefixy o délce n (neexpandované) jsou uloženy až v následujícím uzlu, a to v expandované podobě. Jejich místo totiž nahradil ukazatel, a proto je nutné je uložit až v následujícím uzlu. [11, 6]

V obrázku 3.2 vlevo vidíme strukturu uzlu algoritmu CPE bez použití optimalizace *leaf pushing*, kdy uzel obsahuje obě pole (prefixy i ukazatele na následovníky), zatímco napravo je použita metoda *leaf pushing* a uzel obsahuje pouze jedno pole. Povšimněte si, že např. prefix P5 je uložen až v následovníkovi, a to v expandované podobě (kombinace 111001*, 111010* a 111011* jsou součástí jiných prefixů P8 a P7, takže nepatří k prefixu P5).



Obrázek 3.2: Algoritmus CPE bez (vlevo) a s využitím techniky *leaf pushing* (vpravo). Obrázek převzat a upraven z [4].

Hlavní výhodou algoritmu CPE je zpracování více bitů najednou, a tedy větší rychlost a menší počet přístupu do paměti (čtení). Nevýhodou je ovšem větší paměťová náročnost z důvodu expanze prefixů.

3.4 Algoritmus Tree Bitmap

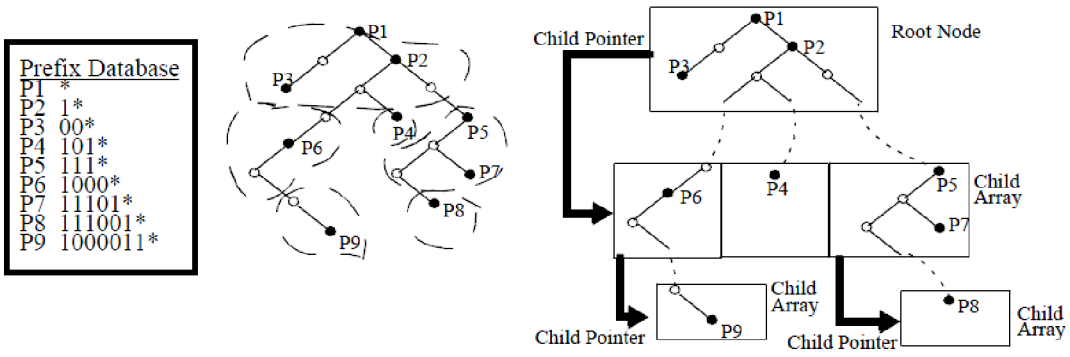
Tree Bitmap je algoritmus založený na stromové struktuře *Trie*, ovšem na rozdíl od algoritmu Unibit Trie umožňuje uložení více bitů vstupního prefixu v rámci jednoho uzlu. Patří tedy mezi *multibit* algoritmy, které zpracovávají více bitů vstupního prefixu současně. Je tedy složitější, než Unibit Trie, ale také výrazně efektivnější.

Každý uzel stromu Tree Bitmap reprezentuje několik uzlů struktury *Trie*, kde počet bitů uložených v každém uzlu je parametrizovatelný. Od tohoto parametru (říkejme mu n, nebo *stride*) se odvíjí počet potomků každého uzlu (2^n) a každý uzel představuje n úrovní struktury *Trie*. [10]

Potomci každého uzlu jsou uloženi v paměti za sebou, takže stačí ukládat ukazatel na prvního potomka a adresy ostatních potomků můžeme dopočítat pomocí offsetu. Místo 2^n ukazatelů na potomky máme tedy pouze jeden ukazatel na prvního potomka, což výrazně zmenšuje paměťové nároky algoritmu. Potomci uzlu, kteří neobsahují žádné prefixy ani svoje potomky, jsou vynecháni. Platnost jednotlivých potomků je uložena v bitmapě (bitovém poli), kde hodnota 1 symbolizuje, že potomek existuje, hodnota 0, že nikoliv. Tato bitmapa se nazývá *externí bitmapa* a je uložena v každém uzlu. [6]

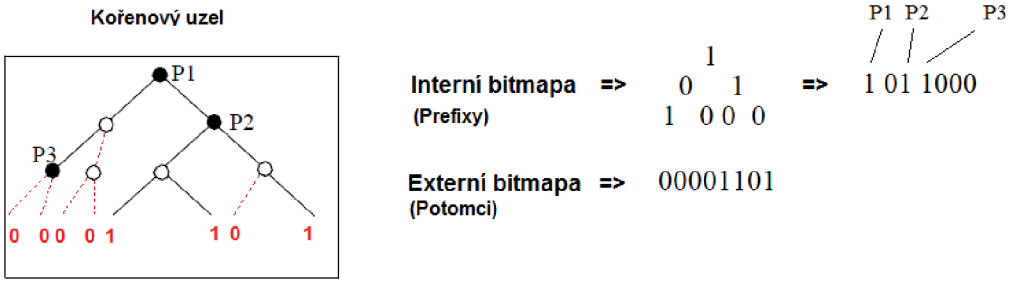
Offset příslušného potomka k ukazateli na prvního potomka tedy zjistíme spočítáním existujících potomků (hodnota 1) uložených v externí bitmapě před hledaným potomkem.

Každý uzel obsahuje ještě další bitmapu pro uložení prefixů obsažených v daném uzlu, kde podobně jako v předchozím případě hodnota 1 znamená, že prefix na dané pozici existuje, hodnota 0, že nikoliv. Tato bitmapa se nazývá *interní bitmapa*. Hodnoty prefixů jsou také uloženy v paměti za sebou, takže postačí pouze jeden ukazatel na první prefix, adresy dalších prefixů dopočítáme stejným způsobem jako u externí bitmapy. Velikost interní bitmapy je $(2^n - 1)$ bitů a velikost externí bitmapy je 2^n bitů.



Obrázek 3.3: Příklad stromu Tree Bitmap s parametrem $n = 3$. Obrázek převzat z [4].

Při vyhledávání v *Trie* strukturách začínáme vždy v kořenovém uzlu a ani Tree Bitmap není výjimkou. V každém kroku vezmeme n bitů vstupní hodnoty, které použijeme jako index do externí bitmapy. Pokud se na dané pozici v externí bitmapě nachází jednička, znamená to, že budeme ve vyhledávání pokračovat v podstromu, jehož ukazatel zjistíme pomocí offsetu (počet jedniček nacházejících se v externí bitmapě před danou pozicí). Před přesunem vyhledávání do daného podstromu se ještě pomocí interní bitmapy zkontrolují prefixy v tomto uzlu, kdy si uložíme referenci na nejdelší shodný prefix v této úrovni. To provedeme tak, že postupně ubíráme nejméně významné bity z indexu (n bitů vstupní hodnoty), dokud na příslušném poli v interní bitmapě nenalezneme hodnotu 1 (tedy existující prefix). Tato vyhledávací sekvence se opakuje, dokud nezpracujeme všechny bity vstupní adresy nebo pokud se dostaneme na konec stromu (listový uzel). Nejdelším nalezeným prefixem je ten, jehož referenci máme uloženu po provedení posledního kroku. [4]



Obrázek 3.4: Ilustrace uložení dat do bitmap. Obrázek převzat a upraven z [4].

Hlavní předností algoritmu je vysoká rychlost vyhledávání, nízká paměťová náročnost (v závislosti na počtu bitů ukládaných v každém uzlu) a snadná HW realizace. Pro efektivitu je důležitá správná volba parametru n (v praxi většinou $n = 3$, nebo $n = 4$, pro testování i vyšší hodnoty). Při vysokých hodnotách n se zvyšují paměťové nároky algoritmu, při nízkých hodnotách se zase snižuje jeho efektivita. Časová složitost algoritmu je lineární a závisí na délce vstupní hodnoty (IP adresy). Jelikož jsou vstupními hodnotami IP adresy s konstantní délkou, je možno říci, že i časová složitost je konstantní. Algoritmus je vhodný pro použití s IPv4 adresami, naopak s IPv6 adresami je jeho paměťová náročnost příliš velká. [6]

Kapitola 4

Požadavky na výslednou aplikaci a její návrh

V této kapitole se budeme věnovat návrhu aplikace pro vizualizaci algoritmů LPM. Nejprve si shrneme základní požadavky na výslednou aplikaci a poté se zamyslíme nad návrhem aplikace, která bude těmto požadavkům vyhovovat.

4.1 Požadavky na výslednou aplikaci

Ještě než se pustíme do návrhu, shrňme si, k čemu by výsledná aplikace vlastně měla sloužit a jaké další vlastnosti by měly být splněny.

Výsledná aplikace by měla přehledně graficky zobrazit principy a chování jednotlivých LPM algoritmů tak, aby ji bylo možno využít jak k pedagogickým účelům v podobě názorných ukázek, tak k samostudiu a následnému pochopení základních principů vyhledávání nejdelších shodných prefixů.

Z tohoto tedy plyne, že aplikace by měla implementovat chování vybraných LPM algoritmů, ale zároveň by bylo vhodné, aby bylo výhledově možné přidávat další algoritmy, aniž by bylo nutné výrazně zasahovat do zdrojového kódu. Co se samotných algoritmů týče, měla by existovat možnost zobrazení jejich vnitřní struktury (u algoritmů využívající stromovou strukturu by mělo být možné zobrazit celý strom skládající se z jednotlivých uzlů apod.).

Jelikož algoritmy operují nad nějakou množinou prefixů, ze které následně vyhledávají nejdelší shodu, je třeba, aby i v programu bylo možno tyto prefixy nějakým způsobem zobrazit a samozřejmě by neměla chybět možnost tyto prefixy přidávat, ukládat nebo načítat ze souboru.

Prefixy samotné budou ve formě IP adres v prefixovém zápisu, takže aplikace bude do jisté míry představovat simulaci vyhledávání next-hop adresy ve směrovací tabulce (nepřímo, výsledkem vyhledávání bude nejdelší shodný prefix z množiny prefixů, ne next-hop adresa). Tento krok povede k lepšímu pochopení algoritmů, jelikož vyhledávání nad IP adresami je relativně známá a dobře představitelná činnost. Pro jednoduchost a názornost budeme používat IPv4 adresy.

Velmi důležitý bude také proces samotného vyhledávání. Na vstupu vyhledávání bude IPv4 adresa, pro kterou budeme hledat nejdelší shodný prefix v tabulce prefixů, což vlastně nebude nic jiného, než nalezení sítě, do které adresa patří. V tomto smyslu si můžeme představit, že vstupní adresa je adresou přečtenou z pole *Destination address* z paketu, který směrujeme a naším úkolem je rozhodnout, do které sítě adresa patří, abychom paket

na základě této znalosti poslali na správné rozhraní směrovače. Proces vyhledávání se bude samozřejmě pro jednotlivé algoritmy LPM lišit, proto je nutné, aby bylo nějakým způsobem možno rozlišit a porovnat jednotlivé kroky vyhledávacího procesu každého algoritmu.

Vzhledem k potencionálnímu pedagogickému uplatnění aplikace je nutné, aby její uživatelské rozhraní bylo přívětivé, přehledné a jednotné tak, aby uživatel měl všechny potřebné informace přehledně zobrazeny a nemusel moc přemýšlet, co kde má hledat, a co které položky představují. Jednotné musí být uživatelské prostředí zejména ve vztahu k různým vyhledávacím algoritmům, tedy aby pro různé algoritmy bylo uživatelské rozhraní pokud možno neměnné.

4.2 Návrh aplikace

Jakmile jsme si ujasnili požadavky, které by měla výsledná aplikace splňovat, můžeme přejít k abstraktnímu návrhu aplikace a zamyslet se nad problémy, které budeme muset při implementaci vyřešit.

Na základě požadavků jsem se rozhodl pro implementaci využít metod *objektově orientovaného programování* (OOP), které je pro aplikaci tohoto typu velmi vhodné a návrh některých komponent značně zjednodušuje. Vzhledem k existenci různých algoritmů, které budou ve výsledku nevyhnutelně mít určitou část metod takřka totožnou či úplně totožnou, se přímo nabízí využít principy jako *abstraktní třídy* s implementací společných metod pro všechny algoritmy či *virtuální metody*, pomocí kterých lze pro různé algoritmy implementovat danou funkčnost individuálně, nicméně s jednotným rozhraním.

4.2.1 Algoritmy LPM a jejich reprezentace

Co se samotných algoritmů týče, nejvhodnějším řešením se nabízelo vytvoření abstraktní třídy, která by definovala standardní sadu metod pro všechny algoritmy. Pro každý algoritmus by poté byla vytvořena nová třída, která by za pomoci dědičnosti dědila rozhraní od této hlavní abstraktní třídy. Některé metody společné pro všechny algoritmy by byly implementovány přímo v dané abstraktní třídě, ostatní metody by byly virtuální a jednotlivé algoritmy by je musely redefinovat v jejich třídách. Další pomocné a individuální metody používané pouze konkrétním algoritmem by bylo možno dodefinovat v třídě příslušející danému algoritmu, takže by nijak neovlivňovaly rozhraní pro ostatní algoritmy. Toto řešení se ukázalo být výhodné jak z hlediska implementace, tak z hlediska přehlednosti kódu a náročnosti přidávání dalších algoritmů, proto bylo nakonec použito i ve finální implementaci.

Konkrétními příklady metod, které bude v případě algoritmů nutné implementovat, jsou například metody pro vyhledávání, metody pro konstrukci jejich vnitřních struktur, metody související s pokročilou vizualizací vyhledávání a podobně.

4.2.2 Reprezentace vnitřních struktur algoritmů LPM

Aby byla vizualizace algoritmů LPM dostatečně názorná, zahrnuje program možnost zobrazení vnitřní struktury jednotlivých algoritmů. U našich tří základních algoritmů se jedná o stromové struktury, kdy je strom tvořen uzly, jejichž obsah se liší podle algoritmu (dalším rozdílem mezi algoritmy je použití binárních a n-árních stromů).

