VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

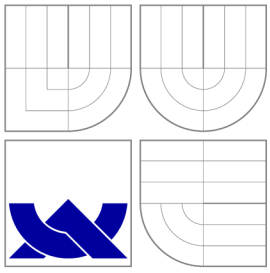
GENEROVÁNÍ CONTENT ADDRESSABLE DELAYED DFA Z REGULÁRNÍCH VÝRAZŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

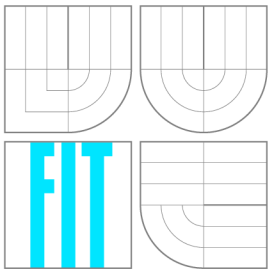
AUTOR PRÁCE
AUTHOR

JAN HAMMER

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GENEROVÁNÍ CONTENT ADDRESSABLE DELAYED DFA Z REGULÁRNÍCH VÝRAZŮ

DEVELOPMENT OF GENERATION OF CONTENT ADDRESSABLE DFA FROM A SET OF REGULAR EXPRESSIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN HAMMER
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KAŠTIL
SUPERVISOR

BRNO 2014

Abstrakt

Práce se zabývá konstrukcí rozšířených typů konečných automatů ze sad regulárních výrazů. Hlavní důraz je na rozšíření CD^{2}FA – Content Addressed Delayed Input DFA, které je navrženo k použití při hloubkové analýze paketů v síti, za účelem snížení paměťové náročnosti a zachování rychlosti výpočtu. Nad takto zkonstruovanými automaty jsou zkoumány statistiky paměťové náročnosti, které ukazují, že CD^{2}FA jsou řádově desetkrát méně paměťově náročné, než původní DFA. Dále jsou prezentovány některá vylepšení procesu konstrukce CD^{2}FA, především vylepšení přípravy adresace stavů za použití perfektního hashování.

Abstract

This work deals with contruction of enhanced types of finite automata from sets of regular expressions. The main focus is on enhancement called CD^{2}FA – Content Addressed Delayed Input DFA, which is designed to be used for deep packet inspection throughout the net, in order to lower memory requirements and retain the throughput. Automata constructed in this manner are used to get memory requirement statistics which show that CD^{2}FAs are about ten times more compact then original DFAs. Then some enhancements dealing with the process of CD^{2}FA construction are presented, particularly enhancement of preparation of state addressing by perfect hashing.

Klíčová slova

hloubková analýza paketů, konečné automaty, perfektní hashování

Keywords

deep packet inspection, finite automata, perfect hashing

Citace

Jan Hammer: Generování Content Addressable Delayed DFA z regulárních výrazů, bakalářská práce, Brno, FIT VUT v Brně, 2014

Generování Content Addressable Delayed DFA z regulárních výrazů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kaštila.

.....
Jan Hammer
20. května 2014

Obsah

Kapitola 1

Úvod

Tato práce se zabývá speciálními rozšířeními modelu konečných automatů, které jsou používány síťovými zařízeními při hloubkové analýze paketů. Je to jednak rozšíření D^2FA – Delayed Input Deterministic Finite Automaton, ale hlavně CD^2FA – Content Addressed Delayed Input Deterministic Finite Automaton, které jej dále rozšiřuje. Tato rozšíření jsou zaměřena na to, aby při zachování stejné funkčnosti a v případě CD^2FA i rychlosti výpočtu, měli menší paměťové nároky.

Oba tyto návrhy automatů jsou založeny na práci pana Saileshe Kumara et al. [2] a [3] a na těchto článcích je celá tato práce založena.

V kapitole 2 je blíže vysvětlena motivace pro tato rozšíření. Je představen D^2FA, který je paměťově úspornější, ale výpočetně pomalejší. Na tomto modelu dále staví rozšíření CD^2FA, které usiluje o skloubení výhod DFA – rychlosti a D^2FA – úspory paměti. Je vysvětleno jak v něm funguje adresování, jak uspořádat data v paměti, aby k němu bylo možno použít bezkolizní perfektní hashování. Dále je popsáno jak funguje proces transformace DFA na D^2FA a následně na CD^2FA.

V kapitole 3 je prezentován návrh na alternativní způsob přípravy perfektního hashování. Dále je v kapitole 4 popis implementace konstrukce CD^2FA z DFA a simulace běhu automatu. Tato implementace je součástí této práce. Konečně v kapitole 5 jsou popsány statistiky CD^2FA, které byly tímto programem zkonstruovány – jedná se hlavně o paměťové nároky původního DFA oproti získanému CD^2FA a tyto výsledky jsou zhodnoceny.

Kapitola 2

Typy rozšíření konečných automatů

Tato práce a obzvláště tato kapitola je založena na práci pana Saileshe Kumara et al., kde jsou představeny typy automatů D^{2}FA [2] a CD^{2}FA [3].

2.1 Hloubková inspekce paketů a regulární výrazy

Hloubková inspekce paketů (deep packet inspection - DPI) je technika výpočetně náročná. Pro současná síťová zařízení je typické analyzovat pouze hlavičku paketu a na jejím základě provádět směrování, filtrování provozu a další úkony. Jedná se o méně dat a pro mnoho účelů je to zcela dostatečné. Přesto může být užitečné analyzovat celý paket – včetně všech užitečných dat. Z tohoto důvodu je vhodné ulehčit hardwaru návrhem efektivních algoritmů – algoritmů, které budou brát v úvahu specifika problematiky DPI.

Pokud zařízení má provádět DPI, musí být definovány vzory, které budou v datech vyhledávány. Tyto vzory jsou definovány pomocí sad regulárních výrazů. Regulární výraz převedený do formy konečného automatu slouží pro rozhodnutí, jestli byl určitý výraz nalezen v datech, nebo nikoli.

V praxi ovšem nastává problém, kterým je velikost těchto konečných automatů. Ta se může pro složitější sady regulárních výrazů pohybovat až v Gigabytech, a to může být pro síťová zařízení neúnosné. Proto vznikly návrhy rozšíření stávajícího modelu konečných automatů, které by při zachování funkčnosti (a pokud možno i výkonu) potřebovaly méně místa v paměti.

2.2 Od regulárních výrazů k CD^{2}FA

Pokud má být konečný automat implementován programem (nebo hardwarem) je přirozené, že se bude jednat o automat deterministický. To mimo jiné znamená, že se bude jednat o automat bez ε-přechodů, tj. při každém provedení přechodu se přečte jeden znak na vstupu. Vstupem budou data, která přicházejí do síťového zařízení po síti. Tedy bude to proud libovolných bytů. Můžeme tedy předpokládat, že vstupní abeceda automatu bude mít 256 symbolů.

V následujících kapitolách je již řeč nikoli o standardních konečných automatech (či jejich speciálních typech), nýbrž o automatech s dalšími definovanými vlastnostmi. Jsou to:

- Delayed Input DFA – D^{2}FA (deterministický konečný automat s pozdrženým vstupem)
- Content Addressed Delayed Input DFA - CD^{2}FA (obsahem adresovaný deterministický konečný automat s pozdrženým vstupem)

Smyslem těchto vylepšení je zmenšit velikost automatu v paměti. Jak uvádí S. Kumar [3], přístupy do hlavní paměti, kde je automat uložen jsou časově nejnáročnější úkon, a to do takové míry, že počet přístupů do této paměti pro jeden vstupní symbol se stává měřítkem rychlosti automatu.

2.2.1 Rozšíření konečných automatů - D^{2}FA

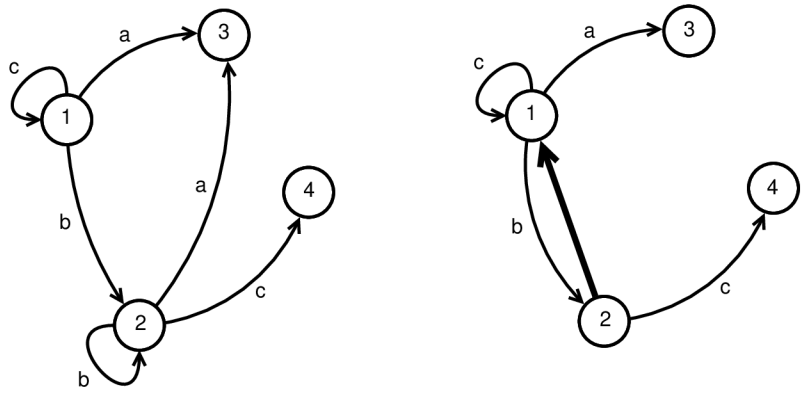
D^{2}FA definuje implicitní přechody (default transitions). Implicitní přechod se provede tehdy, pokud pro současný stav a vstup není definován následující stav standardním způsobem. Při tom ovšem není přečten symbol na vstupu – podobně jako u ε-přechodu.

