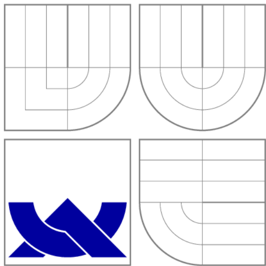
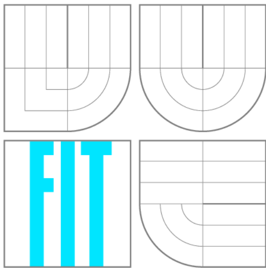


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

STRUKTURY TRIE PRO ZPRACOVÁNÍ ROZSÁHLÝCH TEXTOVÝCH DAT

TRIE STRUCTURES FOR LARGE TEXT DATA PROCESSING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. ANDREJ RAJČOK
AUTHOR

VEDOUCÍ PRÁCE — Doc. RNDr. PAVEL SMRŽ, Ph.D.
SUPERVISOR

BRNO 2016

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Rajčok Andrej, Bc.**

Obor: Inteligentní systémy

Téma: **Struktury trie pro zpracování rozsáhlých textových dat**

Trie Structures for Large Text Data Processing

Kategorie: Umělá inteligence

Pokyny:

1. Prostudujte metody efektivní implementace struktur trie a jejich použití v oblasti zpracování přirozeného jazyka.
2. Seznamte se s nástroji a postupy pro vyznačování pojmenovaných entit v textu a pro ukládání morfologických slovníků v jazycích s bohatou morfologií.
3. Na základě získaných poznatků navrhněte a realizujte systém, který dokáže efektivně ukládat rozsáhlé seznamy entit a morfologické slovníky, označovat výskyty pojmenovaných entit v textu a provádět morfologickou analýzu s rozsáhlými slovníky.
4. Vyhodnoťte realizované řešení a porovnejte zvolený přístup s jinými metodami.
5. Vytvořte stručný plakát prezentující práci, její cíle a výsledky

Literatura:

- dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- funkční prototyp řešení

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrž Pavel, doc. RNDr., Ph.D.**, UPGM FIT VUT

Datum zadání: 1. listopadu 2015

Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký

vedoucí ústavu

Abstrakt

Táto práca rozoberá spracovanie prirodzeného jazyka so zameraním sa na morfologickú analýzu a spoznávanie pomenovaných entít. Rozoberá efektívne vyhľadávanie v slovníkoch a v ňom používaných špecializovaných stručných štruktúr a ich praktické implementácie. Popisuje návrh a implementáciu systému pre spoznávanie pomenovaných entít a morfologického analyzátora za využitia stručných štruktúr a nakoniec porovnáva a testuje ich efektivitu a rýchlosť.

Abstract

This study analyzes natural language processing with emphasis on morphological analysis of inflective languages and systems for named entity recognition. It analyzes effective pattern matching in dictionary by using succint structures and then analyzes practical implementation of succint structures. It describes design and implementation of named entity recognition system and morphological analyzer and compares and test their speed and effectiveness.

Kľúčové slová

spracovanie prirodzeného jazyka, morfologický analyzátor, spoznávanie pomenovaných entít, stručné štruktúry, DAWG, cedar, darts-clone

Keywords

natural language processing, morphological analyzer, named entity recognition, succint structures, DAWG, cedar, darts-clone

Citácia

RAJČOK, Andrej. *Struktury trie pro zpracování rozsáhlých textových dat.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Smrž Pavel.

Struktury trie pro zpracování rozsáhlých textových dat

Prehlásenie

Prehlasujem, že jsom túto diplomovú práci vypracoval samostatne pod vedením pana doc. RNDr. Pavla Smrža Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Andrej Rajčok
24. mája 2016

Poďakovanie

Rád by som poďakoval svojmu vedúcemu doc. RNDr. Pavlovi Smržovi Ph.D. za poskytnutú podporu a rady pri písaní a vytváraní tejto diplomovej práce.

© Andrej Rajčok, 2016.

Táto práca vznikla ako školské dielo na FIT VUT v Brně. Práca je chránená autorským zákonom a jej využitie bez poskytnutia oprávnenia autorom je nezákonné, s výnimkou zákonne definovaných prípadov.