Tyto uzly mají podobně jako jednotlivé algoritmy určité společné znaky, ale např. vnitřní struktura uzlů se v každém algoritmu velmi liší. Proto se pro implementaci uzlů nabízí stejné řešení jako u algoritmů, tedy existence jedné hlavní abstraktní třídy, která implementuje

některé metody společné pro všechny typy uzlů a zároveň poskytuje standardní rozhraní za pomoci virtuálních metod, jejichž implementace se v jednotlivých typech uzlů liší. V tomto případě je ovšem počet takovýchto metod nižší než u algoritmů, jelikož se jednotlivé typy uzlů liší převážně vnitřní implementací, což si vyžaduje různé přístupy a metody pro práci s nimi. To nám ovšem příliš nevadí, protože s daným typem uzlů pracuje pouze algoritmus používající tento typ uzlů, a tedy i implementace metod pro práci s daným typem uzlu je přizpůsobena pro co nejsnazší použití v algoritmu.

Kromě specifické vnitřní struktury a práce s ní musí uzly také obsahovat metody pro jejich vizualizaci, tedy pro grafické vykreslení vlastních uzlů. To, jakým tvarem, barvou, s jakými informacemi apod. bude uzel vykreslen se samozřejmě pro různé typy uzlů liší, proto musí být tyto metody implementovány individuálně pro každý typ uzlu. Principiálně by bylo možné oddělit vnitřní strukturu uzlu a jeho grafickou podobu do dvou tříd, což by umožnilo lepší izolaci těchto dvou problémů a teoreticky i snazší lokalizaci případných chyb. Nicméně nakonec jsem se rozhodl takto oddělenou funkčnost uzlů neimplementovat, a to zejména z důvodu snížení celkového počtu tříd v programu a také z důvodu snazšího předávání potřebných informací mezi těmito dvěma oblastmi. Další podrobnosti o implementaci uzlů jsou detailně popsány v kapitole věnující se jejich konkrétní implementaci.

4.2.3 Vizualizace vyhledávání

Nejdůležitější funkcí výsledné aplikace bude demonstrace vyhledání nejdelšího shodného prefixu nad množinou prefixů a vstupní adresou. Aby byl proces vyhledávání vizualizován s maximální názorností, měla by aplikace uživateli umožnit názorně zobrazit průběh vyhledávacího procesu s jeho jednotlivými kroky, a to individuálně pro každý algoritmus.

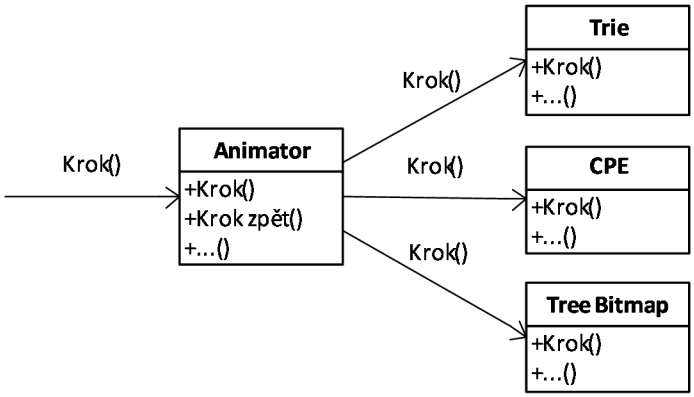
Vyhledávací proces jsem se tedy rozhodl navrhnout jako posloupnost kroků, jejichž činnost bude v reálném čase zobrazena přímo nad vizualizací vnitřní struktury algoritmů, tedy např. nad jejich stromy. Během vyhledávání budou ve stromu názorně zobrazovány uzly, ve kterých právě hledáme, ukazatele použité pro přechod do dalšího uzlu a podobně. Pokud bude proces vyhledávání pro každý algoritmus rozdělen na kroky, ve kterých jsou vykonány určité akce, můžeme vyhledávácí proces zobrazit jako posloupnost akcí od začátku vyhledávání až po jeho konec, a pokud mezi jednotlivé kroky vložíme určitý čekací čas, můžeme proces vyhledávání efektně animovat. Zároveň bude ovšem možné vyhledávací proces krokovat manuálně, tedy ručně přepínat jednotlivé kroky v časových intervalech dle přání uživatele, což je vhodné, například pokud aplikaci využijeme pro vysvětlení principu algoritmů LPM a vizualizaci budeme doprovázet vhodným slovním komentářem, jehož délka může být pro každý krok jiná. Pro maximální názornost bude kroky možné vykonat i v opačném pořadí, tedy se z určitého kroku vrátit k předchozímu, čímž umožníme uživateli vrátit se třeba i na samotný začátek celého vyhledávání. Tím se stane vyhledávaní interaktivním procesem, což by každému uživateli mělo pomoci k plnému pochopení fungování implementovaných algoritmů LPM.

Zároveň aplikace nabídne i možnost vyhledání bez animace, tedy že bude ihned zobrazen výsledek vyhledávání. Tato volba může sloužit např. k rychlé demonstraci problému vyhledání cílové sítě během směrování či k prostému ověření nebo získání výsledku vyhledávání. Aby bylo možno implementovat výše uvedené chování, bude muset každý algoritmus implementovat vyhledávácí proces s možností vykonání jednotlivých kroků a samozřejmě zajistit, aby bylo kroky možné vykonat v obráceném pořadí. Toto lze jednoduše zajistit pomocí reimplementace virutálních metod definovaných v hlavní abstraktní třídě, ze které budou algoritmy odvozeny (viz výše). Jelikož bude během těchto kroků vykonáváno spoustu akcí

souvisejících s korektní vizualizací daného kroku ve stromu daného algoritmu, bude nejvhodnější vytvořit ještě jednu speciální metodu pro vyhledání bez animace, která bude oproštěna od těchto vizualizačních akcí, a tedy i její vykonání bude rychlejší, než pokud bychom v rychlosti provedli posloupnost vizualizačních kroků.

Aby animace vyhledávání vůbec mohla proběhnout, musí být nějak řízena, tedy musí být spouštěny příslušné kroky v určitých časech atd. Proto je vhodné vytvořit novou třídu, která toto chování bude zabezpečovat. Jejím úkolem bude řídit animaci vyhledávání a zároveň komunikovat s algoritmy, aby bylo možné např. určit konec vyhledávání, a tedy zastavit celou animaci.

Teď, když jsme navrhli vytvořit třídu pro řízení animace, se nabízí otázka, jak se vypořádat s možností změny algoritmu během vyhledávání a zda takovou možnost uživateli vůbec povolit. Nejjednodušším řešením by bylo uživateli znemožnit přepnutí algoritmu během vyhledávání. Tato cesta by zajisté byla implementačně nejméně náročná a celkově by nepřinášela výraznější problémy. Na druhou stranu je jasné, že by tato možnost na atraktivitě nepřidala a uživatel by mohl cítit, že je potřeba vynaložit příliš velké úsilí pro změnu algoritmu (v tomto případě by musel ukončit aktivní vyhledávání, přepnout algoritmus a poté případně vyčkat či posunout proces vyhledávání do určitého místa). Proto bude nejvhodnější zvolit alternativní cestu v podobě povolení možnosti přepínání algoritmů během procesu vyhledávání (přepnutím algoritmu se rozumí zobrazení jeho stavu, tedy stromu s označenými uzly apod.). Aby uživatel mohl plynule přecházet mezi jednotlivými algoritmy, aniž by musel každý algoritmus „krokovat“ zvlášť, bude nejvhodnější, aby bylo vyhledávání animováno ve všech algoritmech současně, tedy všechny algoritmy by dělaly kroky vyhledávacího procesu ve stejný čas a synchronizovaně. Tím umožníme uživateli, aby mohl libovolně přepínat zobrazený algoritmus během vyhledávacího procesu a mohl tedy interaktivně sledovat, jak jednotlivé algoritmy postupují v dané fázi vyhledávání.



Obrázek 4.1: Provedení kroku animace pro všechny algoritmy.

Zároveň ovšem vyvstává otázka, jak budou jednotlivé algoritmy vůči sobě synchronizovány. Synchronizace algoritmů tak, aby všechny algoritmy měly v určitý čas např. zpracován stejný počet bitů na vstupu nebude možná, jelikož je zcela zřejmé, že vyhledávací procesy jednotlivých algoritmů se budou počty kroků lišit a navíc algoritmy s nastavitelnou střídou mohou číst mnohonásobně více bitů, než např. algoritmus Trie.

4.2.4 Uživatelské volby a nastavení programu

Aby si uživatel mohl přizpůsobit aplikaci k obrazu svému, bude nutná existence globálního nastavení, ve kterém bude možno měnit parametry jako např. změna barev pro vizualizaci vnitřních struktur algoritmů, možnost měnit určité parametry animace, ale i volby související s nastavením algoritmů, jako třeba nastavitelná délka střídy pro algoritmy CPE a Tree Bitmap.

Nastavení bude zabezpečovat třída, která bude obsahovat jak proměnné nesoucí konkrétní informace o aktuálním nastavení, tak i přístupové metody k těmto proměnným. Dále je důležité, aby nastavení bylo možno uložit a zase načíst ze souboru, takže daná třída musí obsahovat i metody zabezpečující tyto akce.

4.2.5 Uživatelské rozhraní

Aby byly všechny důležité informace uživateli zobrazeny co nejpřehledněji, je nutné zauvažovat nad rozmístěním prvků uživatelského rozhraní a nad prvky jako takovými. S návazností na výše uvedené skutečnosti víme, že uživateli musíme zobrazit vnitřní strukturu algoritmů, která bude pravděpodobně vyžadovat spoustu místa pro její vykreslení. Dále musíme uživateli umožnit kontrolovat vyhledávací proces vůči seznamu prefixů, nad kterými vyhledáváme. Ten by měl být zobrazen současně s vnitřní strukturou, protože pokud by mezi těmito dvěma prvky musel uživatel přepínat, nebylo by to ani pohodlné, ale ani přehledné. Dále musí mít uživatel k dispozici nějaký panel s ovládacími prvky, ze kterého bude moci řídit samotné vyhledávání (krokování, zastavení animace a podobně). Tento panel by měl taktéž být viditelný po celou dobu animace vyhledávání, aby mohl uživatel do animace pohotově zasahovat. V neposlední řadě je také nutné nějakým způsobem zadat hledanou adresu, jejíž bitový tvar by měl být během vyhledávání také viditelný, abychom mohli vyznačovat přečtené bity a uživatel mohl vidět souvislost mezi čtením bitů na vstupu a příslušnými akcemi algoritmů.

Všechny tyto prvky uživatelského rozhraní by tedy měly být během vyhledávání zobrazeny současně. Zamyslíme-li se nad účelem aplikace (tedy demonstrací vyhledávání nad algoritmy LPM), dojdeme k závěru, že tyto prvky mohou být zobrazeny permanentně, pouze se bude v různých fázích používání aplikace měnit jejich obsah. Pro ostatní komunikaci s uživatelem můžeme v aplikaci použít dialogy (např. dialog pro úpravu nastavení).

Můžeme si představit, že seznam prefixů bude náročný spíše na vertikální prostor, naopak jeho šířka asi nebude velká. Ovládací panel bude pravděpodobně obsahovat pouze několik tlačítek, takže jeho šířka nebude velká, stejně tak i jeho výška. Bitový tvar adresy zase bude zabírat pouze jeden řádek s třiceti dvěma nulami a jedničkami. Naopak prostor pro zobrazení vnitřních struktur algoritmů (stromů) bude náročný jak na jeho šířku, tak i na jeho výšku. Proto se jeví jako velmi vhodné seskupit seznam prefixů, ovládací panel a bitovou vstupní adresu do jednoho vertikálního panelu, který se bude nacházet na jedné straně, zatímco všechen ostatní prostor okna programu vyplní prostor pro zobrazování vnitřních struktur algoritmů. Tím zaručíme, že budou všechny důležité prvky viditelné současně, a tedy uživateli dopřejeme základní komfort při používání programu.

Kapitola 5

Implementace programu

V této kapitole se budeme věnovat implementaci výsledné aplikace pro vizualizaci algoritmů LPM. Popíšeme si konkrétní implementaci výsledné aplikace včetně řešení problémů, které s ní byly spojeny.

5.1 Použité technologie

Jelikož jsme si ve fázi návrhu vyjasnili jakou funkčnost chceme v aplikaci implementovat a jak ji chceme implementovat, můžeme přistoupit k výběru vhodných nástrojů pro její implementaci. Vzhledem k nutnosti aplikaci implementovat v grafickém režimu s nejrůznějšími dialogy apod., bylo třeba najít nástroj, který podporuje implementaci těchto náležitostí. Proto jsem se rozhodl výslednou aplikaci implementovat v jazyce C++ s využitím knihovny Qt, která nabízí vše potřebné pro úspěšnou implementaci dříve vytyčených cílů. Dalším důvodem pro tuto volbu byly i mé dřívější zkušenosti s používáním této knihovny.

5.1.1 Knihovna Qt

Qt je framework určený především pro tvorbu aplikací s grafickým uživatelským rozhraním (GUI), ale podporuje i konzolové aplikace. Mimo knihoven pro tvorbu GUI obsahuje i knihovny pro práci se soubory, vlákny, XML, síťové knihovny apod., takže umožňuje rychlou tvorbu rozsáhlých programů.

Qt je multiplatformní, takže kód napsaný v Qt funguje na mnoha platformách (podporované platformy jsou: Windows, Linux/X11, Mac OS X, Symbian, embedded systémy). Velkou výhodou je rozsáhlá dokumentace, která je na kvalitní úrovni a tak velmi ulehčuje používání celé knihovny. Aplikace napsané v Qt lze distribuovat pod open-source licencemi GNU GPL[1] a GNU LGPL[2] nebo pod komerční licencí Qt (ne open-source).

Pro vývoj aplikací s použitím Qt je k dispozici integrované vývojové prostředí *Qt Creator*, které bylo použito i při tvorbě výsledné aplikace a kombinuje v sobě editor kódu, nástroje pro tvorbu GUI, nástroje pro překlad a debugger. Knihovna Qt je vyvíjena od počátku 90. let, verze 1.0 byla vydána v roce 1996. V současné době (leden 2013) je k dispozici verze 5.0, nicméně pro tvorbu aplikace jsem použil Qt verze 4.8.3. Vývoj pak probíhal v prostředí Windows. Informace uvedené v této kapitole byly čerpány z [1, 3].