Pro implicitní přechody musí platit:

- Každý stav může mít nejvýše jeden implicitní přechod, aby se zachoval determinismus.
- Je možné provést více než jeden implicitní přechod za sebou. Proto nesmí vzniknout smyčka implicitních přechodů, jinak by se automat mohl ocitnout v nekonečné smyčce. Implicitní přechody tedy tvoří stromy.

Smysl zavedení D^{2}FA má souvislost s tím, že chceme snížit paměťovou náročnost konečných automatů. Pokud mají dva stavy pro n vstupů stejné následující stavy, lze u jednoho stavu všechny tyto přechody zrušit a nahradit je jedním implicitním přechodem do druhého stavu. Pak je zachována funkčnost automatu beze změny (vzniklý D^{2}FA je ekvivalentní původnímu DFA) a byla ušetřena paměť pro $n - 1$ přechodů.

Příklad zavedení implicitního přechodu je na obrázku 2.1. V levé části je DFA. Na něm mají stavy 1 a 2 definovány přechody pro vstupy a, b a c, přičemž vstupy a a b mají stejný cílový stav, kdežto c má jiný. V pravé části je D^{2}FA, který vznikl odstraněním přechodů se stejným cílem z jednoho stavu a jejich nahrazením implicitním přechodem (vyznačený silnou šipkou). Oba přechody označené symbolem c musí být zachovány, protože se neshodují jejich cíle.



Obrázek 2.1: Zavedení implicitního přechodu D^{2}FA

Na druhou stranu byla potenciálně snížena rychlost algoritmu, protože při provedení implicitního přechodu není zpracován symbol na vstupu – automat „čeká“ na provedení explicitního přechodu. Explicitním přechodem nazveme přechod, který je označen nějakým symbolem. Jedná se o klasický přechod DFA.

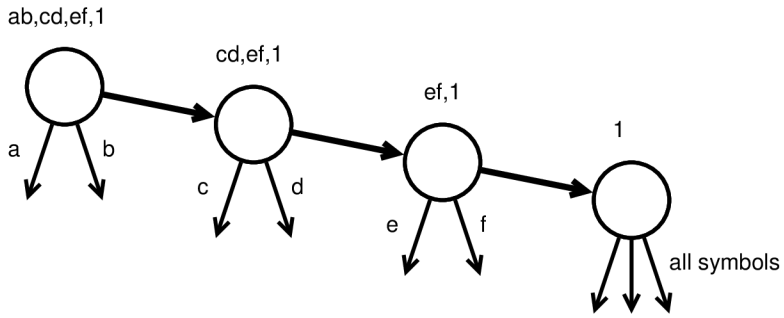
2.2.2 Další rozšíření konečných automatů - CD^{2}FA

CD^{2}FA dále staví na D^{2}FA, cílem tohoto rozšíření je získat výhody klasického DFA – rychlost (při každém provedení přechodu se zpracuje jeden vstupní symbol) i D^{2}FA – menší paměťové nároky. Snížení rychlosti D^{2}FA vychází z předpokladu, že na činnosti automatu je časově nejnáročnější přístup do hlavní paměti, kam se přistoupí vždy, když se má provést přechod. V případě D^{2}FA je to tedy potenciálně i vícekrát pro vstupní symbol. Hlavní myšlenka CD^{2}FA spočívá v tom, že se k adresaci stavů použijí tzv. *content labely*. Content label je přiřazen každému stavu a obsahuje informace o tom, pro které symboly je definován explicitní přechod, a to jak u daného stavu, tak u stavů, které jsou jeho předky, ve stromě implicitních přechodů. Jinými slovy obsahuje seznam symbolů, pro které má stav explicitní přechody a dále celý content label svého předka ve stromě implicitních přechodů. Nakonec obsahuje index kořenového stavu. Jeho podobu demonstrujeme příkladem:

$$ab, cd, ef, 1$$

Content labelem je tedy označen nějaký stav. Z tohoto content labelu vyplývá, že:

- Tento stav má explicitní přechody pro symboly a a b.
- Implicitní přechod z tohoto stavu vede do stavu s content labelem *cd,ef,1*, ten má definovány explicitně přechody pro symboly c a d.
- Stav *cd,ef,1* má implicitní přechod do stavu *ef,1*, ten má explicitní přechody definovány pro symboly e a f.
- Kořenem stromu implicitních přechodů je stav s indexem 1. Tento index je zároveň jeho content labelem.



Obrázek 2.2: příklad CD^{2}FA

Znázornění tohoto schématu je na obrázku 2.2. Silné šipky znázorňují implicitní přechody. Je důležité upozornit, že toto schéma je pouze logickým uspořádáním CD^{2}FA. Ve vnitřní struktuře nezná stav svůj vlastní content label. Ve stavu jsou uloženy content labely jiných stavů, a to těch, do kterých z něj existuje explicitní přechod. V následujícím textu je důležité důkladně rozlišovat content label stavu a content labely ve stavu uloženy.

Jsou-li takto uspořádány content labely a známe-li symbol, který bude následovat (Automat v daném kroku tedy pracuje se dvěma vstupními symboly – aktuálním a následujícím.), je možné přeskočit implicitní přechody a tím ušetřit přístupy do paměti.

Předpokládejme například schéma z obrázku 2.2 a dále předpokládejme, že má právě být proveden nějaký (na obrázku neznázorněný) přechod do stavu *ab,cd,ef,1*. Dále má automat k dispozici symbol, který bude následovat. Pokud je to symbol *a* nebo *b*, přejde do tohoto stavu, protože z content labelu lze vyčíst, že v dalším kroku nebude muset být použit implicitní přechod, protože následovat bude symbol, pro který má tento stav definován explicitní přechod. Pokud je to např. symbol *f*, přejde do stavu *ef,1* , pokud je to např. symbol *h*, přejde do kořenového uzlu 1, protože žádný předchozí stav nemá definován explicitní přechod pro symbol *h*. Kořenový uzel musí mít definovány explicitní přechody pro všechny symboly. Takovýmto způsobem se tedy lze vždy vyhnout provádění implicitních přechodů.

2.3 Adresování stavů v CD²FA pomocí hashování

Máme-li docílit toho, aby CD²FA opravdu přistupoval do paměti pro každý vstupní symbol pouze jednou, musí tomu být přizpůsobeno vnitřní adresování stavů. Vezměme za příklad opět stav s content labelem *ab,cd,ef,1*. Je-li automat ve stavu, ze kterého má přejít do *ab,cd,ef,1*, přečte z hlavní paměti tento content label a nyní již má přejít do dalšího stavu bez dalšího přístupu do hlavní paměti. Ale až v této fázi proběhne porovnání následujícího vstupního symbolu s obsahem content labelu a rozhodnutí, do kterého stavu se vlastně ve skutečnosti přejde. Může to být kterýkoli ze stavů: *ab,cd,ef,1*; *cd,ef,1*; *ef,1*; *1*. Tedy všechny tyto stavy musí být adresovatelné, pouze na základě tohoto jednoho content labelu. Toho docílíme použitím hashování.

Předpokládejme, že stavy *ab,cd,ef,1*; *cd,ef,1*; *ef,1*; *1* jsou uloženy postupně na adresách a_1, a_2, a_3, *1*. Cílem bude, aby platilo:

$$hash(ef, 1) = a_1$$
$$hash(cd, a_1) = a_2 \quad \text{a tedy} \quad hash(cd, hash(ef, 1)) = a_2$$
$$hash(ab, a_2) = a_3 \quad \text{a tedy} \quad hash(ab, hash(cd, hash(ef, 1))) = a_3$$

kde *hash* je vhodná hashovací funkce.

Toho lze docílit tak, že vybereme libovolnou známou hashovací funkci, pro každý stav (tj. content label) ji spočítáme a na výslednou adresu stav uložíme. Je ovšem potřeba vyřešit několik problémů, které při tomto přístupu vyvstanou:

1. Content labely jsou různě velké. Není tedy možné je jednoduše umístit za sebe do homogenního pole.
2. Ve stavech je různý počet content labelů. Jak již bylo řečeno, stav neobsahuje svůj content label, nýbrž content labely těch stavů, do kterých z něj vede explicitní přechod. Kořenový stav obsahuje content labely pro každý možný vstupní symbol. Tedy ani pokud by byly všechny content labely stejně velké, nelze jednoduše umístit stavy za sebe do homogenního pole.
3. Není jasné, jak jsou content labely za sebou ve stavu uloženy.
4. Nejsložitější problém je ten, se kterým je nutno se u hashování vypořádat téměř vždy – řešení kolizí. Klasické přístupy k řešení kolizí nejsou v tomto případě vhodné, protože vyžadují více přístupů do paměti a cílem u CD²FA je vystačit s jedním.