Obsah

1 Úvod **3**

2 Spracovanie prirodzeného jazyka **5**
2.1 Rozdelenie NLP podľa aplikácií a úrovní spracovania 5
2.2 Morfológia a morfologická analýza 7
2.2.1 Morfologická analýza morfologicky bohatých jazykov 7
2.2.2 Rozbor morfológických pravidiel a modelov pre jazyk 8
2.2.3 Aplikácie a praktické metódy morfologickej analýzy 8
2.2.4 Anotácia morfologických kategórií 10
2.3 Extrakcia informácií a spracovanie pomenovaných entít 10
2.3.1 Extrakcia informácií 10
2.3.2 Spoznávanie pomenovaných entít v texte 11
2.3.3 Vývoj spoznávania pomenovaných entít 11
2.3.4 Metódy pre spoznávanie pomenovaných entít 11
2.4 Aplikácie pre spoznávania pomenovaných entít a ich porovnanie 14
2.4.1 Systémy na testovanie a vyhodnocovanie systémov pre extrakciu informácií 16
2.4.2 Formát výstupneho hodnotenia 16

3 Efektívne vyhľadávanie v slovníkoch **17**
3.1 Štruktúry pre efektívne vyhľadávanie 17
3.1.1 Trie 17
3.1.2 Redukovaná trie 18
3.1.3 Deterministický acyklický graf slov 18
3.2 Slovníkový problém a stručné štruktúry 19
3.3 Hardvérovo efektívne algoritmy 20
3.3.1 Nedbajúcnosť na vyrovnávajúcu pamäť 20
3.4 Implementácie stručných datových štruktur 21
3.4.1 Dvojpoľové trie 22
3.4.2 Reprezentácie vybalancovanými zátvorkami 23
3.4.3 Level-ordered unary degree sequence, LOUDS 24
3.4.4 Deterministický acyklický graf slov 24
3.4.5 Porovnanie implementácií 25

4 Návrh a implementácia NER na základe FIGA **26**
4.1 FIT GAzeteer - FIGA 26
4.1.1 Knižnica Fsa p. Daciuka pre prácu s uloženými slovníkmi 26
4.1.2 Rozhranie FIGA 27

4.2 Návrh nového systému NER . . . 28
4.2.1 Knižnica Cedar a Darts-clone . . . 28
4.2.2 Formát dát v slovníku . . . 29
4.2.3 Využitie pôvodného systému FIGA . . . 29
4.2.4 Rozhranie nového systému . . . 30
4.2.5 Návrh systému . . . 31
4.3 Implementácia systému pre spoznávanie pomenovaných entít . . . 31
4.3.1 Dátové štruktury . . . 31
4.3.2 Spracovanie parametrov . . . 32
4.3.3 Algoritmus pre spracovanie textovej formy slovníka . . . 33
4.3.4 Algoritmus pre spracovanie textu . . . 33
4.3.5 Algoritmus pre spracovanie textu s automatickou opravou pravopisu 33
4.3.6 Algoritmus pre spracovanie výsledkov a ich výpis . . . 34
4.3.7 Algorimtus automatického doplňovania textu . . . 34
4.3.8 Algoritmus automatického opravovania pravopisu . . . 35
4.3.9 Podpora UTF-8 . . . 35

5 Návrh a implementácia morfologického analyzátoru **36**
5.1 Návrh morfologického analyzátora . . . 36
5.1.1 Pôvodný morfologický analyzátor . . . 36
5.1.2 Využitie pôvodného systému . . . 36
5.1.3 Formát dát v slovníkoch . . . 37
5.1.4 Použité knižnice . . . 38
5.2 Implementácia morfologického analyzátoru . . . 39
5.2.1 Trieda deriv_fsa . . . 39
5.2.2 Trieda morph_fsa . . . 40
5.2.3 Trieda all_words_fsa . . . 40
5.2.4 Trieda get_form_fsa . . . 40
5.2.5 Trieda numbers_fsa . . . 40