[1] http://www.gnu.org/licenses/gpl.html
[2] http://www.gnu.org/licenses/lgpl.html

5.1.2 Jazyk C++

Knihovna Qt je primárně určena pro jazyk C++. Jedná se o nástupce jazyka C, který oproti C přináší spoustu pokročilých technik, jako podpora objektově orientovaného programování, podpora šablon, možnost přetěžování operátorů, obsluha výjimek a další. [8] C++ je i proto jedním z nejrozšířenějších a nejpoužívanějších programovacích jazyků[3]. Jazyk jako takový je standardizován organizací ISO, jeho nejnovější standard je ISO/ANSI C++11[4] z roku 2011, který ale pro vývoj aplikace nebyl využit.

5.2 Uživatelské rozhraní

Při implementaci jsem nejdříve začal tvorbou uživatelského rozhraní, a to zejména z toho důvodu, abych měl představu o možnostech prvků, které budou použity pro implementaci jednotlivých akcí.

Pro tvorbu uživatelského rozhraní pod knihovnou Qt lze v zásadě využít dvě možnosti – tvorbu jednotlivých prvků manuálně ve zdrojovém kódu a následné nastavení jejich vlastností nebo lze využít možností *Qt Creatoru* a uživatelské rozhraní interaktivně „poskládat“ z dostupných prvků, a poté nastavit jejich vlastnosti v příslušných kolonkách. Já jsem zvolil druhou možnost, jelikož ušetří spoustu zbytečné práce.

Vznikla tak první třída programu – `MainWindow`, která představuje hlavní okno aplikace a umožňuje přístup ke všem prvkům uživatelského rozhraní. Z tohoto důvodu se v ní nachází zejména metody pracující s jednotlivými prvky GUI. V jejím konstruktoru také probíhá inicializace celé aplikace, takže jsou v něm i vytvářeny objekty pro jednotlivé algoritmy, které jsou následně v destruktoru při ukončení aplikace smazány.

Do hlavního okna jsem nahoru umístil menu (sekce 1 na obr. 5.1), jak jsme na něj zvyklí z ostatních programů (v Qt třída `QMenuBar`). V menu se nachází položky, které nejsou používány úplně často a nemusí tedy být umístěny přímo mezi hlavními ovládacími prvky. Jsou to položky pro ukončení aplikace, pro zobrazení editoru prefixů (stará se o načítání/ukládání seznamu prefixů, viz dále), pro úpravu nastavení (dialog nastavení je popsán dále) a položky sloužící pro nápovědu (Nápověda, O Qt, O Programu).

Hlavní prostor okna jsem rozdělil na dva sloupce (pro pozicování prvků jsem použil *Horizontal Layout*) – v levém sloupci, který zabírá větší plochu, jsou vizualizovány vnitřní struktury algoritmů, v pravém sloupci se poté nachází hlavní ovládací prvky. Popis těchto prvků se nachází v sekcích 5.2.2 a 5.2.3.

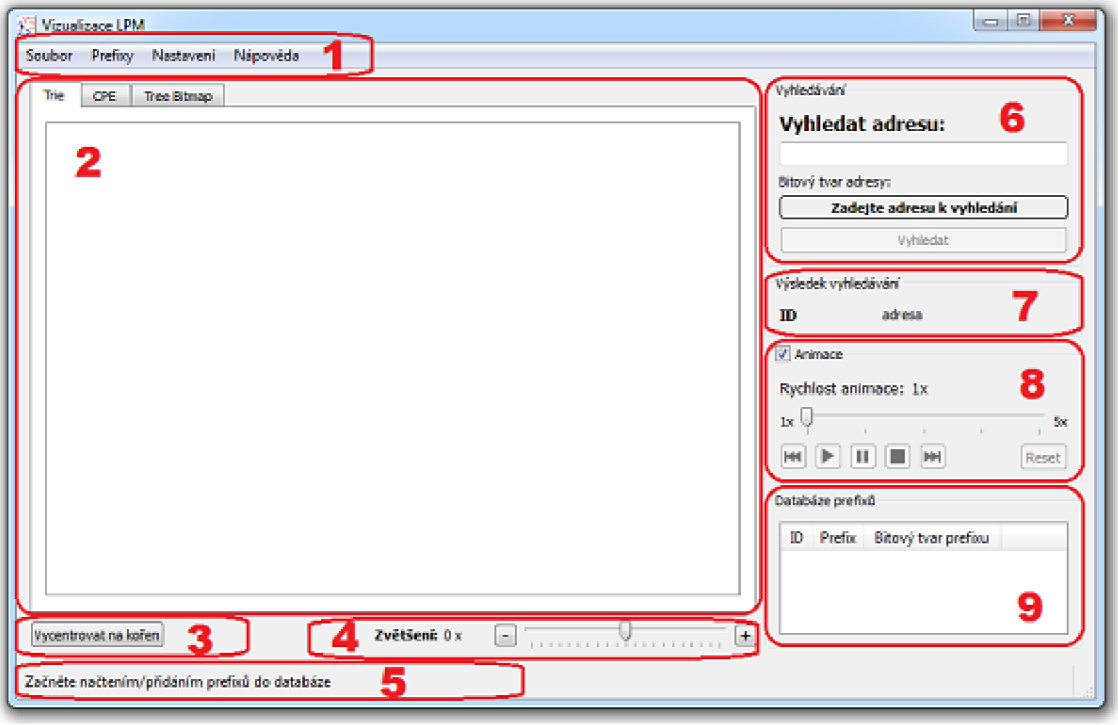
5.2.1 Stavový řádek

V zápatí hlavního okna (sekce 5 na obr. 5.1) se nachází stavový řádek (`QStatusBar`), který slouží k zobrazování jednoduchých informativních sdělení uživateli. Zobrazovány jsou jak rady k užívání programu, tak případné chyby. Během vyhledávání je poté ve stavovém řádku zobrazován komentář k akcím, jež algoritmus aktuálně vykonává.

Jelikož `QStatusBar` přímo neumožňuje zobrazení barevných zpráv či tučného textu, je ve stavovém řádku vložen *Label* (`QLabel`), který tyto možnosti nabízí, a do kterého se tedy samotné zprávy zobrazují. Stavový řádek tedy umožňuje rozlišit jednotlivé druhy zpráv, které zobrazuje – barevně (červeně) jsou zvýrazněny zpravidla chyby nebo velmi důležité zprávy, tučně jsou potom zvýrazněny důležité zprávy.

[3] Dle žebříčku popularity programovacích jazyků, dostupné na URL [cit. 29-04-2013]: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[4] Standard ISO/IEC 14882:2011



Obrázek 5.1: Sekce uživatelského rozhraní.

Jednotlivé zprávy je také možno zobrazit na určitý čas nebo neomezeně dlouho (do přepsání jinou zprávou). K časovanému zobrazení zpráv je použit časovač (`QTimer`). Zprávy jsou ukládány do fronty, takže pokud provedeme více požadavků na zobrazení časovaných zpráv, jsou zprávy zobrazeny v pořadí příchodu požadavku na jejich zobrazení a po stanovenou dobu. Pro zobrazované zprávy existují dva druhy priorit – normální priorita a vysoká priorita, kdy zpráva s vysokou prioritou je zařazena na začátek fronty a předběhne tedy ostatní zprávy (zpráva s vysokou prioritou je zobrazena ihned – nečeká na časované zprávy).

Pro zobrazení zprávy na stavovém řádku slouží metoda `show_statusbar_message()` ve třídě `MainWindow`, jejímiž parametry jsou text zprávy, doba po kterou se má zpráva zobrazit, jestli je zpráva prioritní a jestli se jedná o chybovou zprávu.

5.2.2 Panel ovládacích prvků

Panel sdružující prvky pro ovládání vyhledávání se nachází v pravém sloupci hlavního okna. Pomocí boxů (`QGroupBox`) je rozdělen na určité sekce, které v sobě sdružují dané ovládací prvky.

Úplně nahoře je umístěna sekce *vyhledávání* (sekce 6 na obr. 5.1), která v sobě sdružuje vstupní pole pro zadání hledané adresy (`QInputBox`), dále textový řádek s bitovým tvarem této hledané adresy (`QLabel`) a také tlačítko *Vyhledat* sloužící k zahájení vyhledávání. Hledaná adresa je zadávána ve tvaru normální IPv4 adresy a její správnost (správný tvar a rozsah hodnot) je kontrolována při změně textu v příslušném vstupním poli. Pokud je adresa zadána ve správném tvaru, zobrazí se její bitový tvar. Pokud není, místo bitového tvaru se uživateli zobrazí varování, že adresa má špatný tvar. To uživateli umožňuje okamžitě zkontrolovat, jestli je adresa zadána ve správném tvaru. Během vyhledávání jsou

poté barevně vyznačeny bity, které už byly zpracovány. Tlačítko vyhledat, které spouští vyhledávání, je po spuštění programu neaktivní, tedy na něj uživatel nemůže kliknout. Aby tlačítko přešlo do stavu, kdy je na něj možno kliknout, je třeba splnit dvě podmínky – musí být zadána hledaná adresa ve správném tvaru a také musí být načtena množina prefixů, nad kterými budeme vyhledávat. Takto je jednoduše ošetřeno, aby vyhledávání nebylo spuštěno nad chybnými či žádnými daty.

Pod sekcí vyhledávání se nachází sekce *Výsledek vyhledávání* (sekce 7 na obr. 5.1), která slouží pro zobrazení výsledku vyhledávání. Po dokončení vyhledávání je v ní zobrazen nalezený nejdelší shodný prefix nebo je uživateli sděleno, že vyhledání nebylo úspěšné (v množině prefixů, nad kterými vyhledáváme, nebyl nalezen žádný shodný prefix). Tato sekce je za normálních okolností skryta, tedy uživatel ji během vyhledávání vůbec nevidí. Zobrazena uživateli je až po dokončení vyhledávání, při novém vyhledání je zase skryta. Uživatel tedy vidí výsledek vyhledávání až po dokončení vyhledávání, jelikož během vyhledávání žádný výsledek neexistuje, a tedy není důvod, aby tato sekce zabírala místo v pravém sloupci obrazovky (místo toho poskytneme větší prostor pro sekci *Databáze prefixů*, viz dále.

Dále následuje sekce *Animace* (sekce 8 na obr. 5.1), která v sobě obsahuje veškeré ovládací prvky pro ovládání vyhledávacího procesu. Tento box má nastaven parametr *checkable*, tedy ho lze zaškrtnutím políčka aktivovat či deaktivovat. Pokud si uživatel nepřeje vyhledávací proces animovat po krocích, ale chce vidět rovnou výsledek vyhledávání, jednoduše odškrtne box *Animace* a po stisku tlačítka *Vyhledat* je zobrazen přímo výsledek vyhledávání. Naopak, pokud uživatel chce interaktivní vyhledávací proces, nechá tento box zaškrtnut a poté může řídit vyhledávání pomocí zde se nacházejících ovládacích prvků. Nachází se zde posuvník, kterým lze ovládat rychlost animace, tedy jak rychlá bude změna kroků vyhledávacího procesu. Posuvník nabízí celkem pět stavů (1-krát až 5-krát). Při základní rychlosti 1-krát se kroky střídají každé dvě vteřiny, při rychlosti 5-krát je to tedy 0,4 vteřiny. Tato hodnota je obsažena přímo v kódu jako konstanta (`DEFAULT_TIMER_TICK` v souboru *animator.h*) a s ohledem na plynulost animace je dostatečně velká (pro dlouhé přemýšlení nad jednotlivými kroky je vhodné si proces vyhledávání krokovat manuálně). Dále se zde nachází sada tlačítek, které slouží pro ovládání animace – *krok zpět*, *spustit animaci*, *pozastavit animaci*, *zastavit animaci*, *krok vpřed*, *reset*. Prvních pět tlačítek je opatřeno ikonami po vzoru multimediálních přehrávačů, které vyjadřují jejich význam, tlačítko *reset* je poté zarovnáno doprava a slouží k ukončení vyhledávání (umožní zadat nové parametry vyhledávání). Tlačítka pro ovládání animace jsou v průběhu animace postupně aktivovány a deaktivovány tak, aby uživateli vždy povolily vykonat pouze proveditelnou akci (např. na začátku vyhledávání je deaktivováno tlačítko *krok zpět*). Příkladem budiž tlačítko *přehrát animaci*, jehož stisk deaktivuje tlačítka *krok zpět* a *krok vpřed*. Pokud bychom chtěli využít manuálního krokování, musíme animaci pozastavit a poté již můžeme krokovat manuálně přímo z místa, ve kterém jsme animaci pozastavili. Naopak můžeme třeba prvních pět kroků krokovat manuálně a poté spustit zbytek animace pomocí *přehrát animaci*.

Úplně dole (sekce 9 na obr. 5.1) se poté nachází sekce *Databáze prefixů*, která obsahuje tabulku se seznamem prefixů, nad kterými vyhledáváme. Uživatel tak má stálý přehled a může si ověřovat správnost vyhledávání či podobu vnitřních struktur algoritmů. V této tabulce poté během vyhledávání barevně označujeme prefixy, které v ten moment představují zapamatovaný nejdelší shodný prefix, stejně jako po dokončení vyhledávání v ní zvýrazníme výsledný prefix. Prefixy jsou do tabulky načítány pomocí dialogu *Editor prefixů*, jenž je popsán dále. Editor prefixů lze spustit z horního menu, ale abychom uživateli usnadnili ovládání aplikace, lze editor prefixů vyvolat dvojklikem na tuto tabulku.