2.4 Content labely a řešení adresování

Proměnlivou velikost content labelů a stavů lze vyřešit tak, že se nekořenové stavy rozdělí do skupin. S. Kumar [3] navrhuje dvě třídy content labelů (o velikosti 4, resp. 8 bytů) a maximální počet explicitních přechodů z nekořenového uzlů omezen pěti. Velikost stavu v paměti tedy může nabývat desíti různých hodnot. Minimálně 4 byty (jeden content label o velikosti 4 byty) a maximálně 40 bytů (pět content labelů, všechny o velikosti 8 bytů). Výsledek hashovácí funkce (tedy index stavu) bude potom relativní k začátku každé skupiny stavů. Když budeme znát počet stavů v jednotlivých skupinách a tyto skupiny budou zařazeny za sebou v paměti předem daným způsobem (například vzestupně podle velikosti stavů), je možné na základě výsledku hashovací funkce přesně určit, na které adrese se nachází příslušný stav.

Dále je potřeba přesně adresovat nejen stav, ale i každý content label, který je v tom stavu obsažen. Z content labelu stavu lze odvodit počet content labelů, které daný stav obsahuje. Pokud např. má být proveden přechod do stavu s content labelem *ab,cd,1* (Tento content label je znám dopředu – z předešlého stavu.), je vidět, že z cílového stavu vedou dva explicitní přechody (pro symboly a a b) a tedy obsahuje právě dva content labely.

Pořadí content labelů ve stavu uložených je možné sjednotit s pořadím symbolu v content labelu tohoto stavu v základním tvaru (content labely mají více možných podob – viz níže). Dále je ovšem potřeba znát velikost content labelů v cílovém stavu. Tuto informaci je potřeba v předcházejícím content labelu uchovávat. Tedy v případě, že připustíme jen dvě možné velikosti content labelu, musí být u každého symbolu v content labelu bit (pro více variant content labelů více bitů), který určí velikost cílového content labelu. Content label by tedy vypadal např. takto: $a_1\, b_2, c_1, 1$. Index 1 značí menší, čtyř-bytový content label a index 2 značí větší, osmi-bytový content label.

Tento content label tedy říká, že stav má dva explicitní přechody – pro stav a a b, a tedy tento stav obsahuje dva content labely, a to v daném pořadí. Dále říká, že content label pro stav a má velikost 4 byty a druhý content label – pro stav b – má 8 bytů. Víme tedy přesně, co v paměti kde nalézt – viz obrázek 2.3.

U kořenových stavů je tento problém potřeba řešit jinak – ty nemají v příslušném content labelu informace o své vnitřní struktuře. Je proto nutné, aby byly všechny content labely zarovnány v paměti na maximální velikost content labelu. Pak k nim lze přistupovat jednoduše indexováním.

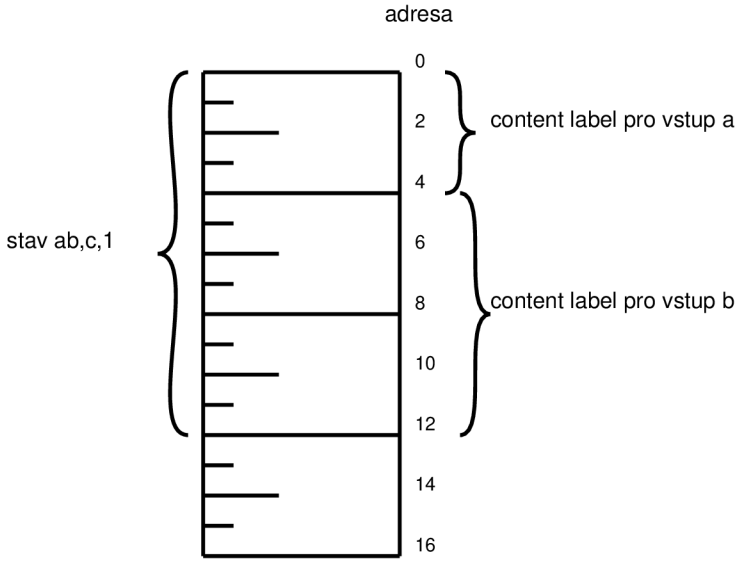
Automat tedy v každém kroku postupuje tímto způsobem:

1. Na začátku kroku je k dispozici content label, v každém kroku je cílem získat content label následující.

2. Z content labelu, který máme k dispozici zjistíme skupinu cílového stavu. Pokud se jedná o kořenový stav, je content labelem jen jeho index. Jinak je skupina definována velikostí stavu, kterou je možné z content labelu odvodit sečtením velikostí jednotlivých content labelů.

3. Získáme adresu stavu cílového stavu. Je-li to kořenový stav, adresou je přímo content label. Jinak adresu získáme pomocí hashování. Adresa se vždy vztahuje k dané skupině stavů.

4. Nechceme ale přistoupit k celému stavu, nýbrž jen k jednomu content labelu (v tomto stavu uloženému), který odpovídá symbolu, který bude následovat na vstupu v příš-



Obrázek 2.3: Uložení stavu $a_1 b_2, c, 1$ v paměti

tím kroku. Jeho pořadí ve stavu, jeho velikost i velikost stavů před ním lze vyčíst z předešlého content labelu.

Dále musí každý symbol obsahovat bit, který určí ke kterému stavu patří (to, co je v textu symbolizováno čárkou). Tedy že v content labelu *ab,cd,e,1* je to ten stav samotný, který má explicitní přechody pro symboly a a b; je to jeho rodič ve stromu implicitních přechodů, který má explicitní přechody pro stavy c a d atd.

Dále je nutné uchovávat informace o koncových stavech. Každý symbol má bit, který určuje, jestli je příslušný stav koncový.

Zbývá vyřešit problematiku kolizí v hashování. Navržené vylepšení této metody je uvedeno v kapitole 3, zde je popis původního řešení. Hashování je nutno řešit pouze pro nekořenové stavy. Kořenové stavy jsou uloženy v tabulce a jejich index přímo odpovídá jejich content labelu.

Princip řešení kolizí spočívá v tom, že budeme hledat *perfektní hashování* – takové, kde ke kolizím vůbec nedochází. Pro jeden stav, typicky existuje více možných podob content labelu – *kandidátní jména*, ty musí zachovat stejný význam. V content labelu je irelevantní pořadí symbolů, které náleží jednomu stavu. Dále má stav maximálně jeden explicitní přechod pro jeden symbol, tedy je možné opakovat symboly bez změny významu content labelu. V důsledku mají všechny následující content labely stejný význam:

ab, c, 1 *ba, c, 1* *aab, cc, 1* *bbab, c, 1*

Ovšem z hlediska hashovací funkce nejsou tyto content labely stejné – jsou to různé řetězce. Pokud tedy nastane kolize, je možné změnit content label, bez změny jeho významu a získat nový index. Pokud by takové řešení nebylo dostatečné, tj. stále by nebylo možné sestavit hashovací tabulku bez kolizí, přichází v úvahu další dvě řešení:

1. Přidat do content labelů další bity – tzv. *diskriminátory*, které nebudou mít vliv na jeho význam, pouze dále rozšíří množinu kandidátních jmen.

2. Zvětšení hashovací tabulky.

V obou případech se ovšem jedná o neužitečné místo, které zvyšuje paměťové nároky automatu.

Postup při hledání perfektního hashování je tedy následující:

1. Všechny nekořenové stavy rozděl do skupin podle jejich velikosti – tedy paměti, kterou stav zabere. Ta závisí na počtu content labelů, které stav obsahuje a jejich velikosti.

   Pro každou skupinu se dále hledá perfektní hashování:

2. Pro všechny content labely ze skupiny vygeneruj všechny kandidátní jména a použij je jako vstup do hashovací funkce – výsledek je tedy množina kandidátních indexů pro každý content label.

3. Sestroj bipartitní graf $G(V_1 + V_2, E)$, kde množina vrcholů V_1 odpovídá množině content labelů a V_2 odpovídá všem kandidátním indexům. Množina hran E zahrnuje všechny hrany (u, v) takové, že $u \in V_1, v \in V_2$ a v je kandidátní index u.