6 Testy a experimenty **43**
6.1 Testy potencionálnych knižníc . . . 43
6.1.1 Rozhrania knižníc . . . 43
6.1.2 Rýchlosť a efektivita knižníc . . . 43
6.2 Prenositelnosť knižníc . . . 44
6.3 Porovnanie efektivity a rýchlosti implementácií NER . . . 44
6.4 Rýchlosť systému NER . . . 45
6.4.1 Porovnanie s inými implementáciami . . . 46
6.5 Rýchlosť a efektivita morfologického analyzátoru . . . 47
6.6 Prenositeľnosť . . . 48

7 Záver **49**

Literatúra **51**

Prílohy **54**
Zoznam príloh . . . 55

A Obsah CD **56**

Kapitola 1

Úvod

Už od počiatku vývoja výpočtovej techniky bola snaha využiť výpočtovú techniku na spracovanie textu a jeho pochopenie. Spracovanie prirodzeného jazyka sa stalo jednou zo základných úloh pre výpočtovú techniku a v súčasnosti jeho dôležitosť iba stúpa.

Každý deň na twiteri pribudne približne štyridsať miliónov nových tvítov, prebehne približne štyri a pol miliardy vyhľadávaní cez Google a cez internet prejde cez dvesto miliónov gigabajtov dát. Prehliadače ako Google stíhajú indexovať a ukladať informácie o stránkach a ich obsahu, ale aj oni dokážu spracovať iba zlomok dát, ktoré pribúdajú každý deň na internet.

Pre získanie informácií z tohto mohutného toku dát, treba použiť čo najefektívnejšie a najrýchlejšie metódy ich spracovania.

Práca sa zameriava na prvý krok pre väčšinu aplikácií spracovania textu, a to lineárne prechádzanie textu, vyhľadanie a vyznačenie reťazcov zo slovníka a následná anotácia týchto častí textu informáciami zo slovníka. Konkrétne spracováva dva prípady tohto problému, a to spoznávanie pomenovaných entít v texte a morfologickú analýzu flektívnych jazykov.

Cieľom práce je vytvoriť nové systémy pre spoznávanie pomenovaných entít a morfologickú analýzu, ktoré budú rýchlejšie a efektívnejšie ako v súčasnosti používané systémy.

Práca sa skladá z piatich hlavných častí. V prvej časti sa rozoberá obor spracovania prirodzeného jazyka, morfologická analýza, modely jazykov a funkčné aplikácie morfologickej analýzy. Ďalej sa rozoberá extrakcia infromácií so zameranín na systémy pre spoznávanie pomenovaných entít, používané metódy a implementácie.

V druhej časti sa rozoberá problém efektívneho vyhľadávania vo veľkých slovníkoch. Definuje sa teoretický model reprezentujúci tento problém, štruktúry pre efektívne vyhľadávanie a definujú sa stručné dátové štruktúry. Následne sú rozobrané praktické implementácie rozličných stručných štruktúr, ako napríklad dvojpoľové trie, popis ich teoretického modelu a niekoľko praktických implementácií jednotlivých teoretických modelov.

Tretia časť obsahuje popis návrhu a implementácie systému pre spoznávanie pomenovaných entít, jej základné vlastnosti, formát vstupných a výstupných dát, formát použitých slovníkov a popis použitých algoritmov.

Štvrtá časť obsahuje popis návrhu a implementácie systému morfologickej analýzy, jej základné vlastnosti, formát vstupných a výstupných dát, formát použitých slovníkov a popis použitých algoritmov.

Posledná časť sa venuje testom rýchlosti systémov pre spoznávanie pomenovaných entít a testom rýchlosti systémov pre morfologickú analýzu. Obsahuje výsledky experimentov s rôznymi knižnicami a implementáciami na nich založených.

V rámci semestrálnej práce bol implementovaný prototyp systému pre spoznávanie pomenovaných entít, ktorý sa následne v diplomovej práci optimalizoval a doplnil o ďalšiu funkcionalitu, viď kapitola 4 a sekcia 6.1.