5.2.3 Zobrazení vnitřních struktur algoritmů

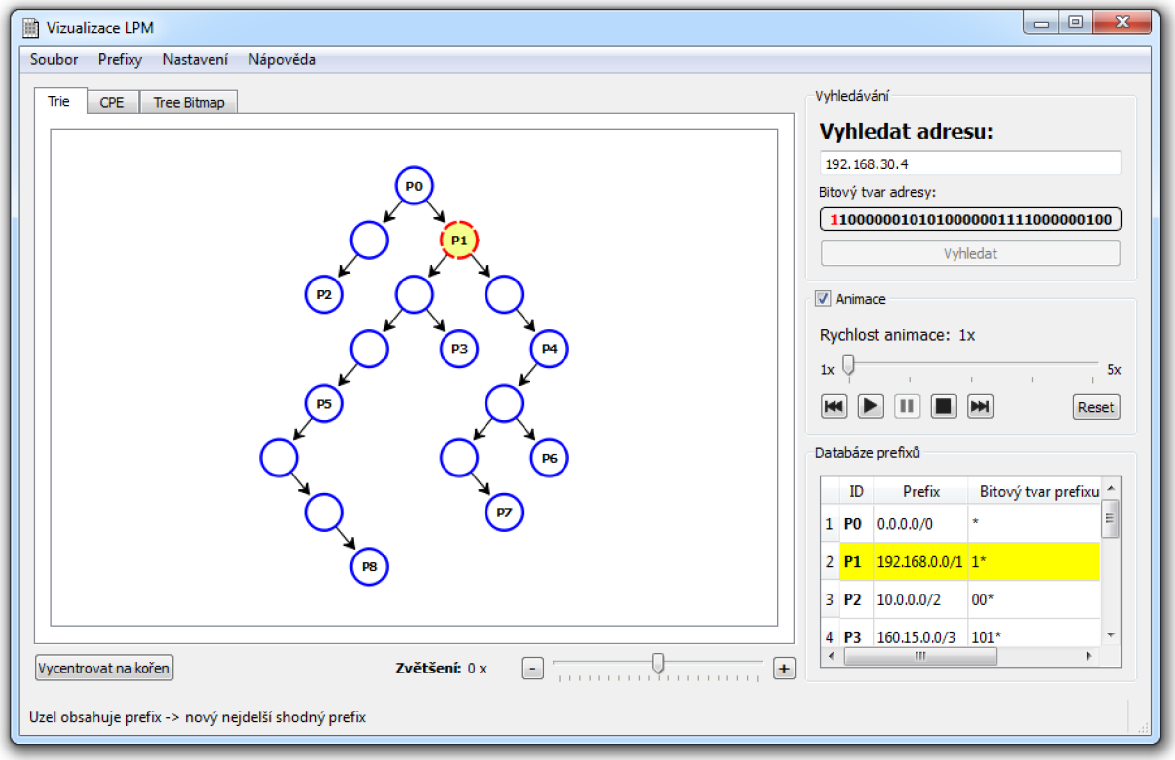
Aby bylo možné pro každý algoritmus vizualizovat jeho vnitřní strukturu, musí být tato struktura graficky vykreslena na nějakou plochu k tomu určenou. V Qt k těmto účelům slouží prvek `QGraphicsView`, do kterého je možno libovolně vykreslovat grafické objekty (grafické objekty jsou vykreslovány do grafické scény – `QGraphicsScene`, kterou je poté teprve možno zobrazit v `QGraphicsView` – vykreslování objektů do scény bude podrobně popsáno v kapitole 5.6).

Vzhledem k návrhu, kdy jsme si řekli, že bude možno mezi algoritmy během animace vyhledávání přepínat, by nebylo praktické při každém přepnutí algoritmu znovu vykreslovat celou scénu daného algoritmu. Proto je nutné, aby pro každý algoritmus existoval `QGraphicsView`, do kterého bude vykreslena jeho vnitřní struktura a během přepínání algoritmů se uživateli zobrazí daný `QGraphicsView`. Pro přepínání algoritmů jsem proto využil záložky (`QTabWidget`), kdy má každý algoritmus svoji záložku, jejímž obsahem je `QGraphicsView`, obsahující vizualizaci jeho struktury (sekce 2 na obr. 5.1). Uživatel tak může překlikávat mezi algoritmy a vždy se mu zobrazí relevantní data, aniž by se něco muselo znovu překreslovat. Také je takto možné vizualizovat vyhledávací kroky i u algoritmů které nejsou aktivní, tedy jejichž struktura není aktuálně viditelná. Během přepnutí záložky lze také detekovat změnu aktuálního algoritmu a následně provést nutné akce. Detekce změny algoritmu a vykonání příslušných akcí je implementováno v metodě `tab_algorithm_changed()` ve třídě `MainWindow`, která také nabízí prostředky ke zjištění aktivního algoritmu (metoda `get_active_algorithm()`).

Pokud budeme vizualizovat vnitřní struktury algoritmů při použití velké množiny prefixů, je jasné že vytvořené stromy budou velmi prostorově náročné. Proto je třeba uživateli umožnit měnit velikost vykreslené scény nebo měnit velikost vykreslených objektů. Druhá možnost se jeví jako značně lepší nápad. Naštěstí Qt nad grafickými scénami umožňuje provádět transformace, tedy i měnit jejich měřítko (metoda `scale()` u třídy `QGraphicsView`). Objekty scény tak lze zvětšit či zmenšit. K tomu slouží posuvník, který je umístěn napravo pod záložkami s algoritmy (sekce 4 na obr. 5.1). Posuvník umožňuje pohyb na obě strany (výchozí poloha – zvětšení 0-krát se nachází uprostřed rozsahu posuvníku) a umožňuje scénu zvětšit (posun doprava) či zmenšit (posun doleva) až 5-krát, a to s krokem 0,5-krát (celkem 10 kroků na každou stranu od středu). Uživatel si tak může scénu přiblížit nebo oddálit dle svých preferencí. Tyto akce se vždy vztahují pouze k aktivní scéně, tedy nijak nemanipuluje s ostatními scénami.

Pohyb po zvětšené scéně je uživateli umožněn pomocí posuvníků, které se zobrazí, pokud je scéna větší než fyzický prostor, který má k dispozici. Aby bylo ovládání jednodušší a uživatel nemusel posuvníky používat, je také umožněno použít *mouse drag*, tedy podržet levé tlačítko myši a hýbat scénou myší.

Nalevo od posuvníku (sekce 3 na obr. 5.1) pro změnu velikosti scény se pak nachází tlačítko *Vycentrovat na kořen*, které scénu posune tak, aby uživatel viděl kořenový uzel daného stromu. To uživateli ušetří hledání kořene stromu při značně přiblížené scéně.



Obrázek 5.2: Uživatelské rozhraní aplikace s ukázkou spuštěného vyhledávacího procesu.

5.2.4 Klávesové zkratky

Aby bylo užívání aplikace pro uživatele co nejpříjemnější, implementoval jsem pro většinu akcí klávesové zkratky, takže lze program ovládat takřka pouze za pomoci klávesnice. Seznam zkratek je uveden v následující tabulce.

Akce	**Klávesová zkratka**
Zvětšení plátna (přiblížení)	+
Zmenšení plátna (oddálení)	-
Vycentrovat na kořen	Ctrl+C
Přehrát/pozastavit animaci	Space (mezerník)
Předchozí krok animace	Šipka doleva
Další krok animace	Šipka doprava
Stop animace	Ctrl+X
Vyhledat	Ctrl+S
Upravit nastavení	Ctrl+P
Spustit editor prefixů	Ctrl+L
Ukončení programu	Ctrl+Q
Nápověda	F1

Tabulka 5.1: Klávesové zkratky pro ovládání programu.

5.3 Nastavení programu

Aby si mohl uživatel přizpůsobit aplikaci dle svých preferencí, existuje v programu nastavení. Informace týkající se konkrétního nastavení v sobě nese třída `preferences`. Ta obsahuje privátní proměnné, které v sobě udržují informace o konkrétním nastavení, ale také přístupové metody k těmto proměnným (sada metod *get* – získání hodnoty a *set* – nastavení hodnoty pro každou proměnnou).

Aby bylo možno nastavení ukládat a načítat, existují v třídě `preferences` také metody pro uložení a načtení celého nastavení do souboru (`save()` a `load()`). Konfigurační soubor se jmenuje *config.xml* (konstanta v souboru *preferences.h*) a standardně se nachází v adresáři s aplikací. Jedná se o XML soubor, kde jsou jednotlivá nastavení uložena jako atributy příslušných tagů.

Standardně je objekt představující globální nastavení programu vytvořen a inicializován v konstruktoru třídy `MainWindow`. Bezprostředně po vytvoření objektu jsou jednotlivé položky nastavení inicializovány načtením nastavení z konfiguračního souboru. Pokud konfigurační soubor neexistuje nebo pokud je jeho struktura chybná, je načteno základní nastavení programu (metoda `load_default()` ve třídě `preferences`), které je poté ihned uloženo do konfiguračního souboru tak, aby příště proběhlo načtení nastavení bez problému.

Zobrazení a úpravu jednotlivých položek nastavení umožňuje dialogové okno určené k tomuto účelu (třída `preferences_dialog`). Jeho zobrazení je možno provést z horního menu hlavního okna nebo pomocí klávesové zkratky (Ctrl+P). Položky nastavení jsou zde organizovány do dvou záložek. V první záložce lze upravit barvy a atributy písma (*font*), kterými vykreslujeme vnitřní struktury algoritmů, ve druhé záložce se pak nachází nastavení související s animací a algoritmy (u algoritmů lze například nastavit použitou délku střídy). Abychom mohli obnovit původní nastavení, pokud by si uživatel změnu nastavení rozmyslel, je ve třídě `preferences_dialog` vytvořen pomocný objekt nastavení (instance třídy `preferences`), který je inicializován hodnotami globálního nastavení. Veškeré změny nastavení se poté aplikují na pomocný objekt, a až pokud uživatel změnu nastavení potvrdí stiskem tlačítka *Použít* jsou hodnoty nového nastavení zkopírovány do globálního nastavení, které je také ihned uloženo do konfiguračního souboru. Během tohoto procesu jsou také detekovány změny klíčových položek nastavení týkající se algoritmů (např. délka střídy), po jejichž změně je nutné znovu vykreslit vnitřní strukturu daného algoritmu nebo provést další akce. Tyto akce jsou poté provedeny bezprostředně po potvrzení nového nastavení.

5.4 Editor prefixů

Pro vložení sady prefixů, nad kterou budeme později vyhledávat, slouží *editor prefixů*. Jedná se o dialogové okno, ve kterém je tabulka, ukazující seznam těchto prefixů. K dispozici jsou akce pro správu prefixů, jako načtení ze souboru či uložení do souboru.

5.4.1 Tvar prefixů a jejich reprezentace

Prefixy samotné jsou vkládány jako textové řetězce v podobě IP adres v prefixovém tvaru (`xxx.xxx.xxx.xxx/xx`), kde část oddělená lomítkem určuje počet použitých bitů z dané adresy. Do tohoto bitového tvaru je na konec přidán znak `*` vyjadřující, že další bity mohou mít libovolnou hodnotu. V tabulce jsou pak tyto prefixy rozděleny do sloupců *adresa* (celá IP adresa), *prefix* (počet použitých bitů z této adresy) a *bitový tvar* (vlastní bitový tvar ukončený znakem `*`). Aby bylo možno prefixy jednotně identifikovat, je každému prefixu

přiřazen unikátní název skládající se z písmena *P* a čísla řádku tabulky, na kterém se daný prefix nachází. Tento atribut je pak v tabulce označen jako *ID* a tvoří tak další sloupec.

Každý vloženy prefix je pak reprezentován strukturou `prefix_item`, která v sobě obsahuje položky odpovídající sloupcům v tabulce, tedy *adresa*, *prefix*, *bitovy_tvar* a *id*. Z těchto struktur je po zavření editoru prefixů vytvořen lineární seznam, jehož položky přesně odpovídají řádkům v tabulce.

5.4.2 Třída `prefix_editor`

Samotný editor prefixů představuje třída `prefix_editor`, která obstarává jak uživatelské rozhraní editoru prefixů (dědí od třídy `QDialog`), tak i další metody pro práci s prefixy. Třída v sobě také udržuje výše zmíněný seznam prefixů a metody pro přístup k němu. Mimo seznamu prefixů v podobě struktur také obsahuje metodu pro vrácení seznamu prefixů pouze v bitovém tvaru, který se využívá při tvorbě vnitřních struktur algoritmů.

Instance třídy `prefix_editor` je vytvořena v konstruktoru třídy `MainWindow` a je posléze využívána po celou dobu běhu programu tak, aby algoritmy mohly využívat metod pro získání prefixů (trvale v sobě udržuje aktuální seznam prefixů).

Jakmile dokončíme editaci prefixů v editoru prefixů, je aktuální seznam prefixů také zobrazen v hlavním okně aplikace v sekci *Databáze prefixů*.

5.4.3 Akce v editoru prefixů

V editoru prefixů lze prefixy přidávat jednotlivě (pomocí vstupního okna – tlačítko *Přidat prefix*) nebo lze prefixy načíst ze souboru. K tomuto slouží tlačítka *Načíst* a *Přidat ze souboru*. Rozdíl mezi nimi je, že *Načíst* smaže existující prefixy, kdežto *Přidat ze souboru* nikoliv. Při přidávání prefixů je kontrolována jejich unikátnost, aby byly vyloučeny duplicitní prefixy (unikátnost je kontrolována na základě bitového tvaru). Po přidání prefixů jsou prefixy seřazeny podle bitového tvaru od nejkratšího po nejdelší, ale zároveň i podle jejich hodnoty (*00** bude před *11**). Tohoto chování je docíleno pomocí dvojitého řazení, kdy řadíme nejprve podle hodnoty a následně až podle délky.

Seznam prefixů lze také uložit do souboru (tlačítko *Uložit*). Prefixy jsou ukládány v textové podobě IP adres v prefixovém tvaru, každý prefix na samostatném řádku souboru. K tvorbě seznamu prefixů tak lze použít libovolný textový editor.