4. Pokus se najít pefektní párování grafu.

5. Pokud je perfektní párování nalezeno, přiřaď content labelům jejich kandidátní jména podle tohoto párování

6. Jinak, pokud perfektní párování nelze nalézt, zvětši maximální hodnotu diskriminátoru nebo zvětši hashovací tabulku a pokračuj bodem 2.

2.5 Princip konstrukce CD^{2}FA

Při transformaci DFA na CD^{2}FA se nejprve vytvoří D^{2}FA. To ale není nic jiného, než určení, kde budou zavedeny implicitní přechody. V dalším kroku se D^{2}FA transformuje na CD^{2}FA. Tedy vygenerují se content labely a připraví struktura uložení stavů pro hashování.

Je zřejmé, že pro každý DFA lze vytvořit více než jeden ekvivalentní D^{2}FA resp. CD^{2}FA. Cílem ale je sestrojit co možná nejlepší CD^{2}FA. Nejlepší automat zde znamená automat s nejmenšími paměťovými nároky, ale za zachování rychlosti zpracování vstupu (tedy zachování pravidla, že pro jeden vstupní symbol se provede jedno načtení z hlavní paměti) a samozřejmě za zachování ekvivalence s původním DFA.

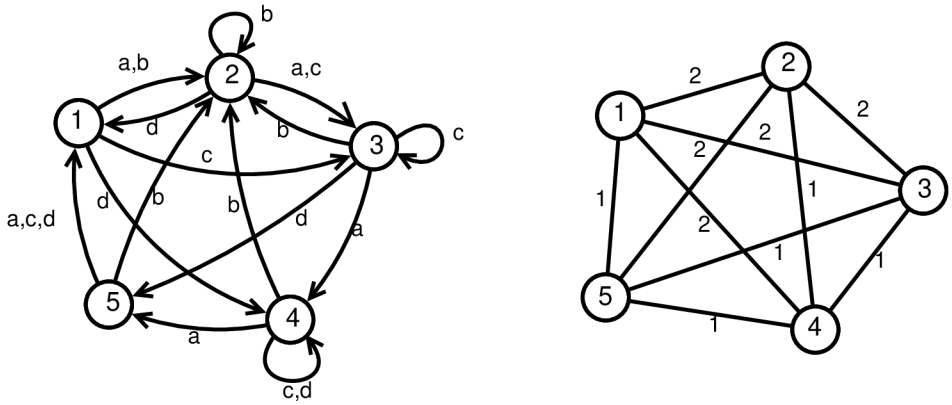
Pokud je cílem sestavit CD^{2}FA, který bude co nejméně paměťově náročný, je potřeba brát v úvahu následující faktory:

- Nejdůležitějším faktorem je počet přechodů, které se podaří odstranit tím, že zavedeme implicitní přechody.

- Je žádoucí mít mělké stromy implicitních přechodů. Prodlužuje-li se cesta od nějakého stavu ke kořenovému stavu, zvětšuje se příslušný content label, protože ten vždy obsahuje i content label všech svých předků ve stromě implicitních přechodů.

- Pokud omezíme velikost content labelů, bude potřeba dodržovat určitou maximální hloubku stromu implicitních přechodů, dále je třeba omezit počet explicitních přechodů vedoucích z jednoho stavu. Toto vše se totiž musí do content labelu vejít.

Pro vytvoření CD^{2}FA s vhodnými vlastnostmi navrhl S. Kumar [3] algoritmus CRO (creation, reduction, optimization), který probíhá ve třech fázích.



Obrázek 2.4: Vytvoření grafu společných přechodů z DFA

2.5.1 Konstrukční fáze

Aby bylo eliminováno co nejvíce přechodů, je potřeba vytvořit implicitní přechod mezi takovými stavy, které mají co možná nejvíce společných explicitních přechodů. Společným přechodem se rozumí takový přechod, který má stejný cíl a provede se při stejném vstupu. Proto je nejdříve sestaven *graf společných přechodů* (space reduction graph). Jeho vrcholy tvoří stavy původního DFA, mezi každými dvěma vrcholy je hrana jejíž hodnota odpovídá počtu společných přechodů. Příklad grafu společných přechodů sestaveného z DFA je na obrázku 2.4 Na základě tohoto grafu je následně vytvořen tzv. *les implicitních přechodů* (spanning forest). Je to množina stromů, jejichž větve jsou tvořeny implicitními přechody. Jedná se o stromy, protože, jak bylo řečeno v části 2.2.1, je nutné zamezit smyčce implicitních přechodů. Tyto stromy jsou konstruovány použitím varianty Kruskalova algoritmu [1].

Tyto hrany jsou následně procházeny sestupně od těch s nejvyšší hodnotou. U každé hrany je vyhodnoceno, jestli ji lze přidat do lesa implicitních přechodů. To lze tehdy, pokud jsou splněna dvě pravidla.

1. Nevznikne smyčka implicitních přechodů
2. Nevznikne strom s průměrem větším, než 2.

Druhá z podmínek zaručí, že pokud je vždy vhodně zvolen kořen stromu, jeho hloubka nikdy nepřesáhne hodnotu jedna a tedy nejdelší možná cesta implicitních přechodů má délku jedna.

Dále je vhodné, aby kořeny stromu byly ty stavy, do kterých vede více explicitních přechodů. Tomu lze napomoci tak, že hrany grafu společných přechodů budou, kromě počtu společných přechodů, uchovávat ještě sekundární hodnotu. Tou bude počet explicitních přechodů, které vedou do jednoho ze stavů této hrany. Pokud bude více hran grafu společných přechodů mít stejnou primární hodnotu, přednost dostanou ty s vyšší sekundární hodnotou.

2.5.2 Fáze redukce

Během fáze redukce usilujeme o zvýšení váhy lesa implicitních přechodů. Váha lesa implicitních přechodů je součtem váhy všech jeho stromů, přičemž váha stromu je součtem všech vah jeho větví (implicitních přechodů) a váha větve je rovna hodnotě příslušné hrany v grafu společných přechodů.

Stromy jsou zkoumány sestupně podle váhy následujícím způsobem: strom je zrušen a každý jeho uzel je připojen k nějakému jinému stromu, tedy k nějakému kořenovému uzlu (aby byla zachována hloubka stromů rovna jedné) tak, aby výsledná nová podoba lesa implicitních přechodů měla co největší váhu. Pro každý uzel zkoumaného stromu je

tedy potřeba porovnat váhy hran mezi tímto uzlem a každým kořenovým uzlem a zvolit tu nejvyšší. Pokud je výsledná váha lesa vyšší, než před zrušením daného stromu, je tato změna zachována a proces zkoumání stromů se spustí znovu od začátku. Jinak je zachována původní podoba lesa. Proces skončí tehdy, až jsou prozkoumány všechny stromy a není provedena žádná změna.

2.5.3 Fáze optimalizace

I po skončení fáze redukce mají všechny stromy maximální hloubku jedna. Ve fázi optimalizace se některé stromy prohloubí, pokud to bude znamenat celkovou úsporu paměti. Tato fáze probíhá až poté, co jsou vygenerovány a přiřazeny content labely.

Nejprve se pro každý stav spočítá velikost content labelů pro ty přechody, které vedou do tohoto stavu (S_{in}) a velikost content labelů pro ty přechody, které vedou z tohoto stavu (S_{out}). Vhodní kandidáti na prodloužení cesty implicitních přechodů jsou ty stavy, u kterých je vysoká hodnota $S_{out} - S_{in}$. Poté se postupuje od stavů, které mají tuto hodnotu nejvyšší a zkoumá se, jestli se připojením k nějakému stavu, který je přímým potomkem kořenového stavu, ušetří paměť (tzn. content labely budou ve výsledku obsahovat méně symbolů). Pokud ano, je původní stav připojen k novému stavu. Algoritmus pokračuje, dokud není po přezkoumání všech uzlů graf beze změn.

2.6 Optimalizace content labelů

Kořenové stavy mají explicitně definovány přechody pro všechny symboly vstupní abecedy automatu. Pro tyto stavy zavedeme tzv. *obvyklý stav*. Obvyklý stav pro daný kořenový stav, je ten, který je nejčastějším cílem přechodů kořenového stavu. Pokud by takových bylo více, zvolí se jeden z nich. Zavedeme abecedu kořenových stavů – to je množina symbolů, pro které jsou v nějakém kořenovém stavu explicitně definovány přechody. Na začátku je shodná se vstupní abecedou automatu. Pokud explicitně definujeme pro každý kořenový stav jeho obvyklý stav, je možné všechny původní přechody z kořenových stavů do příslušných obvyklých stavů z paměti vypustit – redukovat abecedu pro daný kořenový stav.