Kapitola 2

Spracovanie prirodzeného jazyka

Spracovanie prirodzeného jazyka (NLP - Natural Language Proccessing) je rozsah výpočtových techník zameraných na automatickú analýzu a reprezentáciu prirodzených ľudských jazykov na jednej alebo viacerých úrovniach s cieľom dosiahnuť spracovanie jazyka na úrovni ľudského spracovania pre široký záber úloh a aplikácií. [19]

2.1 Rozdelenie NLP podľa aplikácií a úrovní spracovania

V súčasnosti NLP pokrýva širokú škálu aplikácií, pre ktoré sa vytvorili teoretické základy, ako aj rôzne implementácie. Prakticky ľubovoľná úloha alebo aplikácia pracujúca so spracovaním textu môže použiť NLP techniky. Nasledujúce aplikácie sú v súčasnosti hlavným záujmom výzkumu a vývoja NLP: [16]

- Extrakcia informácií - zaoberá sa získaním štrukturovaných informácií z neštrukturovaného alebo semi-štrukturovaného textu. Cieľom extrakcie informácií je získať dostatočne kvalitné informácie tak, aby ich ostatné systémy a ľudia mohli ľahko používať. Zameriava sa na spoznávanie, označovanie a extrakciu kľúčových informácií do štrukturovanej reprezentácie pomocou veľkých kolekcií textov a slovníkov. Medzi kľúčové informácie patria napr. osobnosti, firmy, organizácie alebo lokality.

- Získavanie informácií - zaoberá sa získavaním informácií z textu, ktoré sú relevantné k žiadaným dátam, čiže poskytuje potencionálne relevantné dokumenty ako odpoveď na uživateľský požiadavok, napr. vyhľadávania Google.

- Automatický preklad - najstaršia časť NLP sa zaoberá prekladaním medzi rôznymi jazykmi, kde sa využívajú rozličné prístupy, od zamerania sa na preklad jednotlivých slov po metódy s vysokou úrovňou analýzy.

- Dialogické systémy - systémy schopné viesť dialóg s človekom. Tieto systémy sú rôznej úrovne, od inteligentných chladničiek až po semi-inteligentné systémy ako Siri alebo Cortana.

Moja práca sa zaoberá systémom pre spoznávanie pomenovaných entít a morfologickou analýzou. Systémy pre spoznávanie pomenovaných entít sú základným blokom pre extrakciu informácií a morfologický analyzátor je zas základným blokom pre spracovanie textu v morfologicky bohatých jazykoch, viď. sekciu 2.2.1. Oba pracujú s rôznymi vstupnými

informáciami a výstupnými informáciami, čo sa dá najlepšie popísať úrovňami spracovania, kde každa úroveň je definovaná spracovávanými dátami a má špecifické algoritmy pre spracovanie. NLP môžeme rozdeliť na tieto úrovne spracovania: [19]

- Fonológia sa zaoberá interpratáciou rečových zvukov vo vnútri slov aj medzi slovami, a je využívaná v rámci dialogických systémov.
- Morfológia sa zaoberá rozkladaním slov na najmenšie jednotky s významom - morfémy. Keďže význam jednotlivých morfén sa nemení a ostáva rovnaký pre rôzne slová, ľudia sú schopný pochopiť aj neznáme slová tým, že ich rozdelia na morfémy, ktorým rozumejú a význam slova získajú poskladaním významov jednotlivých morfém. Podobne aj NLP sa snaží pomocou významov jednotlivých morfém získať celkový význam slova a vety.
- Lexikálny level sa zaoberá priradzovaním významu k jednotlivým slovám. Môžu sa pritom využívať rozličné zložité slovníky, ktoré uschovávajú rôzne množstvo informácií o danom slove, napr. od typu slova ako napr. sloveso, až po sémantickú triedu slova a jeho kontext.
- Syntaktická úroveň sa zaoberá analýzou slov vo vete s cieľom pochopenia gramatickej štruktúry vety. Toto vyžaduje syntaktický analyzátor s gramatikou. Výstupom syntaktickej analýzy je reprezentácia vety, ktorá vyjadruje štrukturálne vzťahy medzi slovami.
- Semantické spracovanie sa snaží zistiť možný význam slova zameraním sa na interakciu medzi význammi jednotlivých slov. Toto spracovanie zahrňuje sémantické zjednoznačnenie slov s viacerými význammi, kde toto zjednoznačenie dovoľuje určiť a použiť správne práve jeden význam určeného mnohovýznamového slova v sémantickej reprezentácií vety. Existuje veľký počet metód, ktoré môžu byť implementované na dosiahnutie zjednoznačenia slova, niektoré využívajú frekvenciu výskytu jednotlivých významov, iné lokálny kontext slova, ďalšie celkový kontext slova v rámci celého dokumentu.
- Analýza na úrovni reči sa zaoberá analýzou jednotiek o dĺžke niekoľkých viet, kde vytvára prepojenia medzi jednotlivými komponetami viet. Pri tejto analýze sa využívajú hlavne dve metódy - anaforické vyriešenie, napríklad nahradzovanie zámien entitami, na ktoré odkazujú, a rozoznanie textovej štruktúry, ktorá rozhoduje o funkcii vety, čo pomáha ku dosiahnutiu zmysluplnej reprezentácie textu.
- Pragmatická úroveň sa zaoberá použitím jazyka v situáciach, a využíva celkový kontext textu. Cieľom je vysvetliť, ako sa v texte objavil ďalší význam bez toho, aby bol v texte zakódovaný. NLP pri pragmatickom spracovaní využíva odvodzovacie modely alebo obrovské obsahovo-vyčerpávajúce vedomostné báze.