Prefixy lze samozřejmě mazat (uživatel musí v tabulce označit řádky ke smazání myší) a upravovat (dvojklikem na řádek v tabulce). Volba *Smazat vše* pak vymaže veškeré prefixy. Pro pohodlnost pak lze v editoru prefixů použít následující klávesové zkratky:

Akce	**Klávesová zkratka**
Načíst ze souboru	Ctrl+O
Uložit do souboru	Ctrl+S
Přidat ze souboru	Ctrl+F
Přidat prefix	Ctrl+P
Smazat prefix	Delete
Smazat všechny prefixy	Ctrl+Delete
Hotovo/zavřít	Ctrl+D

Tabulka 5.2: Klávesové zkratky pro ovládání editoru prefixů.

5.5 Algoritmy a související třídy

K vizualizaci každého algoritmu je potřeba vytvořit několik odvozených tříd a v několika existujících třídách doplnit jejich metody. Nyní si stručně popíšeme jaké třídy v programu existují a k čemu obecně slouží. Popis konkrétních implementačních problému je předmětem dalších kapitol.

Základní abstraktní třídou, od které jsou odvozeny třídy ostatních algoritmů, je třída `abstract_algorithm`. Ta v sobě obsahuje deklarace virtuálních metod, které je třeba pro každý algoritmus reimplementovat a také definice několika metod, které jsou obecně použitelné pro libovolný algoritmus. Pro každý algoritmus je poté vytvořena nová třída, která dědí od třídy `abstract_algorithm` a redefinuje její virtuální metody, ale definuje i své interní metody. V programu jsou to třídy `visualization_trie`, `visualization_tree_bitmap` a `visualization_cpe`. Tyto třídy v sobě obsahují převážně metody pro tvorbu jejich vnitřních struktur a metody pro animaci vyhledávacího procesu daného algoritmu.

Aby bylo možno vizualizovat vnitřní struktury algoritmů, v našem případě stromy, musí pro každý algoritmus existovat třída představující grafické znázornění uzlu daného algoritmu. Podobně jako u algoritmů existuje základní abstraktní třída `node`, od které jsou poté odvozeny třídy pro uzly konkrétních algoritmů. Ta v sobě obsahuje několik virtuálních metod k redefinici, ale také definice všeobecně použitelných metod. Důležitou vlastností třídy `node` je, že dědí od třídy `QGraphicsWidget`, což znamená že instance odvozených tříd jsou grafickými objekty, které můžeme v Qt vykreslit do grafické scény. Odvozené třídy (`node_trie`, `node_tree_bitmap` a `node_cpe`) poté představují uzly konkrétních algoritmů.

Třídou, která řídí animaci vyhledávání pro všechny algoritmy, je poté třída `animator`.

5.5.1 Třída `algorithms`

Aby byla usnadněna implementace dalších algoritmů, byla vytvořena třída `algorithms`, která v sobě seskupuje metody pro práci se všemi algoritmy. Jedná se o metody, které musí být volány pro všechny existující algoritmy, a proto je výhodné tato volání seskupit do jediné metody, která zařídí správná volání v příslušných algoritmech. Po přidání nového algoritmu je poté nutné doplnit tato volání v metodách třídy `algorithms`, ale už nikde jinde (až na speciální případy). Ušetříme si tak spoustu práce s lokalizací dalších změn.

`Algorithms` je také centrálním bodem pro tvorbu či smazání všech algoritmů. Proto když v konstruktoru třídy `MainWindow` chceme vytvořit instance algoritmů, voláme pouze metodu `create_algorithms()` třídy `algorithms`, čímž také přispíváme k přehlednosti kódu.

Zároveň jsou v `algorithms` pro každý algoritmus vytvořeny proměnné uchovávající stav sdíleného uživatelského rozhraní. To je nutné kvůli možnosti přepínání algoritmů během animace, kdy algoritmy mají označeny různé prefixy, zpracovány různý počet bitů vstupu apod. Proto musí mít každý algoritmus uložen stav těchto prvků tak, abychom ho při přepnutí na jiný algoritmus a zpět mohli zase obnovit.

5.6 Tvorba a vizualizace vnitřních struktur algoritmů

Jak již bylo naznačeno, pro každý algoritmus musí být vytvořena třída, která představuje jeho uzel. Nejprve je třeba dát uzlu nějaký tvar, což souvisí s vnitřním uspořádáním uzlu, tedy s tím, co všechno chceme zobrazovat jako obsah uzlu. Například u Trie je uzel kulatý, naopak u CPE je uzel obdélníkového tvaru. Samotný tvar uzlu je definován v metodě

`paint()` příslušné třídy, což je virtuální metoda třídy `QGraphicsWidget` a stará se o vykreslení objektu. Zde definujeme jak tvar objektu, tak jeho velikost (může být fixní nebo se přizpůsobit velikosti vložených objektů) a barvu (barvy bereme z nastavení programu).

Do objektu, představujícího uzel, lze také vložit další objekty, určené pro zobrazení vnitřní struktury uzlu. Toho je využito hlavně u uzlů CPE a Tree Bitmap, které obsahují *Labely*, do kterých můžeme umístit text (v našem případě je využíváme pro znázornění prefixů). Rozmístění těchto objektů uvnitř uzlu je možné zařídit pomocí tzv. *Layoutu*, kde můžeme objekty rozmístit do mřížky, řádků či sloupců.

Jakmile je vyřešen vzhled uzlu, je třeba ještě implementovat metody pro práci s daným uzlem, tedy metody pro vkládání prefixů, ukazatelů, metody pro přístup k prefixům apod. Tyto metody jsou samozřejmě závislé na typu uzlu a pro každý algoritmus se velmi liší.

Aby bylo uzly možno během animace zvýrazňovat, má každý uzel tři možnosti zvýraznění: *normální* (uzel je vykreslen normálními barvami), *zvýrazněný* (uzel je vykreslen jinými barvami) a *výsledný uzel* (uzel je výsledkem vyhledávání – barevně zvýrazněn). Implementace je jednoduchá – stačí změnit vykreslovací barvu v metodě `paint()`. Všechny barvy lze nastavit v nastavení programu.

Samotné metody, které slouží ke konstrukci vnitřních struktur se nacházejí v třídách představujících jednotlivé algoritmy (jsou redefinicí virtuální metody `construct_graph()`, která je původně definována ve třídě `abstract_algorithm`) a ke konstrukci využívají seznam prefixů získaný z třídy `prefix_editor`.

5.6.1 Třída `arrow`

Protože jsou ve skutečnosti jednotlivé uzly stromů spojeny pomocí ukazatelů, existuje i v programu grafická reprezentace ukazatele, tedy šipka, kterou představuje třída `arrow`. Šipka spojuje dva objekty, přičemž koncovým objektem je vždy uzel. Počátečním objektem může být jak uzel, tak v případě CPE a Tree Bitmap *Label*, který se nachází v poli ukazatelů daného uzlu.

Šipka samotná je složena ze dvou objektů – čáry a koncového polygonu, který má tvar šipky. Koncový polygon musí být vůči čáře a cílovému objektu vhodně natočen, aby výsledný tvar opravdu vypadal jako šipka. Proto jsem se při implementaci tohoto problému inspiroval ukázkovým příkladem *Diagram scene example* [2], který je součástí *Qt Examples* – tedy balíku příkladů demonstrujících použití Qt, a který obsahoval šipky pro propojování objektů a uvedenou situaci řešil.

Šipku lze, podobně jako uzly, barevně zvýraznit, což se během animace používá pro zvýraznění ukazatele při přechodu do dalšího uzlu.

5.6.2 Grafická scéna

Všechny grafické objekty musí v Qt být před zobrazením v `QGraphicsView` vykresleny do grafické scény (`QGraphicsScene`). Proto v programu existuje třída `graphics_scene`, která dědí od `QGraphicsScene` a implementuje metody pro vkládání uzlů jednotlivých algoritmů a také šipek, které je propojují.

V našem případě jsou tyto metody vytvořeny pro každý typ uzlu zvlášť, protože uzly CPE a Tree Bitmap vyžadují po jejich vytvoření ještě volání metod pro nastavení vnitřního *Layoutu* (nelze je volat v jejich konstruktoru). Parametrem těchto metod je pozice, na kterou uzel vkládáme.

Ve `graphics_scene` si také vytváříme seznamy uzlů jednotlivých typů, které poté můžeme využít např. když uživatel v nastavení změní *font*, který je použit pro popisky v uzlech.

Po takové změně je třeba ve všech uzlech změnit *font* těchto popisků, což někdy může vést i ke změně velikosti těchto uzlů.

Další funkcí `graphics_scene` je vycentrování scény nad nějakým objektem (uzlem či ukazatelem – centrujeme na jeho střed). To je využíváno během animace, kdy je na každý zvýrazněný uzel zároveň i centrována scéna, aby uživatel nemusel uzly hledat ručně. V programu jsou implementovány dvě varianty tohoto centrování, a to rychlé, kdy je scéna okamžitě vycentrována na cílový objekt nebo animované, kdy je centrování prováděno plynulou změnou z posledního centrovaného bodu nebo středu scény na cílový objekt. Který druh animace bude použit, lze zvolit v nastavení programu.

Animace je prováděna rozložením celkové změny pozice do padesáti kroků (konstanta v *graphics_scene.h*), čímž získáme hodnotu posunutí v osách x a y pro jeden krok. Celá animace centrování z A do B poté trvá 1/4 času (konstanta v *animator.h*) animačního kroku (používán pro krokování vyhledávání). Pro inteligentní rozložení kroků do tohoto času je použit nástroj `QTimeLine`, který jednotlivé kroky vhodně rozloží do daného časového intervalu, takže animace ve skutečnosti neprovede padesát změn pozice, ale méně.

5.6.3 Uzel Trie

Uzel algoritmu Trie je definován ve třídě `node_trie`. Uzel má kulatý tvar a lze do něj umístit jeden prefix, jehož *ID* bude zobrazeno ve středu uzlu. Každý uzel může mít dva potomky – pravý a levý. V třídě tedy existují metody pro přidání potomků, dále k získání ukazatelů na jednotlivé potomky a také k získání a nastavení ukazatele na rodičovský uzel. Pro nastavení a získání prefixu z uzlu slouží metody `set_prefix()` a `get_prefix()`.

5.6.4 Uzel CPE

Uzel CPE je definován ve třídě `node_cpe`. Uzel má obdélníkový tvar a lze do něj umístit 2^n prefixů, kde n je délka střídy. Uzel také může mít až 2^n potomků. Délku střídy je možné nastavit v nastavení programu (po její změně musí být celý strom CPE rekonstruován).

V nastavení lze také zvolit, zdali má být použita optimalizace *leaf pushing*. Pokud tato optimalizace není použita, má uzel podobu tří řad *Labelů*, kdy horní řada představuje popisky s bitovou hodnotou jednotlivých sloupců, prostřední řada představuje pole s prefixy, kde jsou znázorněny uložené prefixy a spodní řada představuje pole ukazatelů na potomky, odkud jsou následně mapovány šipky na jednotlivé potomky. Pokud je optimalizace použita, má uzel pouze dvě řady *Labelů*, kdy spodní řada v sobě kombinuje pole pro uložení prefixů i ukazatelů.

Třída tak obsahuje metody pro přístup k n-té položce jednotlivých polí (prefixy, ukazatele), kdy n odpovídá indexu vytvořenému ze vstupních bitů. Uzel si také uchovává ukazatele na jednotlivé *Labely*, které je poté možno barevně zvýrazňovat, podobně jako celý uzel.

5.6.5 Uzel Tree Bitmap

Uzel Tree Bitmap je definován ve třídě `node_tree_bitmap`. Uzel má obdélníkový tvar a lze do něj umístit až $2^n - 1$ prefixů a může mít až 2^n potomků, kde n je délka střídy, kterou lze nastavit v nastavení (po její změně je třeba celý strom rekonstruovat).

V uzlu jsou poté zobrazeny jednotlivé *bitmapy* (interní, externí) a pole *Labelů* představující pole pro ukazatele či prefixy. Podobně jako u CPE lze jednotlivé *Labely* barevně zvýraznit.

Přestože jsou v reálné implementaci algoritmu Tree Bitmap v uzlu uloženy pouze ukazatele na tato pole, my tato pole vizualizujeme přímo v uzlu, protože jinak bychom tato pole museli zobrazovat někde bokem, což by nebylo moc přehledné. Nicméně princip zůstal zachován, takže se položky v polích neindexují přímo indexem ze vstupních bitů, ale je třeba spočítat jedničky v příslušné bitmapě a jejich počet poté použít jako index do příslušného pole. Proto podobně jako u CPE třída obsahuje metody pro práci s n-tou položkou (prefix, ukazatel), kde n je ovšem index do příslušné bitmapy, který je poté nutno přepočítat na reálný index do konkrétního pole.

Abychom uživateli maximálně usnadnili pochopení algoritmu Tree Bitmap, lze u každého uzlu Tree Bitmap po najetí myši nad něj zobrazit podobu stromu Trie, který tento uzel reprezentuje. Každý uzel si tak při vkládání prefixů na dané pozice zároveň konstruuje vnitřní strom Trie, který tomuto odpovídá. Každý uzel Tree Bitmap tak zároveň obsahuje další instanci grafické scény, do které vytváří strom Trie, a také objekt `QGraphicsView` pro její zobrazení. Tyto objekty existují permanentně, při najetí myší nad daný uzel je pouze tento `QGraphicsView` uživateli zobrazen metodou `setVisible(true)`, při opuštění myši zase schován metodou `setVisible(false)`.

Pro konstrukci stromu Trie jsou využity standardní uzly definované třídou `node_trie`, pouze je celá scéna zmenšena, aby nezabírala příliš místa. V případě použití velkých střídy a masivního počtu prefixů v uzlech tak ovšem nelze zaručit dobrou viditelnost takového stromu, proto je doporučeno používat spíše menší velikosti střídy.