Pro každý kořenový stav se takto redukuje vstupní abeceda. Pokud se toto provede s každým kořenovým stavem a u všech je redukována vstupní abeceda, sjednocením všech množin redukovaných vstupních abeced pro všechny kořenové stavy lze získat novou (redukovanou) abecedu kořenových stavů. Jinými slovy, pokud jsou některé symboly při redukci odstraněny u každého kořenového stavu, lze je odstranit i z abecedy kořenových stavů.

Dále zavedeme abecedu nekořenových stavů. To je množina symbolů, pro které je explicitně definován přechod v některém nekořenovém stavu.

Sjednocením kořenové a nekořenové abecedy vznikne nová – redukovaná – vstupní abeceda. Pokud tato abeceda je redukována natolik, že klesne počet bitů, které jsou nutné pro reprezentaci symbolu, vytvoříme *tabulku pro překlad symbolů*. V opačném případě není důvod redukovanou abecedu zachovat. Tabulka pro překlad symbolů obsahuje tolik záznamů, kolik symbolů měla původní abeceda – tedy pravděpodobně 256 záznamů. Pro ty symboly, které jsou obsaženy v redukované vstupní abecedě, obsahuje tabulka překlad. Pro ostatní symboly obsahuje unikátní symbol.

2.6.1 Konečný obsah content labelu

Pokud předpokládáme, že redukcí vstupní abecedy se podaří uspořit bit v reprezentaci každého symbolu, bude tedy symbol reprezentován sedmi bity. Dále předpokládáme, že nebude více kořenových stavů, než lze reprezentovat deseti bity. Každý symbol musí dále obsahovat:

- velikost cílového content labelu – 1 bit
- označení, ke kterému stavu ve stromu implicitních přechodů náleží – 1 bit
- určení, zda je příslušný stav konečný – 1 bit
- diskriminátor – 0 bitů nebo více

Tedy symbol je reprezentován minimálně deseti bity a kořenový stav také deseti bity. Za těchto okolností je tedy možné uložit content label obsahující maximálně dva symboly do paměti o velikosti čtyři byty a content label o maximálně pěti symbolech do paměti o velikosti osm bytů.

Kapitola 3

Vylepšení konstrukce CD^{2}FA

3.1 Příprava hashování

V kapitole 2.4 bylo napsáno, jakým způsobem lze zařídit perfektní (bezkolizní) hashování content labelu tak, aby výsledkem vždy byla správná adresa stavu, který patří k tomuto content labelu. Princip spočíval ve vygenerování všech kandidátních jmen pro všechny content labely ve skupině, následně byly spočítány výsledky hashovací funkce pro všechna tato jména – kandidátní adresy. Dále byla hledána kombinace těchto kandidátních adres taková, aby v jejich hashování nikdy nenastala kolize. Zde bude prezentováno vylepšení tohoto postupu. Vylepšení spočívá v tom, že není nutné generovat všechny kandidátní jména a počítat pro všechna tato jména hashovací funkci. Algoritmus předpokládá, že máme k dispozici dvě funkce (jejich realizace je diskutována níže 3.2.2 a 3.2.1):

1. *next(CN)*

   Zavedení této funkce způsobí to, že množina kandidátních jmen se stane ordinální. Tedy pro každé kandidátní jméno *CN* lze jednoznačně určit, které jméno je další v pořadí a žádné není vynecháno.

2. *first(CN)*

   Tato funkce získá z libovolného kandidátního jména *CN* jeho původní (první) tvar.

Nyní, máme-li připravit hashování pro skupinu o n content labelech (první až n-tý content label), použijeme dvě pole A a B, obě o velikosti n. Pole A uchovává aktuální kandidátní jména pro content labely. Pole B indikuje, zda je již daný index obsazen, nebo nikoli – položky tedy budou pouze dvoustavové (volno/obsazeno). Na začátku je pole A prázdné a všechny položky pole B jsou ve stavu volno. Dále mějme index i, který určuje aktuální položku pole A, na začátku je $i = 1$. Algoritmus probíhá takto:

opakuj, dokud pole A není naplněno:

1. najdi první další vhodné kandidátní jméno pro i-tý prvek
2. pokud se podařilo, označ příslušnou hodnotu v poli B jako obsazenou; inkrementuj i
3. jinak, pokud se nepodařilo, uvolni i-tý prvek pole A, a příslušný prvek v poli B označ jako volno; dekrementuj i
4. pokud je $i < 1$ neexistuje perfektní hashování – skonči s neúspěchem

Příslušný prvek v poli B i-tému prvku v poli A je prvek $B[j]$ kde:

$$j = hash(A[i]) \bmod n$$

První další vhodné kandidátní jméno je takové jméno, které získáme aplikováním funkce $CL = next(CL)$ tak dlouho, dokud příslušná položka v poli B není volná. Pokud je položka v poli A prázdná, použije se první kandidátní jméno (funkce *first*).

Jedná se o jistou obdobu prohledávání stavového prostoru metodou Backtracking. Je ovšem třeba mít funkce *first* a *next*.

3.2 Funkce *first*, funkce *next* a její části

Jak bylo uvedeno v kapitole 2.4, máme k dispozici tři způsoby, jak změnit výsledek hashovací funkce pro nějaký content label a přitom nezměnit jeho význam: záměna pořadí symbolů, opakování nějakého symbolu a změna diskriminátoru. Také lze tyto možnosti kombinovat.

Pokud budeme takto manipulovat s nějakým content labelem, jedná se vždy jen o jednu jeho část, která reprezentuje explicitní přechody pro jeden stav. Například, máme-li content label *abc,de,1*, budou funkce aplikovány jen na část *abc*. Část *de,1* je pouze obsah content labelu rodiče tohoto stavu ve stromě implicitních přechodů. Jedná se tedy o práci s klasickými řetězci.

3.2.1 Funkce *first*

Definujme první (základní) kandidátní jméno jako to jméno kde:

1. Symboly jsou seřazeny dle ordinální hodnoty vzestupně.
2. Žádný symbol se neopakuje.
3. Diskriminátor je roven nule.

Funkce *first* jednoduše zajistí tyto tři body – seřadí symboly v řetězci, odstraní duplicitní symboly a vynuluje diskriminátor.

3.2.2 Funkce *next*

Funkce *next* je klíčová část celého algoritmu. Na základě vstupního kandidátního jména se vygeneruje další, a to deterministicky, a také tak, aby, se prošla celá množina kandidátních jmen, bude-li třeba.

Funkce *next* má k dispozici tři funkce – *permute*, *pad* a *discriminate*.

Nejprve se pokusí získat další kandidátní jméno pomocí funkce *permute*. Ta vytvoří nové kandidátní jméno záměnou pořadí prvků, pokud je to možné. Není to možné tehdy, pokud symboly v řetězci jsou seřazeny sestupně (tedy opačně, než u prvního kandidátního jména) – pak už byly všechny permutace symbolů použity.

Pokud neuspěje funkce *permute*, pokusí se získat nové kandidátní jméno funkce *pad*, která vhodně zduplikuje nějaký prvek. Řetězec vrátí seřazený (základní tvar z hlediska funkce *permute*), aby mohla později být na řetězec znovu aplikována funkce *permute*. Funkce neuspěje tehdy, pokud již má kandidátní jméno maximální možnou délku.

Pokud neuspěje ani funkce *pad*, pokusí se získat nové kandidátní jméno funkce *discriminate*, která inkrementuje diskriminátor, pokud nedosáhl maximální velikosti. Řetězec vrátí

seřazený a bez duplicitních znaků, aby na něj mohli později být znovu aplikovány funkce *permute* a *pad*. Pokud by selhala i tato funkce, není možné vygenerovat další kandidátní jméno za současných podmínek.

3.2.3 Funkce *permute*

První (základní) kandidátní jméno je lexikograficky nejmenší - symboly jsou seřazeny vzestupně podle ordinální hodnoty. Aby funkce *permute* zajistila, že nebude při jejím postupném volání vynecháno žádné kandidátní jméno, je nutné zajistit, aby vrátila (lexikograficky) nejmenší kandidátní jméno, které je větší než vstupní kandidátní jméno. Toho lze dosáhnout následujícím postupem:

1. Analyzuj řetězec zprava doleva. Najdi první takovou dvojici sousedících symbolů, kde levý je menší, než pravý. Pokud taková dvojice není, skonči s neúspěchem – řetězec je seřazen opačně.

   Tedy hledáme významově nejmenší symbol, který lze zvýšit. Nazvěme levý symbol z této dvojice aktuálním.