V rámci mojej práce morfologický analyzátor vracia údaje pre systémy pracujúce na lexikálnej úrovni, napríklad prekladače, alebo syntaktickej úrovni, ako napríklad systémy pre označovanie častí reči. Systém pre spoznávanie pomenovaných entít zas pracuje na lexikálnej úrovni a vracia údaje napríklad pre sémantické spracovanie.

Keďže oba systémy pracujú na nízkych úrovňach, tak spracovávajú textový formát a vracajú označkovaný text. Spracovanie textu prebieha po slovách alebo častiach slov, ktoré sa porovnávajú oproti predpripraveným slovníkom, v ktorých sú uložené dodatočné informácie

pre označenie textu. Jednotlivé implementácie systémov majú tak tri základné odlišnosti, spôsob vytvárania predpripravenýc slovníkov, uloženie slovníkov a efektívne vyhľadávanie v nich a spôsob spracovania textu.

2.2 Morfológia a morfologická analýza

Morfológia je lingvistický obor, zaoberajúci sa identifikáciou, analýzou a popisom štruktúry morfém a iných lingvistických štruktúr ako afixy, koreň slova a intonácia.

Morfologická analýza je získavanie syntaktických informácií a vzťahov z rozboru slova a jeho tvaru. Praktická implementácia morfologickej analýzy potom pozostáva zo spracovania vstupného slova a vrátenie jeho lemy a jeho morfologických kategórií reprezentovaných sadou značiek. [5]

V rámci textu sú spomínané ešte následné lingvistické termíny:

- Lema je základný tvar slova.
- Lexéma je slovo v ľubovoľnom tvare.
- Morféma je najmenšia jednotka jazyka, napríklad slovný koreň, spojka alebo predpona.

2.2.1 Morfologická analýza morfologicky bohatých jazykov

Morfologická analýza sa v rámci systémov na spracovanie prirodzeného jazyka využíva už od počiatku spracovania prirodzeného jazyka. Automatický preklad začal využívaním jednoduchých syntaktických pravidiel, neskôr využíval lexikálnu základňu a aj systémy strojového učenia. V 70. rokoch ale stále bolo efektívnejšie a presnejšie ručné ľudské prekladanie. [16]

Pre anglický jazyk dosiahla analýza vynikajúce výsledky už začiatkom 90. rokoch, kedy sa podarilo vytvoriť parser pre anglický jazyk dosahujúci deväťdesiat percentnú úspešnosť pri rozbore textu, čím sa dosiahla vynikajúca úroveň pre anglický jazyk.

Anglický jazyk ale patrí medzi jazyky s menej zložitou morfológiou, kedy sa v rámci morfologických tvarov slova predáva väčšinou iba informácia o čísle, čase alebo osobe. Existujú aj jazyky bez morfológie, ako je napríklad klasická čínština, ktorá predáva všetky syntaktické a sémantické informácie pomocou poradia slov vo vete, ale čo je dôležitejšie, existujú aj jazyky zo zložitejšou morfológiou, ako sú napríklad slovanské jazyky, fínsky jazyk, hebrejský jazyk alebo arabský. Pre tieto jazyky dosahovali použité techniky minimálneho úspechu a pre tieto jazyky bolo treba ďalšieho výskumu efektívnejších spôsobov morfologickej analýzy.