5.6.6 Pozicování uzlů ve stromu

Jakmile jsou vytvořeny všechny uzly stromu, nastává otázka jejich napozicování tak, aby výsledný strom jednak dodržoval zvyklosti pro kreslení stromů (levý potomek se nachází na levé straně apod.), ale hlavně, aby výsledný strom zabíral co nejmenší plochu. Pozicování je standardně vhodné provádět až po sestrojení celého stromu, protože průběžné pozicování během jeho tvorby by značně prodloužilo dobu sestrojení celého stromu.

To se nakonec ukázalo jako docela velký problém. Zatímco vertikální rozmístění uzlů není problémem, kdy stačí každou úroveň stromu odsadit od té předchozí (získáme pozici rodičovského uzlu a v ose y k ní přičteme několikanásobek výšky uzlu), tak skutečný problém se skrývá v horizontálním rozmístění uzlů (tedy přiřazení pozice v ose x). V tomto případě nelze jednoduše přičíst či odečíst nějakou hodnotu, protože musíme brát v úvahu překrývání jednotlivých podstromů.

Původně jsem pro Trie sestrojil *naivní algoritmus*, který každému uzlu přiřadil jeho index zleva v rámci dané úrovně, který poté sloužil k určení posunutí daného uzlu vůči kořeni (u uzlů levého podstromu bylo posunutí záporné, u pravého podstromu zase kladné). Aby toto fungovalo, musel algoritmus počítat i s neexistujícími uzly, pro které bylo vynecháno místo. Algoritmus se tedy choval, jakoby ve stromu existovaly všechny uzly na všech úrovních, ale rozmísťoval pouze existující uzly. Tím vznikaly prázdná místa a celý strom rostl do šířky. To vedlo k tomu, že při použití velkého množství úrovní (např. 15) byl celý strom extrémně široký a ani při maximálním zvětšení scény nebyly uzly takřka viditelné.

Proto jsem po následném pátrání po řešení tohoto problému objevil několik algoritmů, které jsou určeny pro pozicování uzlů ve stromech. Pro implementaci jsem zvolil algoritmus publikovaný v článku *Tidier Drawnings of Trees* [9], také označovaný jako **Reingold-Tilfordův algoritmus** po jeho autorech. Tento algoritmus je určen pro binární stromy a pracuje na principu rekurzivního přibližování podstromů až na určitou minimální vzdálenost. Tím zajistí minimální možnou šířku výsledného stromu při dodržení zásad pro vykres-

lování stromů. Aby bylo možno algoritmus použít, musely být ve třídě `node_trie` přidány proměnné a metody nutné k běhu tohoto algoritmu. Algoritmus samotný je implementován v souborech *node_positioning.c* a *node_positioning.h* a při jeho implementaci jsem se inspiroval kódem uvedeným v článku [9].

Jelikož je algoritmus *Reingold-Tilford* určen pouze pro binární stromy, nebylo možné jej použít pro CPE ani Tree Bitmap. Proto bylo potřeba pro n-ární stromy implementovat jiný algoritmus. Poučen z předchozích chyb u Trie jsem implementoval *jednoduchý algoritmus*, který pracuje na jiném principu. Algoritmus na každé úrovni stromu spočítá počet existujících uzlů, pro které poté vypočítá celkovou šířku včetně mezer mezi uzly. Poté celou tuto úroveň (všechny uzly) zarovná na střed kořenového uzlu, tedy pravá i levá část od středu je stejně velká. Tím vznikne strom, kdy jsou na každé úrovni všechny uzly seskupeny k sobě a každá z těchto úrovní je zarovnána na střed kořenového uzlu. Velikost stromu je poté malá, ale rozmístění uzlů porušuje zásady kreslení stromů jako např. že nejpravější potomek nějakého uzlu se může nacházet na jeho levé straně a podobně.

Obrázek 5.3: Rozdíl mezi *jednoduchým* (dole) a *Walkerovým* algoritmem (nahoře).

Aby tedy bylo možné pěkně napozicovat i n-ární stromy, implementoval jsem poziční algoritmus [15], jenž je podle jeho autora také nazýván **Walkerův algoritmus**. Ten vychází z algoritmu *Reingold-Tilford*, ale upravuje jej pro použití v n-árních stromech. Algorit-

mus samotný provádí celkem dva průchody stromem, kdy v prvním průchodu *postorder* jsou každému uzlu přiřazeny dočasné souřadnice a ve druhém průchodu *preorder* jsou pak uzlům přiřazeny konečné souřadnice v ose x. Algoritmus je implementován v souborech *node_positioning.c* a *node_positioning.h*. Při jeho implementaci jsem se inspiroval jak kódem uvedeným v článku [15], tak vzorovou implementací z článku [16]. Aby jej bylo možné úspěšně použít s algoritmy CPE a Tree Bitmap, musely v jejich třídách být reimplementovány virtuální metody, jejichž původní definice se nachází v abstraktní třídě `node` a jsou využívány právě *Walkerovým algoritmem*.

Jelikož může být běh *Walkerova algoritmu* pro velké stromy zdlouhavý, může si uživatel v nastavení zvolit, zda pro rozmístění uzlů n-árních stromů použít *Walkerův* nebo *jednoduchý* algoritmus. Rozdíl mezi těmito algoritmy je znázorněn na obrázku 5.3.

Jakmile jsou uzly ve stromech správně napozicovány, poslední akcí je použití metody `fitInView()` třídy `QGraphicsView`, která rozměry scény upraví tak, aby byly všechny její objekty viditelné v příslušném `QGraphicsView`. Uživatel tak vidí celý strom, aniž by musel se scénou nějak hýbat a následně si může scénu dle libosti přiblížit či oddálit.

5.7 Animace vyhledávacího procesu

Jakmile jsou vykresleny vnitřní struktury algoritmů, můžeme nad nimi zahájit vyhledávání. Jak již bylo naznačeno v sekci 5.2.2, vyhledávání lze provést dvěma způsoby, a to *animované*, kdy můžeme interaktivně řídit vizualizaci vyhledávacího procesu, a *okamžité*, kdy je přímo zobrazen výsledek vyhledávání.

Během vyhledávání je z jasných důvodů zakázáno měnit nastavení algoritmů (délky stříd, *leaf pushing* u CPE) a také provádět změny v editoru prefixů.

5.7.1 Řízení animace

Řízení vyhledávacího procesu má na starosti třída `animator`. Ta zařizuje komunikaci s jednotlivými algoritmy na jedné straně a uživatelským prostředím a třídou `MainWindow` na straně druhé.

Jejím základem jsou metody pro řízení animace, které následně vyvolají provedení příslušných kroků samotnými algoritmy. Tyto metody kontrolují možnost provedení těchto kroků a tak detekují např. konec vyhledávání nebo že nelze udělat krok zpět. Pro každý algoritmus jsou zde definovány metody `step_next_algoritmus()` a `step_prev_algoritmus()`, které toto zabezpečují. Pro okamžité vyhledání pak zde existuje metoda `finish_all()`, která provede okamžité vyhledání pro všechny algoritmy. Animační akce *přehrát* pak spočívá v použití časovače pro opakované spouštění metod pro vyvolání dalšího kroku až do dokončení vyhledávání nebo zastavení animace uživatelem. Perioda časovače je dána nastavením rychlosti animace na posuvníku v hlavním okně.

Algoritmům také poskytuje podpůrné metody pro komunikaci s prvky uživatelského rozhraní a umožňuje jim třeba zvýrazňovat řádky v *tabulce prefixů* nebo zobrazovat zprávy ve *stavovém panelu* hlavního okna.

Jelikož může každému algoritmu vyhledávání trvat různý počet kroků, existuje pro každý algoritmus proměnná udržující počet kroků, které algoritmus provedl. Podle této proměnné pak můžeme kontrolovat, zdali je možné udělat krok zpět či nikoliv. Hlavním důvodem její existence je ale uplatnění v situaci, kdy chceme z dokončeného vyhledávání udělat krok zpět. Abychom přesně reflektovali stav jednotlivých algoritmů, musíme je vrátit

do pozice ve které byly, když aktivnímu (zobrazenému) algoritmu chyběl do konce vyhledávání jeden krok. Pokud totiž aktivní algoritmus dokončí vyhledávání, je u všech ostatních algoritmů vyhledávání také dokončeno. Tím může vzniknout rozdíl v počtu kroků mezi jednotlivými algoritmy, kdy aktivní algoritmus může vyhledání dokončit třeba v deseti krocích, kdežto jiný algoritmus ve dvanácti krocích. Když tedy poté u aktivního algoritmu vyvoláme *krok zpět*, musíme u druhého algoritmu udělat rovnou tři kroky zpět, abychom při případném přepnutí algoritmu přesně reflektovali stav v dané době. Naopak, pokud bude mít aktivní algoritmus na kontě více kroků než ostatní algoritmy, akce *krok zpět* bude prováděna pouze v aktivním algoritmu, a to do té doby, než se počty kroků algoritmů srovnají.

Tímto je tedy zajištěna základní synchronizace mezi jednotlivými algoritmy. Vyspělejší synchronizace např. podle počtu bitů přečtených ze vstupu není možná, protože při použití variantních stříd u jednotlivých algoritmů mohou být rozdíly mezi algoritmy příliš velké. Pro názornost mají také jednotlivé algoritmy implementovány různý počet kroků mezi přesunem do určitého uzlu a přechodem na jeho potomka, takže i v tomto se počty jejich kroků liší.

5.7.2 Implementace vyhledávacího procesu v algoritmech

Každý algoritmus musí reimplementovat několik virutálních funkcí jejichž definice se původně nachází v abstraktní třídě `abstract_algorithm`. Samotné vyhledávání pak musí být implementováno dvakrát, jednou pro použití při okamžitém *vyhledávání bez animace* a podruhé pro *vyhledávání s animací*. Každý algoritmus pochopitelně implementuje svůj individuální vyhledávací proces tak, jak je uveden v kapitole 3.

Každý algoritmus má také pro potřeby vyhledávání několik privátních proměnných, které jsou využívány pro obě varianty vyhledávacího procesu. Proto musí také reimplementovat metodu `clear_search()`, která slouží k vymazání obsahu těchto proměnných a k přípravě na nové vyhledávání. Součástí tohoto procesu je také nastavení barevného označení označených uzlů zpět na normální barvy. Další metody `result()` a `result_node()` pak musí být po dokončení vyhledávání schopny vrátit ukazatele na výsledek vyhledávání (nalezený prefix) a uzel, ve kterém se výsledek vyhledávání nachází.

První implementace vyhledávacího procesu je jediná funkce `find_prefix()`, kde proběhne celý vyhledávací proces (zpravidla pomocí několika cyklů) a nakonec je pouze označen výsledný uzel a prefix. Jedná se tedy o úplně základní implementaci vyhledávacího procesu, na které je poté založena jeho druhá implementace.

Druhá implementace vyhledávacího procesu je určena k použití při animaci a vyhledávací proces v ní musí být rozložen do jednotlivých kroků. Je tedy vhodné ji implementovat jako *stavový automat*, který přechází mezi jednotlivými stavy vyhledávacího procesu. Proto je třeba reimplementovat funkce `step_prev()` a `step_next()`, které slouží pouze pro nastavení dalšího stavu na základě současného stavu a případně pro inicializaci vyhledávacích proměnných (např. nastavení kořenového uzlu při začátku vyhledávání). Jak již jejich název napovídá, jsou určeny pro *krok vpřed* a *krok vzad*. Po nastavení aktuálního stavu je v nich poté hned volána metoda `step()`, která pro každý stav implementuje akce, které se mají vykonat. Tyto funkce poté vracejí buď hodnotu `true`, když bylo možné akce vykonat nebo hodnotu `false`, která znamená, že akce nebylo možné vykonat – pro metodu `step_next()` to znamená, že vyhledávání bylo dokončeno, zatímco pro metodu `step_prev()` to znamená, že se nacházíme na samotném začátku vyhledávání a není možno se vrátit.

Metoda `step()` tedy pro každý stav implementuje akce, které se mají provést. To závisí plně na tom, jak rozvrhneme kroky jednotlivých algoritmů. Při jejich implementaci je

ovšem nutné dbát na to, že mohou být vykonávány i v opačném pořadí a vhodně je proto přizpůsobit. Během těchto kroků zpravidla barevně zvýrazňujeme uzel, ve kterém se právě nacházíme, ukazatel přes který přecházíme do dalšího uzlu nebo prefix, který jsme si právě zapamatovali jako zatím nejdelší shodný. Pokud nějaký objekt (uzel či ukazatel) zvýrazňujeme, také při tom voláme metodu pro *centrování scény* nad daný objekt, čímž zajistíme, že bude mít uživatel stálý přehled, kde se zrovna ve struktuře nacházíme a nebude se po scéně muset pohybovat ručně pomocí posuvníků či myši. Pokud čteme nějaké bity ze vstupu, potom je také zvýrazňujeme v poli s bitovým tvarem hledané adresy. Zároveň během těchto akcí zobrazujeme ve *stavovém řádku* komentář, který popisuje prováděné akce. Některé z těchto akcí jsou zabezpečovány metodami ve třídě `animator`, která následně komunikuje s uživatelským rozhraním ve třídě `MainWindow`.

U algoritmu Trie jsou tyto stavy v zásadě pouze tři – stav *uzel* (nacházíme se v uzlu, případně si zapamatujeme nový nejdelší shodný prefix), stav *ukazatel* (čteme bit ze vstupu a kontrolujeme existenci ukazatele, který případně zvýrazníme) a stav *konec* (vyhledávání bylo dokončeno, označujeme výsledný uzel a prefix).

U algoritmu CPE je těchto stavů už pět – *uzel* (nacházíme se v novém uzlu, při použití *leaf pushing* je nutno zkontrolovat výskyt prefixu na indexu, na kterém se v rodičovském uzlu nacházel ukazatel přes který jsme přešli do tohoto uzlu), *čtení bitů a kontrola prefixu* (čteme bity ze vstupu a na příslušném indexu v poli prefixů hledáme výskyt prefixu), *kontrola ukazatele* (v poli ukazatelů kontrolujeme výskyt ukazatele, při *leaf pushing* je tento stav přeskočen protože tato kontrola je provedena v předchozím stavu), *přechod do dalšího uzlu* (zvýraznění ukazatele a přechod do dalšího uzlu) a *konec* (konec vyhledávání, zvýraznění výsledku vyhledávání).