2. Zaměň aktuální symbol za nejmenší z těch, které jsou od něj vpravo.

   Tedy chceme jej zvýšit nejméně, jak to lze (je jisté, že všechny symboly vpravo od aktuálního jsou větší).

3. Seřaď všechny symboly vpravo od místa, kde byl aktuální symbol, vzestupně.

   Zvýšení lexikografické hodnoty zajistila výměna aktuálního symbolu za symbol větší, zbytek řetězce tedy musí být co možná nejmenší.

3.2.4 Funkce *pad*

Tato funkce přijímá kromě kandidátního jména ještě jeden vstup – maximální délku kandidátního jména. To se musí vejít do místem omezeného content labelu. Funkce *pad* nebere v úvahu pořadí symbolů – vždy vrátí kandidátní jméno v seřazené podobě. Důležité tedy je pouze to, kolikrát se který symbol v řetězci opakuje. Nejprve se tedy sestaví tabulka četnosti znaků T. Například pro řetězec *abaaccd* by $T = 3, 1, 2, 1$ postupně pro znaky *a*, *b*, *c*, *d*. Cílem je vygenerovat řetězec za použití stejných znaků tak, aby: 1. celkový počet znaků se nezvýšil, není-li to nutné (pochopitelně se nemůže nikdy snížit). 2. lexikografická hodnota řetězce byla co nejnižší, ale větší než původně. Toho lze dosáhnout následujícím způsobem:

1. Analyzuj T zprava, ale přeskoč první (nejpravější) hodnotu. Najdi první hodnotu $T[i]$ větší než jedna.

   Tedy hledáme lexikograficky nejméně významný znak, abychom zmenšili jeho četnost. Nesmí ovšem zmizet – tj. četnost nemůže klesnout na nulu.

2. Není-li žádná nalezena, zvětši počet znaků: $T[1] = T[n] + 1; \qquad T[n] = 1$ a skonči. Pokud nelze dále zvýšit počet znaků, protože by se přesáhla maximální délka řetězce, skonči s neúspěchem

   Zvětšíme-li počet znaků, je zaručeno, že řetězec je použit poprvé – opakujeme nejmenší (první) znak.

3. Jinak dekrementuj četnost i-tého znaku T[i] a inkrementuj četnost následujícího znaku $T[i+1]$

   Zmenšíme četnost nejméně významné možné hodnoty.

4. je-li poslední hodnota $T[n] > 1$, pak tento přebytek přičti do $T[i+1]$, tedy: $T[i+1] = T[n] - 1; \qquad T[n] = 1$

3.2.5 Funkce *discriminate*

Funkce seřadí řetězec podle velikosti, odstraní duplicitní znaky a inkrementuje diskriminátor.

Kapitola 4

Implementace konstrukce CD^{2}FA

Automat CD^{2}FA je implementován v jazyce Python a zakomponován do knihovny Netbench. Implementace je realizována ve třídě *cddfa*, která dědí ze třídy *b_dfa*. To jí umožňuje použít metody třídy *b_dfa* pro načtení automatu – metoda *create_by_parser*, převedení obecného automatu na automat deterministický – metoda *determinise* a převedení deterministického konečného automatu na automat minimální – metoda *minimise*. Třída *cddfa* dále implementuje metodu *_make_cddfa*, která převede deterministický konečný automat na CD^{2}FA. Tyto tři metody jsou sdruženy v metodě třídy cddfa *compute*.

Dále tato třída implementuje metodu *run*, která provede simulaci přijímání řetězce automatem CD^{2}FA. Jejím jediným vstupním argumentem je daný řetězec, výstupem je logická hodnota – *True*, pokud byl řetězec přijat, jinak *False*.

Dále třída implementuje metodu *report_memory_usage*, která zobrazí následující statistiky:

- Počet stavů a přechodů před transformací DFA na CD^{2}FA
- Počet bytů, které jsou potřeba k uložení původního DFA
- Počet bytů, které jsou potřeba k uložení sestaveného CD^{2}FA a poměr mezi touto hodnotou a původními paměťovými nároky
- Počet stromů implicitních přechodů v CD^{2}FA

4.1 Popis implementovaných tříd

Třída *cCL*

Tato třída implementuje content label. Její součástí jsou tato data:

- seznam řetězců (*CL*) – jsou to řetězce symbolů, které definují explicitní přechody pro jednotlivé stavy ve stromě implicitních přechodů. Pro content label kořenového stavu je seznam prázdný.
- Index kořenového uzlu stromu implicitních přechodů (*root*).
- Seznam logických hodnot 0/1, který indikuje, které stavy na cestě od daného stavu ke kořenovému stavu jsou přijímací (*fin_state*).
- Diskriminátor (*disc*)

- Původní délku content labelu (*length*), tedy délku před tím, než mohla být ovlivněna funkcí pad (3.2.4).
- Ukazatel na stav, ke kterému content label patří (*state*)

Dále implementuje třída metodu *get_size*. Ta spočítá, kolik paměti bude content label potřebovat, vzhledem k navrženému zarovnávání paměti na 4 byty, nebo 8 bytů.

Třída *cTree*

Tato třída implementuje strom implicitních přechodů. Z dat obsahuje pouze ukazatel na kořen (*root*) a váhu stromu (*weight*). Uzly jsou implementovány třídou *cNode*. Metody jsou zpravidla pouze delegovány kořenovému uzlu a tedy řešeny ve třídě *cNode*.

Třída *cNode*

Tato třída reprezentuje uzel stromu implicitních přechodů. Obsahuje číslo stavu (*state_num*), seznam ukazatelů na sousedící uzly (*others*) a ukazatel na rodiče ve stromě (*parent*). Implementuje tyto metody, které jsou všechny realizovány rekurzivně:

- *find* – nalezne ve stromu prvek podle jeho čísla, vrátí ukazatel nebo *None*, pokud neuspěje
- *get_diameter* – spočítá průměr stromu
- *establish_parents* – inicializuje ukazatele na rodiče ve stromě
- *add_edge* – přidá větev do stromu
- *tree_to_list* – vrátí seznam čísel uzlů ve stromě
- *print_tree* – vytiskne strukturu stromu – slouží pro ladící účely

Třída *cSRG_edge*

Reprezentace jedné hrany grafu společných přechodů. Obsahuje čísla stavů, které spojuje (*state1* a *state2*) a hodnotu hrany (*value*). Dále implementuje metodu *get_indegree*, která vrátí počet explicitních přechodů vedoucích do některého z uzlů hrany.

Třída *cState*

Třída reprezentuje stav automatu. Obsahuje tyto datové položky:

- *num* – číslo stavu
- *indegree* – počet přechodů původního DFA, které vedly do tohoto stavu
- *tree* – ukazatel na strom implicitních přechodů (*cTree*), ve kterém se stav nachází
- *CLW* – content label stavu (*cCL*)
- *CLs* – seznam content labelů (*cCL*), které mají být ve stavu obsaženy
- *final_index* – index stavu ve skupině stavů

- *group_index* – skupina stavů, kořenové stavy mají skupinu 0, ostatní podle svojí velikosti (tj. podle počtu a velikosti content labelů v *CLs*)
- *size* – budoucí velikost stavu v paměti i s přihlédnutím k zarovnání v paměti
- *CL_size_in* – velikost content labelů, které vedou do tohoto stavu
- *CL_size_out* – velikost content labelů, které vedou z tohoto stavu
- *parent_state* – stav, který je rodičem ve stromě implicitních přechodů

Dále třída implementuje metody *is_root*, která vrací logickou hodnotu v závislosti na tom, jestli je stav kořenem stromu, *get_size*, která vrátí velikost stavu a *get_distance_to_root*, která vrátí délku cesty od stavu ke kořeni stromu.

4.2 Popis třídy *cddfa*

Třída *cddfa* je implementací CD^{2}FA. Obsahuje tyto datové položky:

- *states* – seznam stavů (*cState*)
- *final_states* – seznam koncových stavů
- *starting_state* – počáteční stav
- *alphabet* – seznam symbolů vstupní abecedy
- *transitions* – slovník přechodů, klíčem je výchozí stav a symbol, hodnotou je následující stav
- *roots* – počet kořenových stavů
- *non_roots* – počet nekořenových stavů
- *state_offsets* – seznam indexů v paměti, kde začínají jednotlivé skupiny stavů. Skupina kořenových stavů začíná na indexu 0 (ten není v seznamu – je implicitní), skupina jedna na indexu *0 + počet kořenových stavů*, atd.
- *use_alt_hash_constr* – určuje, jaká funkce se má použít pro přípravu hashování

4.2.1 Metody pro konstrukci automatu

CD^{2}FA je konstruován z DFA ve funkci *_make_cddfa*. V ní jsou postupně volány funkce, které realizují proces konstrukce.