U Tree Bitmap je také pět stavů a jejich význam je takřka totožný jako u CPE, samozřejmě s tím rozdílem, že indexujeme do jednotlivých *bitmap* a následně až poté do polí s prefixy a ukazateli.

Aby správně fungovalo krokování animace vzad, bylo u CPE, Tree Bitmap, ale i Trie potřeba implementovat několik podpůrných funkcí, které umožňují nalezení a označení předchozího zapamatovaného nejdelšího shodného prefixu (musíme se vracet k rodičovským uzlům a během toho také vracet bity přečtené ze vstupu) nebo získat index do polí ukazatelů rodičovského uzlu, abychom v nich mohli označit danou šipku (pouze u CPE a Tree Bitmap).

5.7.3 Přepínání algoritmů během animace a související problémy

Protože jsou některé prvky uživatelského rozhraní v programu unikátní (vyskytují se pouze jednou pro všechny algoritmy), může je vždy využívat pouze jeden, v našem případě aktivní algoritmus. Těmito prvky jsou *stavový řádek*, do kterého algoritmy vypisují komentář k prováděným akcím, dále *Label s bitovým tvarem hledané adresy*, do kterého algoritmy označují přečtené bity, *tabulka prefixů*, ve které algoritmy označují aktuálně zapamatovaný nejdelší shodný prefix a také *posuvník pro přiblížení grafické scény*, kde scéna každého algoritmu může být přiblížena individuálně. Po přepnutí algoritmu ovšem chceme, aby tyto prvky zobrazovaly stav platný pro tento algoritmus.

Proto ve třídě `algorithms` existují proměnné a metody, pomocí kterých jsou tyto informace ukládány individuálně pro každý algoritmus. Během animace tedy všechny algoritmy normálně posílají zprávy určené pro zobrazení ve *stavovém řádku* či požadavky pro zvýraznění prefixu v *tabulce prefixů* nebo zvýraznění bitů vstupu těmto metodám, které tyto informace ukládají do proměnných pro daný algoritmus. Data jsou ale poté zobrazena pouze

pro aktivní algoritmus, zatímco u neaktivních algoritmů jsou jenom uložena. Jakmile dojde k přepnutí algoritmu, jsou stavy těchto prvků načteny pro nově aktivní algoritmus a odpovídají tak jeho stavu.

Každý algoritmus má vlastní grafickou scénu s reprezentací jeho vnitřní struktury, ve které během vyhledávání normálně může barevně vyznačovat jednotlivé uzly i když tato scéna není viditelná, takže samotné přepínání scén není problém.

5.8 Uživatelská nápověda

Aby uživatel nemusel hledat detaily o použití programu v této práci, je v programu zakomponována uživatelská nápověda. Ta popisuje jednotlivé sekce uživatelského rozhraní a jejich účel, klávesové zkratky, práci s editorem prefixů, vyhledávání, a také jednotlivé položky nastavení. Kromě toho jsou zde uvedeny informace o algoritmech LPM, včetně pseudokódů jejich vyhledávacích procesů, tak jak jsou v programu implementovány. Nápověda tedy uživateli poskytuje praktické rady pro práci s programem.

Obsah nápovědy je uložen v souboru *help.html* ve složce *help*, která se nachází v adresáři s programem. Jelikož se jedná o *html* soubor, lze obsah nápovědy snadno editovat. V programu je pak nápověda zobrazena v dialogu s prvkem `QTextBrowser`, který umí interpretovat jednoduché *html* dokumenty.

5.9 Možnosti rozšíření aplikace

Aplikaci lze v současné podobě dále rozvíjet a doplňovat do ní další funkčnost. Do budoucna je možné program rozšířit o zejména další algoritmy LPM. Pro usnadnění jejich implementace jsem vypracoval příručku, která popisuje, které algoritmy jsou pro případnou implementaci v aplikaci vhodné a jak lze do programu další algoritmy přidat včetně technických detailů (které metody je třeba reimplementovat a jak). Příručka se nachází v příloze A.

Dále by bylo možné implementovat podporu pro IPv6 (vstupní adresa i prefixy), ale musela by se vyřešit otázka kombinování dvou typů IP adres (IPv4 a IPv6) v algoritmech.

Také by připadala v úvahu možnost v aplikaci animovat i samotnou tvorbu vnitřních struktur jednotlivých algoritmů. Výpočetně by to ovšem byl úkol poměrně náročný, protože by po každém přidání nového uzlu musel proběhnout poziční algoritmus (v případě stromů). Také doba trvání takové animace by při rozsáhlých stromech (či jiných vnitřních strukturách) byla dlouhá. Proto by při případné implementaci bylo asi nejvhodnější, aby tato konstrukce vnitřních struktur byla implementována v externím okně, kde by uživatel mohl kdykoliv tuto animaci přerušit a vrátit se k vyhledávání nad již hotovými strukturami do hlavního okna programu.

Kapitola 6

Závěr

Tato práce popisuje návrh a implementaci programu, který slouží k vizualizaci algoritmů pro vyhledání nejdelšího shodného prefixu (LPM). Tento problém spadá do oblasti počítačových sítí, kde jsou algoritmy LPM využívány zejména při klasifikaci paketů. Proto bylo důležité nejprve nastudovat tuto problematiku, jejíž znalost je nezbytná pro pochopení samotných algoritmů LPM.

V úvodní části pak jsou popsány tři algoritmy LPM, které jsou ve výsledném programu implementovány. Tyto algoritmy jsou založeny na různých myšlenkách a přístupech. Také složitost jednotlivých algoritmů je různá – jsou zastoupeny jak jednoduché algoritmy (*Trie*), tak i pokročilé algoritmy jako *Tree Bitmap*, nebo *CPE*. Všechny materiály, ze kterých jsem při studiu a následné implementaci algoritmů v programu vycházel, jsou uvedeny v seznamu literatury.

Po nastudování jednotlivých algoritmů následovala fáze návrhu samotného programu. Program byl navržen tak, aby přehledně a názorně vizualizoval vyhledávací proces jednotlivých algoritmů, stejně jako jejich vnitřní strukturu. Při návrhu bylo také zohledněno, aby bylo možno program použít pro pedagogické účely – byla navržena možnost řízení vyhledávacího procesu samotným uživatelem, stejně jako možnost přepínání mezi algoritmy během vyhledávání.

Poté už je popsána samotná implementace výsledného programu. Program disponuje grafickým uživatelským prostředím s přehledně rozmístěnými ovládacími prvky. Obsahuje nástroj pro tvorbu prefixů, nad kterými je pak možno zahájit vyhledávání. Prefixy samotné je také možno ukládat i načítat ze souborů. Pro každý algoritmus je graficky zobrazena vnitřní struktura, která reprezentuje tyto prefixy, a ve které jsou během vyhledávání vyznačovány jednotlivé položky, což napomáhá celkové názornosti. U některých algoritmů lze nastavit jejich parametry (délka střídy, optimalizace), takže si uživatel může přizpůsobit jejich zobrazení dle libosti. Vyhledávání je pak možno provést rychle se zobrazením výsledku nebo je umožněno vyhledávací proces krokovat a lépe tak pochopit princip jednotlivých algoritmů.

Program je vhodný pro demonstraci vyhledávání nad rozumně velkou množinou prefixů, pro příliš velké množiny prefixů může být vizualizace vnitřních struktur algoritmů náročná. Stejně tak při velkých délkách střídy (8) jsou jejich uzly příliš velké a složité, proto je vhodné používat délky spíše menší.

V práci jsou nakonec naznačeny i možnosti rozšíření programu o další algoritmy a případně i funkčnost.

Literatura

[1] *Qt Project* [online]. 2013, [Aktualizováno 2013] [cit. 2013-04-22]. Dostupné z: <http://qt-project.org/>.

[2] *Qt Project: Diagram Scene Example* [online]. 2013, [Aktualizováno 2013] [cit. 2013-04-22]. Dostupné z: <https://qt-project.org/doc/qt-4.7/graphicsview-diagramscene.html>.

[3] Blanchette, J. a Summerfield, M. *C++ GUI Programming with Qt 4*. 2. vyd. Upper Sadle River, NJ, USA: Prentice Hall, 2008. 752 s. ISBN 0-13-235416-0.

[4] Eatherton, W., Varghese, G. a Dittia, Z. Tree bitmap: hardware/software IP lookups with incremental updates. *ACM SIGCOMM Computer Communication Review*. Duben 2004, roč. 34, č. 2. S. 97 – 122. ISSN 0146-4833. Dostupné z: <http://portal.acm.org/citation.cfm?doid=997150.997160>.

[5] Hinden, R. a Deering, S. *Internet Protocol Version 6 (IPv6) Addressing Architecture* [RFC 3513]. duben 2003. Dostupné z: <http://www.ietf.org/rfc/rfc3513.txt>.

[6] Kováčik, M. *Algoritmy pro vyhledání nejdelšího shodného prefixu.* FIT VUT v Brně, 2012. 62 s., 1 s. příl. Diplomová práce.

[7] Medhi, D. a Ramasamy, K. *Network routing: Algorithms, Protocols, and Architectures*. Amsterdam: Elsevier, 2007. 824 s. The Morgan Kaufmann Series in Networking Series. ISBN 0-12-088588-3.

[8] Prata, S. *Mistrovství v C++*. Přeložil Sokol, B. 1. vyd. Praha: Computer Press, 2001. 966 s. ISBN 80-7226-339-0.

[9] Reingold, E. M. a Tilford, J. S. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering*. Březen 1981, roč. 7, č. 2. S. 223 – 228. ISSN 0098-5589. Dostupné z: <http://dx.doi.org/10.1109/TSE.1981.234519>.

[10] Skačan, M. *Algoritmy pro vyhledání nejdelšího shodného prefixu.* FIT VUT v Brně, 2010. 31 s. Bakalářská práce.

[11] Srinivasan, V. a Varghese, G. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*. únor 1999, roč. 17, č. 1. S. 1 – 40. ISSN 0734-2071. Dostupné z: <http://portal.acm.org/citation.cfm?doid=296502.296503>.

[12] Tanenbaum, A. S. *Computer networks*. 4. vyd. New Jersey: Prentice-Hall, 2003. 384 s. ISBN 0-13-066102-3.

[13] Tobola, J. *Longest Prefix Match.* FIT VUT v Brně, 2009. 21 s., 1 s. příl. Pojednání k tématu dizertační práce.

[14] Varghese, G. *Network algorithmics: An Interdisciplinary Approach To Designing Fast Networked Devices.* Amsterdam: Elsevier, 2005. 465 s. The Morgan Kaufmann Series in Networking Series. ISBN 0-12-088477-1.

[15] Walker, J. Q. *A Node-Positioning Algorithm for General Trees.* Sitterson Hall Chapel Hill, NC 27599-3175, USA: University of North Carolina at Chapel Hill, Department of Computer Science, září 1989. 29 s. A TextLab Report [TR89-034]. Dostupné z: <`www.cs.unc.edu/techreports/89-034.pdf`>.

[16] Walker, J. Q. *Positioning Nodes For General Trees* [online]. 1991, [Aktualizováno 1991-02-01] [cit. 2013-04-09]. Dr. Dobb's: The world of software development. Dostupné z: <`http://www.drdobbs.com/positioning-nodes-for-general-trees/184402320?pgno=2`>.

Příloha A

Příručka pro doplnění algoritmů do programu

Tato příručka popisuje přidávání dalších algoritmů do implementace programu. Jejím cílem je usnadnit orientaci v kódu a vytyčit změny, které je nutné provést za účelem úspěšného přidání dalších algoritmů.

A.1 Vlastnosti algoritmu vhodného pro přidání do aplikace

Algoritmů pro vyhledání nejdelšího shodného prefixu samozřejmě existuje více, než jsme si jich popsali a implementovali v programu. Proto je možné do programu doplnit implementaci dalších algoritmů, které pomocí něj můžeme vizualizovat. Algoritmus vhodný pro implementaci v naší aplikaci ovšem musí splňovat několik kritérií.

Nejprve si musíme uvědomit, že abychom nějaký algoritmus mohli vizualizovat, musí mít daný algoritmus vnitřní strukturu, která je k vizualizaci vhodná. Co se implementačního hlediska týče, *Qt* při patřičně vynaloženém úsilí nabízí prostředky k vizualizaci takřka čehokoliv, takže zde problém není. Problém je, že vizualizace některých struktur je jednak komplikovaná, ale hlavně nemusí být vizuálně názorná a přehledná. Proto jsou v programu implementovány algoritmy využívající stromové struktury, jejichž vizualizace je nejen poměrně jednoduchá, ale hlavně přehledná a názorná. Naopak například vizualizace tabulek s rozptýlenými položkami (*hashovacích tabulek*) nebude moc atraktivní a při velkém počtu položek pak i nepřehledná. Proto zvažte, zdali algoritmus, který se chystáte doimplementovat, má vnitřní strukturu, jejíž vizualizace bude uživatelsky atraktivní a přehledná.

Další vlastností, kterou musí algoritmus splňovat, je možnost rozdělení vyhledávacího procesu do kroků, které poté bude možno vizualizovat nad jeho vnitřní strukturou. V programu využíváme krokování animace vyhledávacího procesu, a proto by i další algoritmy měly mít vyhledávací proces rozdělen do kroků, které bude možno vykonat i v opačném pořadí. Předpokládám, že u většiny algoritmů by toto nemělo činit potíže, nicméně než se rozhodnete nějaký algoritmus doimplementovat, zauvažujte i nad tímto problémem a jeho řešením.

A.2 Implementace algoritmu a souvisejících tříd

Pro úspěšnou implementaci nového algoritmu je potřeba vytvořit dvě nové třídy. Jednu pro implementaci algoritmu jako takového, a druhou pro vizualizaci jeho vnitřní struktury. Při

implementaci doporučuji nejprve začít vizualizací vnitřní struktury a až následně samotným algoritmem vyhledávání.