_create_space_reduction_graph

Funkce analyzuje všechny dvojice stavů a pro každou dvojici spočítá, kolik společných přechodů tyto stavy mají. Tj. kolik přechodů má z obou stavů stejný cílový stav, pokud je přijat stejný symbol na vstupu. Pro každou dvojici stavů je vytvořen objekt *cSRG_edge* – hrana grafu společných přechodů. Seznam těchto hran je (primárně) seřazen sestupně podle hodnoty hrany a (sekundárně) podle počtu přechodů, které do dvou daných stavů ústí. Takto seřazený seznam hran reprezentuje graf společných přechodů.

Pokud je potřeba brát v úvahu hrany pouze od určité hodnoty, lze to specifikovat konstantou MAX_CL_LEN. Hrany s nižší hodnotou v grafu potom nebudou. V našem případě je tato hodnota nastavena na 5. To ve výsledku způsobí, že content labely nebudou obsahovat více než 5 symbolů.

Graf společných přechodů, resp. seznam hran grafu je výstupem funkce.

_create_spanning_forest

Funkce realizuje konstrukční fázi CD^{2}FA (2.5.1) – analyzuje seznam hran (tak jak jsou seřazeny z předchozí funkce) grafu společných přechodů a sestavuje je do stromů následovně:

1. Pokud není ani jeden ze stavů hrany v nějakém stromě, tato hrana vytvoří nový strom.
2. Pokud je jen jeden uzel ve stromě, připoj k tomuto stromu tuto hranu, pokud by vzniklý strom neměl průměr větší než 2, jinak nedělej nic.
3. Pokud jsou oba uzly v nějakém stromě, nedělej nic – pokud by to byl stejný strom, vznikla by smyčka. Pokud by to byl jiný strom, stromy by se spojily a výsledný strom by měl příliš velký průměr (tj. větší než 2).

Takto je vytvořen seznam stromů (*cTree*) – les implicitních přechodů (*spanning_forest*), který je výsledkem funkce. Ještě je nutno vzít v úvahu, že kvůli omezení na minimálné hodnotu hrany v předchozí funkci se může stát, že některé stavy nebudou obsaženy v žádném stromě. Tyto stavy pak musí vytvořit jednoprvkové stromy a být přidány do lesa.

_spanning_forest_reduction

Funkce realizuje redukční fázi CD^{2}FA (2.5.2). Nejprve přiřadí všem stromům váhu, tj. součet vah všech větví, seřadí les podle vah stromů a spočítá celkovou váhu lesa. Poté postupuje od stromů s největší vahou, odečte váhu aktuálního stromu od lesa, najde pro každý uzel z tohoto stromu nový kořenový uzel ze všech ostatních stromů a připočte váhy těchto nových větví k lesu. Pokud se celková váha lesa zvýšila, změna je zachována, jinak je vrácena zpět.

Takto se postupuje tak dlouho, dokud nejsou všechny stromy analyzovány beze změny. Vstupem i výstupem funkce je les implicitních přechodů (*spanning_forest*), který je ovšem na výstupu redukovaný. Maximální průměr stromu 2 je zachován, ale stromy jsou „košatější“ (z kořene vede více větví) a je jich méně.

_generate_content_labels

V této funkci je každému stavu přiřazen content label. Kořenovým stavům jsou jednoduše přiřazeny indexy. U nekořenových stavů jsou porovnány jejich explicitní přechody s explicitními přechody kořene stromu, do kterého patří a do jejich content labelu (*CLW – content label wrapper*) jsou přidány ty symboly, u kterých se liší cílový stav. Poté je také inicializován řetězec konečných stavů (*fin_state*).

_spanning_forest_optimization

Tato funkce odpovídá optimalizační fázi CD^{2}FA (2.5.3). V první fázi jsou každému stavu přiřazeny hodnoty *CL_size_in* – velikost content labelů, které vedou do stavu – a *CL_size_out* – velikost content labelů, které vedou ze stavu. Stavy jsou seřazeny sestupně podle hodnoty $CL\_size\_in - CL\_size\_out$, v tomto pořadí se budou analyzovat.

Každý nekořenový stav je poté zkušebně připojen k ostatním stavům, které mají vzdálenost od kořene 1. Pokud některá z těchto změn vede k celkové úspoře paměti, tj. k menším relevantním content labelům, je provedena. O tuto změnu se postará funkce *_reattache_state*, která přijímá na vstupu dva stavy, které spojí. Přegeneruje příslušné content labely a změní strukturu stromů. Rekurzivně je volána i pro všechny potomky ve stromě, aby i pro ně byly správně přegenerovány content labely.

_assign_content_labels

V této funkci jsou přiřazeny content labely do těch stavů, ve kterých mají být skutečně uloženy (položka *CLs*). Tedy jsou přiřazeny do těch stavů, ze kterých vede explicitní přechod do stavu s daným content labelem.

_optimize_content_labels

Tato funkce provádí redukci vstupní abecedy (2.6). Nejprve volá funkci *_reduce_root_alphabet*, která provede redukci abecedy kořenových uzlů. Ta pro kořenový stav vyhodnotí, který stav je pro něj stavem obvyklým. Všechny symboly, které vedou do jiného než obvyklého stavu jsou přidány do abecedy kořenových uzlů (která je na začátku prázdná). Takto se postupuje u všech kořenových uzlů.

Poté je volána funkce *_reduce_non_root_alphabet*, která provede redukci abecedy nekořenových uzlů. Ta jednoduše u těchto uzlů akumuluje symboly, pro které jsou v nich definovány explicitní přechody a výsledkem je abeceda nekořenových uzlů.

Tyto dvě abecedy jsou poté sjednoceny do redukované vstupní abecedy. Výsledkem funkce je hodnota *bits_for_symbol*. Ta se rovná $\lceil \log_2 n \rceil$, kde n je počet symbolů v redukované abecedě. Tato hodnota udává, kolik bitů je potřeba na vyjádření jednoho symbolu v content labelu.

Dále funkce spočítá hodnotu *bits_for_root*, která se rovná $\lceil \log_2 n \rceil$, kde n je počet kořenových uzlů. Tato hodnota udává, kolik bitů je potřeba na vyjádření indexu kořenového uzlu v content labelu.

_prepare_hashing

Tato funkce připraví content labely a pořadí stavů v paměti tak, aby bylo možné používat hashování bez kolizí. Pracuje s nekořenovými stavy. Ty nejprve rozdělí do skupin, podle jejich velikosti – tj. podle sumy velikostí content labelů v nich uložených. Poté volá pro každou skupinu funkci *_prepare_hashing_for_group*, která realizuje přípravu perfektního hashování původním algoritmem popsaným v části 2.4 nebo funkci *_prepare_hashing_for_group_alt*, která realizuje přípravu hashování novým způsobem popsaným v kapitole 3. Oba způsoby využívají funkci *_get_next_candidate_label*, která je implementací funkce *next* (3.2.2). Dále jsou zde implementovány další již diskutované funkce:

- *_get_next_candidate_label_by_permuting* – implementace *permute* (3.2.3)
- *_get_next_candidate_label_by_padding* – implementace *pad* (3.2.4)
- *_get_next_candidate_label_by_discriminator* – implementace *discriminate* (3.2.5)
- *_get_basic_form_of_CL* – implementace *first* (3.2.1)

4.3 Simulace běhu automatu

K simulaci běhu automatu slouží funkce *run*, která musí být volána až po transformaci DFA na CD^{2}FA. Jako argument vyžaduje vstupní řetězec a její výsledek je logická hodnota *True* – řetězec byl automatem přijmut, nebo *False* – řetězec nebyl přijmut.

Problém, který nebyl diskutován v článku S. Kumara [3], je první krok automatu. Problém spočívá v tom, že konstrukce automatu nebere v potaz to, který stav automatu je počáteční. CD^{2}FA je navržen tak, že vždy přechází do stavu, ze kterého je definován explicitní přechod pro ten symbol, který bude následovat. Ovšem pro počáteční stav toto platit nemusí.

Pokud by např. byl počáteční stav *ab,1*, a první symbol byl *c*, automat nebude fungovat. Začíná ve stavu, ze kterého umí provést jen přechod pro symbol *a* nebo symbol *b*. Stav dále nezná svůj vlastní content label, takže ani nebude vědět, který stav je kořenem stromu, do kterého patří. Dále ani není možné identifikovat, že tento problém nastal – počáteční stav neví pro které symboly má definovány explicitní přechody a kde jsou v něm uloženy – to všechno jsou informace, které automat jinak získává z předešlého content labelu. Proto nestačí ani přidat počátečnímu uzlu informaci o tom, který uzel je kořenem příslušného stromu.

Východiskem je, při konstrukci automatu dbát na to, aby počáteční uzel byl vždy uzlem kořenovým v nějakém stromě implicitních přechodů. Ten pak obsahuje explicitní přechody pro všechny symboly (i když některé s pomocí svého obvyklého přechodu, viz 2.6) a daný problém nenastane.

Samotný algoritmus simulace probíhá následovně:

1. Za aktuální stav označ počáteční stav

   Opakuj, dokud není zpracován celý vstupní řetězec

2. Načti aktuální symbol a, pokud je to možné, načti následující symbol

3. Z aktuálního stavu načti content label, který odpovídá aktuálnímu vstupnímu symbolu.

4. Pokud je aktuální symbol posledním (nebylo možné načíst následující symbol v kroku 2), přečti z aktuálního content labelu (načteného v kroku 3) informaci o tom, jestli je daný stav koncový (položka *fin_state*). Pokud ano, ukonči funkci s výsledkem *True*, jinak s výsledkem *False*.

5. Najdi, zda se v aktuálním content labelu nachází následující symbol, pokud ne, přejdi do kořenového stavu, uvedeného v aktuálním content labelu (položka *root*) a pokračuj bodem 2, pokud je tam uveden vícekrát, vezmi ten nejlevější (nejdále od kořene)

6. Prováděj hashovací funkci, jejíž vstup je nejpravější část content labelu a rodičovský uzel, a to tak dlouho, dokud nebude vstupem ten symbol, který byl nalezen v content labelu v kroku 5. Poté proveď hashovací funkci naposledy.

7. Aktualizuj aktuální stav. Ten je určen skupinou stavu (tu lze vyčíst z content labelu) a jeho indexem, který se získá z hashování. Pokračuj bodem 2.

Bod 6 tedy může vypadat takto: Aktuální content label je *ab,cd,e,1*, aktuální symbol je *b*. Hashování bude provedeno následovně:

$parent\_index = hash(e, 1)$

$parent\_index = hash(cd, parent\_index)$

$parent\_index = hash(ab, parent\_index)$

V předchozím kroku byl použit následující symbol, proto: $result = parent\_index$

Kapitola 5

Vyhodnocení efektivity CD^{2}FA

Projekt Snort [4] obsahuje řadu regulárních výrazů, které jsou reálně používány v hloubkové analýze paketů. Na příkladech těchto regulárních výrazů proběhly experimenty.

Hlavním cílem CD^{2}FA je snížení paměťových nároků automatu. Tabulka 5.1 uvádí srovnání automatů, které vznikly na základě regulárních výrazů programu Snort. První dva sloupce uvádí počet stavů a přechodů v původním DFA. Třetí sloupec uvádí paměťové nároky původního DFA. Dále je zde uveden počet stromů implicitních přechodů ve výsledném CD^{2}FA, paměťové nároky výsledného CD^{2}FA a poměr paměťových nároků $CD^2FA : DFA$.

Tyto údaje jsou velmi podobné těm, které uvádí S. Kumar [3] a tedy lze dojít k závěru, že jeho výsledky byly potvrzeny.

	Stavy DFA	Přechody DFA	DFA paměť	stromů CD^{2}FA	CD^{2}FA paměť	poměr
1	1900	486400	5392200 B	164	343966 B	0.064
2	435	111360	1002240 B	38	78879 B	0.079
3	947	242432	2500080 B	194	410815 B	0.164
4	2219	568064	6816768 B	585	1201573 B	0.176
5	1080	276480	3065040 B	83	173658 B	0.057
6	60	15360	92160 B	18	37049 B	0.402

Obrázek 5.1: Srovnání DFA a CD^{2}FA

5.1 Použití CD^{2}FA

Výhody CD^{2}FA oproti tradičním DFA se zdají býti značné, ale aby byly opravdu dosaženy uvedené výsledky, je potřeba, aby byla zachována řada podmínek.

Předně, automaty, které jsou generovány z regulárních výrazů programu Snort (nebo obecněji – které jsou používány v DPI), jsou značně specifické. U obecných automatů (nebo dokonce náhodně vygenerovaných automatů) nelze CD^{2}FA vůbec použít. Při testování takových automatů se téměř všechny stavy staly kořeny jednoprvkových stromů, což bylo způsobeno omezením velikosti content labelu uvedeném v části 2.6.1, který může obsahovat maximálně pět symbolů. Pokud toto omezení bylo zvýšeno nebo odstraněno, vygenerované content labely měli délku mnoha desítek symbolů. U obecných konečných automatů tedy transformace na CD^{2}FA nemá smysl.

Dalším předpokladem je, že se podaří redukovat abecedu alespoň na polovinu a není příliš mnoho kořenových stavů. Potom je možné reprezentovat symbol sedmi bity a kořenový

stav deseti bity. Pak se content label vejde do čtyřech bytů, pokud obsahuje maximálně dva symboly, jinak do osmi bytů, pokud obsahuje maximálně pět symbolů, jak je požadováno v části 2.4. Dále nesmí být mnoho kořenových stavů, protože by se příliš zvětšil počet bitů potřebných pro jejich reprezentaci a ty by se nevešly do content labelu.

Dále se předpokládá, že bude nalezeno perfektní hashování jak je uvedeno v 2.4, a to pokud možno bez použití diskriminátorů (na diskriminátory už v content labelu nezbývají bity) a zvětšování hashovací tabulky. To ale není zaručeno, a to hlavně z toho důvodu, že některé content labely nemají příliš velké množiny kandidátních jmen. Vezměme jako příklad content label *ab,1*. Chceme-li se vyhnout použití diskriminátoru, lze generovat další kandidátní jména pouze funkcemi *permute* a *pad*. Funkce *pad* ale nemůže přidávat symboly pokud se content label má vejít do čtyřech bytů. Tedy získáme pouze dvě kandidátní jména funkcí permute - *ab,1* a *ba,1*. Pokud budou existovat tři stavy s tímto content labelem, nebude za daných podmínek perfektní hashování nalezeno.

Dalším předpokladem je, že mnoho údajů může být uloženo v cache paměti (tabulka překladu symbolů – viz 2.6, indexy počátků skupin stavů – viz 2.4 atd.) a že přístup k jiným údajům, než k obsahům stavů je z časového hlediska zanedbatelný.

Kapitola 6

Závěr

V programové části práce se podařilo implementovat konstrukci CD^{2}FA z DFA a simulaci jeho běhu. Byly provedeny experimenty, při kterých byl vytvořen CD^{2}FA na základě regulárních výrazů programu Snort a zhodnoceny statistiky využití paměti. Tyto statistiky potvrzují, že pro dané regulární výrazy je CD^{2}FA řádově asi desetkrát paměťově úspornější, než standardní DFA, což odpovídá výsledkům S. Kumara [3], ale za splnění řady předpokladů – nepříliš vysokého počtu kořenových uzlů, vysoké hodnoty hran grafu společných přechodů, nalezení perfektního hashování bez využití diskriminátoru a několika dalších. Tyto vlastnosti zpravidla automaty založené na regulárních výrazech používané pro účely DPI ovšem splňují.

Je zde prezentován vylepšený postup při přípravě perfektního hashování, které je používáno k adresování stavů. Dále je zde řešen problém započetí činnosti automatu.

Jako další možnost pokračování se nabízí implementace samotného automatu v hardwaru (resp. ve VHDL) a zhodnocení reálné rychlosti CD^{2}FA oproti DFA a zhodnocení reálného provozu obecně. Dále by bylo možné vylepšit proces konstrukce automatu tak, aby vhodně dynamicky nastavil podle potřeby parametry konstrukce – cílová velikost content labelu, počet bitů v diskriminátoru atp.

Literatura

[1] Kruskal, J. B.: On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. of the American Mathematical Society*, 1956: s. 7:48–50.

[2] Kumar, S.; Dharmapurikar, S.; Yu, F.; aj.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *SIGCOMM '06*, 2006: s. 339–350, iSBN 1-59593-308-5.

[3] Kumar, S.; Turner, J.; Williams, J.: Advanced algorithms for fast and scalable deep packet inspection. *ANCS '06*, 2006: s. 81–92, iSBN 1-59593-580-0.

[4] Roesch, M.: Snort - Lightweight Intrusion Detection for Networks. *LISA '99*, 1999: s. 229–238.