Všechny metody v této kapitole jsou uvedeny bez parametrů a návratových typů, jejich přesné prototypy lze nalézt v příslušných třídách.

A.2.1 Vnitřní struktura algoritmu

Implementace vnitřní struktury algoritmů plně závisí na implementovaném algoritmu. Stromové struktury, které jsou použity ve všech algoritmech implementovaných v programu, se skládají z jednotlivých uzlů. Proto je pro každý algoritmus vytvořena třída reprezentující uzel jeho stromu. Budete-li implementovat algoritmus využívající stromové struktury, můžete se při jejich tvorbě inspirovat již implementovanými třídami, viz dále. Pokud bude třeba využít jiných struktur, je třeba je implementovat ručně dle potřeby. Také se předpokládá vytvoření pomocných metod, které budou maximálně usnadňovat použití těchto struktur ve vyhledávacím procesu daného algoritmu.

Pro implementaci stromových struktur existuje abstraktní třída `node`, která obsahuje definice virtuálních metod, jenž je třeba implementovat. Pro každý algoritmus je pak vytvořena třída `node_nazevAlgoritmu`, která dědí od třídy `node` a tyto metody reimplementuje a také přidává vlastní specifické metody.

Univerzální metody třídy `node`

Mezi univerzální metody, které je možno použít pro libovolný algoritmus (a nejen pro stromové struktury), patří metody pro práci s grafickou scénou:

- `set_scene()` – nastavení scény, ve které bude uzel vykreslen
- `get_scene()` – získání ukazatele na scénu, ve které je uzel vykreslen

Dále metody pro nastavení barev pozadí a okraje uzlu, které lze použít při vykreslování uzlu:

- `set_background_color()` – nastavení barvy pozadí
- `get_backgroung_color()` – získání barvy pozadí
- `set_border_color()` – nastavení barvy okraje
- `get_border_color()` – získání barvy okraje

Také metody pro indikaci zvýraznění uzlu, které slouží k určení, zdali je uzel barevně zvýrazněn:

- `set_selected()` – nastavení zvýraznění uzlu
- `get_selected()` – jestli je uzel zvýrazněn
- `set_selected_result()` – nastavení zvýraznění uzlu jako výsledek vyhledávání
- `get_selected_result()` – jestli je uzel zvýrazněn jako výsledek vyhledávání

A nakonec nastavení typu uzlu, který je využit při mapování ukazatelů na tento uzel (nový typ uzlu je třeba přidat do výčtového typu `node_types`, definovaného v souboru *node.h*):

- `set_type()` – nastavení typu uzlu
- `get_type()` – získání typu uzlu

Virtuální metody třídy `node`

Třída `node` dále obsahuje několik virtuálních metod, jejichž reimplementaci je nutno provést pro každý algoritmus zvlášť. Pro použití ve všech algoritmech je pak určena metoda `update_all_texts()`, která slouží k aktualizaci textových popisků v uzlech po změně nastavení (barva, font).

U algoritmů využívajících stromové struktury můžete pro rozmístění jejich uzlů použít *Walkerův algoritmus*, který je univerzálně implementován pro uzly odvozené od třídy `node`. Nicméně je třeba reimplementovat několik virtuálních metod, které Walkerův algoritmus pro práci využívá, a které se v závislosti na vnitřní struktuře jednotlivých uzlů liší. Pokud není uvedeno jinak, metody vrací ukazatele.

- `firstchild()` – vrací nejlevějšího potomka uzlu
- `lastchild()` – vrací nejpravějšího potomka uzlu
- `leftsibling()` – vrací uzel nalevo, který má stejného rodiče jako aktuální uzel
- `rightsibling()` – vrací uzel napravo, který má stejného rodiče jako aktuální uzel
- `leftneighbor()` – vrací levého souseda tohoto uzlu (stejná úroveň, ale jiný rodič)
- `set_leftneighbor()` – nastaví levého souseda tohoto uzlu
- `parentt()` – vrací rodičovský uzel
- `xcoord()` – vrací relativní x-pozici uzlu (`int`)
- `set_xcoord()` – nastaví x-pozici uzlu (`int`)
- `prelim()` – vrací hodnotu proměnné `prelim` (`int`) – pomocná proměnná Walkerova algoritmu
- `set_prelim()` – nastaví hodnotu proměnné `prelim` (`int`)
- `modifier()` – vrací hodnotu proměnné `modifier` (`int`) – pomocná proměnná Walkerova algoritmu
- `set_modifier()` – nastaví hodnotu proměnné `modifier` (`int`)
- `clear_walker_vars()` – vynuluje pomocné proměnné (`xcoord, prelim, modifier`).

Pro vykreslení každého uzlu pak je třeba reimplementovat metodu `paint()`, která je zděděna od třídy `QGraphicsWidget`, a která slouží k vykreslení daného objektu (jeho tvar, barvy...).

A.2.2 Algoritmus, vyhledávací proces

Implementace vyhledávacího procesu algoritmu, stejně jako algoritmy související s konstrukcí jeho vnitřní struktury, je pro každý algoritmus individuální. Pro odvozování algoritmů existuje abstraktní třída `abstract_algorithm`, která obsahuje virtuální metody k reimplementaci, ale i univerzální metody. Pro každý algoritmus je pak třeba vytvořit novou třídu (v programu jsou to třídy `visualization_nazevAlgoritmu`), která bude dědit od `abstract_algorithm` a bude obsahovat reimplementace jejích virtuálních metod a své vlastní podpůrné metody.

Univerzální metody třídy `abstract_algorithm`

Mezi základní univerzální metody patří metody pro práci s grafickou scénou:

- `create_scene()` – vytvoří novou grafickou scénu pro vizualizaci vnitřních struktur daného algoritmu, parametrem je ukazatel na `QGraphicsView`, do kterého bude scéna vykreslena
- `get_scene()` – vrací ukazatel na vytvořenou grafickou scénu

Dále metody pro práci s centrálním animačním objektem:

- `set_animator()` – nastavení objektu řídícího animaci
- `get_animator()` – vrací ukazatel na řídící objekt animace

Také metody pro určení, zdali byla vizualizace vnitřní struktury přizpůsobena grafické scéně:

- `set_fitted_in_view()` – nastavení, jestli byla scéna přizpůsobena (`bool`)
- `get_fitted_in_view()` – ověření, jestli byla scéna přizpůsobena (`bool`)

Pro nastavení hledané adresy před začátkem vyhledávání pak slouží následující metody:

- `set_input()` – nastavení hledané adresy (její bitový tvar)
- `get_input()` – získání hledané adresy (bitový tvar)

A nakonec nastavení typu algoritmu, který je využit při identifikaci dat pro/od daného algoritmu (nový algoritmus je třeba přidat do výčtového typu `algoritmus`, definovaného v souboru *abstract_algorithm.h*):

- `set_algorithm_type()` – nastavení typu algoritmu
- `get_algorithm_type()` – získání typu algoritmu

Virtuální metody třídy `abstract_algorithm`

Přestože jsou virtuální metody z `abstract_algorithm` určeny pro reimplementaci ve všech algoritmech, některé z nich není bezpodmínečně nutné reimplementovat v algoritmech, které nevyužívají Walkerův algoritmus (nemají stromovou strukturu). Jsou to:

- `reposition_tree()` – provede rozmístění uzlů stromu zvoleným algoritmem (dle nastavení)
- `clear_walker_vars_all()` – ve všech uzlech stromu vymaže hodnoty pomocných proměnných Walkerova algoritmu (voláním metody `clear_walker_vars()` pro každý uzel stromu)

Další metody už je nutné reimplementovat v každém algoritmu:

- `center_to_root()` – vycentruje grafickou scénu na kořenový uzel jeho struktury (voláním metody `center_to_node()` na kořenový uzel v příslušné scéně)

- `construct_graph()` – na základě seznamu prefixů vytvoří vnitřní struktury reprezentující uložení těchto prefixů daným algoritmem
- `destroy_structure()` – smaže celou vnitřní strukturu algoritmu (včetně objektů z grafické scény)

Pro vyhledávání daným algoritmem pak slouží následující metody:

- `find_prefix()` – provede okamžité vyhledání, bez animace
- `result()` – vrátí výsledek vyhledávání (nalezený nejdelší shodný prefix)
- `result_node()` – vrátí ukazatel na uzel obsahující výsledek vyhledávání
- `clear_search()` – vymaže proměnné a ukazatele používané při vyhledávání, provádí se před novým vyhledáváním
- `step_next()` – nastaví stavový automat na provedení dalšího kroku algoritmu (vyhledávání s animací) a tento krok provede (vrací `true`, pokud byl krok proveden nebo `false`, pokud krok nebyl proveden)
- `step_prev()` – nastaví stavový automat na provedení předchozího kroku algoritmu (vyhledávání s animací) a tento krok provede (vrací `true`, pokud byl krok proveden nebo `false`, pokud nebyl)
- `step()` – vykonání samotných akcí příslušného kroku, metoda je volána v `step_prev()` a `step_next()`

Pro zjednodušení vyhledávacího procesu je pak vhodné vytvořit několik pomocných metod, které zpracovávají činnost vyskytující se na více místech tohoto procesu. Jejich implementace je ovšem plně závislá na konkrétním algoritmu.

A.3 Úpravy v existujících třídách

Jakmile budou implementovány třídy představující nový algoritmus a jeho uzly, je třeba nový algoritmus začlenit do programu a umožnit jeho používání. Proto je nutné upravit několik metod v již existujících třídách.

Pro usnadnění lokalizace míst, kde je třeba provést změny, jsou v kódu tato místa označena komentářem „`/********* SEM DOPLNTE DALSI ALGORITMY *********/`“.

A.3.1 Úpravy tříd pro práci s algoritmem

Pro zobrazení vnitřní struktury nového algoritmu je třeba do uživatelského rozhraní hlavního okna přidat do objektu `tabWidget` další záložku, do které je pak třeba vložit objekt `QGraphicsView`, který umožní zobrazení grafiky. To lze učinit pomocí nástroje *Qt Designer* nebo ruční úpravou kódu v souboru *mainwindow.ui*.

Poté je třeba upravit metody třídy `algorithms`. Ta obsahuje metody zjednodušující práci s více algoritmy současně a také obsahuje proměnné uchovávající stav některých prvků uživatelského rozhraní. Také obsahuje metody pro tvorbu či smazání objektů, představující jednotlivé algoritmy. Proto je třeba doplnit jak tyto proměnné, tak kód v metodách a někdy i vytvořit celé nové metody. Při jejich tvorbě jako vzor použijte již existující metody pro jiné algoritmy, a změňte pouze příslušné detaily (proměnné, volání jiných metod, ...). Co se

úpravy metod týče, příslušná místa jsou vždy označena výše uvedeným komentářem a kód, který je třeba doplnit, se vždy nachází těsně nad tímto komentářem (kód je třeba zkopírovat a patřičně změnit).

Aby bylo možné zobrazit vnitřní strukturu nově přidaného algoritmu, je třeba do třídy `graphics_scene` přidat metodu pro přidávání uzlů nového algoritmu do grafické scény. Při její tvorbě doporučuji inspirovat se existujícími metodami pro přidávání uzlů ostatních algoritmů. Dále je třeba uchovávat seznam uzlů tohoto typu, který je pak použit v metodě `update_all_texts()` pro aktualizaci textových popisků ve všech uzlech.

Ve třídě `MainWindow`, která má na starosti především interakci s uživatelským rozhraním, je nutno v metodě `setup_ui_details()` na vyznačených místech přidat kód pro nastavení objektu `QGraphicsView`, který byl vytvořen pro zobrazení vnitřních struktur nového algoritmu. Jedná se o nastavení vykreslování (*antialiasing*) a také umožnění posunu scény pomocí myši. V metodě `zoom()` je pak nutné přidat možnost přiblížení této scény. Nakonec je třeba v metodě `tab_algorithm_changed()` přidat kód související se změnou algoritmu přepnutím záložky. Inspirujte se existujícím kódem.

Pro umožnění vyhledávání a vyhledávání s animací novým algoritmem se musí provést změny ve třídě `animator`. Především je nutné doplnit metody pro krokování algoritmu – vykonání kroku zpět a kroku vpřed. Při jejich tvorbě se inspirujte existujícím kódem. Do některých existujících metod je pak třeba na označená místa doplnit volání metod nového algoritmu (např. vyhledání bez animace nebo určování počtu kroků jednotlivých algoritmů).

A.3.2 Přidání položek do nastavení

Pokud přidaný algoritmus vyžaduje nastavení nějakého parametru uživatelem, je třeba tento parametr přidat do uživatelského nastavení. To vyžaduje úpravu dvou tříd.

Do třídy `preferences` je třeba přidat příslušné proměnné a metody pro přístup k nim. Také je nutné upravit metody `save()`, `load()` a `load_default()`, kde je třeba doplnit kód pro ukládání a načítání těchto parametrů do souboru. V metodě `load_default()` pak doplnit výchozí nastavení tohoto algoritmu, které je použito, pokud načtení ze souboru selže.

Jakmile jsou provedeny úpravy třídy `preferences`, je třeba upravit dialog reprezentující nastavení a do něj doplnit ovládací prvky pro úpravu daných parametrů. Ten je představován třídou `preferences_dialog`. Dále je třeba v metodě `setup_preferences()` doplnit kód pro zkopírování nastavení těchto prvků z globálního nastavení do lokálního nastavení, které je pak prostřednictvím ovládacích prvků dialogu měněno. V metodě `on_use_btn_clicked()` je pak toto lokální nastavení překopírováno zpět do globálního nastavení (potvrzení změn) – také zde je třeba doplnit příslušný kód.

Pokud změna některého z parametrů musí způsobit překreslení vnitřní struktury algoritmu nebo jiné následné akce, je třeba při kopírování dočasného nastavení zpět do globálního změnu těchto parametrů detekovat a následně doplnit příslušné akce po schování dialogu – v metodě `on_actionUpravit_nastaven_triggered()` ve třídě `MainWindow`.