Pre tieto jazyky predstavovala morfologická analýza podstatný problém, keďže techniky využívané pre angličtinu nedosahovali dostatočnú úspešnosť už pre angličtinu príbuzný jazyk - nemčinu, ktorá ale narozdiel od angličtiny patrí medzi morfologicky zložitejšie jazyky. V rámci týchto jazykov sa väčšina syntaktických a sémantických informácií vyjadruje pomocou rôznych tvarov slov. Tým sa morfologická analýza a rozbor stáva jedným zo základných blokov pre ľubovoľnú aplikáciu využívajúcu spracovanie takýchto tzv. morfologicky bohatých jazykov. [28]

2.2.2 Rozbor morfológických pravidiel a modelov pre jazyk

Morfologické pravidlá

V rámci morfológie existujú dva druhy pravidiel, skloňovacie a tvaroslovné pravidlá. Skloňovacie pravidá popisujú vytváranie rôznych tvarov lexém, s tým, že sa nemení základný význam slova. Tvaroslovné pravidlá vytvárajú nové slová, lexémy, buď derivivációu z pôvodných lexém alebo spojovaním lexém s lexémami alebo afixami, príklady pre jednotlivé pravidlá sú na obrázku 2.1. [5]

1. historický -> historické -> historickými
2. pre + historický -> prehistorický
3. história -> historik

Obr. 2.1: Príklady troch typov pravidiel. 1 je skloňovacie pravidlo 2 je tvaroslovné pravidlo za využitia konkatenácie a 3 je tvaroslovné pravidlo za využitia derivácie.

Morfologické modely jazykov

Existujú tri základné modely jazykov a to Morfémovo založená morfológia, Lexikálne založená morfológia a Morfológia založená na slovách.

- Morfémovo založená morfológia analyzuje slovné tvary ako usporiadania morfém. Morféma je v rámci modelu braná ako minimálna jednotka jazyka s významom. Slovo je analyzované ako samostatné morfémy konkatenované za sebou a vytvárajúce slovný tvar. Tento prístup má však problém s niektorými slovotvornými procesmi, ako napríklad analogický proces alebo nekonkatenačné procesy. [5]
- Morfológia založená na slovách využíva vzťah medzi vzorom a lemou, kde namiesto definovania pravidiel pre konkatenáciu morfém sa využívajú zovšeobecnené pravidlá medzi slovnou formou a skloňovacím vzorom. Slová sú kategorizované podľa vzorov, ku ktorým patria, a takto sa dajú kategorizovať aj nové slová a následne analyzovať na základe analogického aplikovania pravidiel. [5]
- Lexikálna morfológia berie jednotlivé slovné tvary ako výsledok aplikovania pravidiel na slovný tvar alebo koreň slova. Využíva vyššie zmienené slovotvorné a skloňovacie pravidlá. [5]

2.2.3 Aplikácie a praktické metódy morfologickej analýzy

Praktická morfologická analýza využíva väčšinou model morfológie založenej na slovách, alebo model lexikálnej morfológie. Praktická morfologická analýza ďalej využíva aj morfologické slovníky, kde sú uložené jednotlivé slovné tvary a ich morfologické kategórie. Jazyk má priemerne niekoľko miliónov rôznych tvarov slov, preto sa väčšina morfologických analyzátorov líši formátom dát uložených v morfologických slovníkoch, spôsobom ich generovania a spôsobom ich efektívneho prehľadávania.

Anglický morfologický analyzátor

Anglické morfologické analyzátori väčšinou využívajú slovník lem, slovník nepravidelných slov a set pravidiel pre generovania slovných tvarov z lem. Následne analyzátori z týchto

troch položiek vytvoria jednotný morfologický slovník, oproti ktorému sa porovnávajú vstupné dáta. Presnosť tohto slovníku závisí hlavne na slovníku nepravidielných slov, keďže set pravidiel a slovník lem dosiahli pre angličtinu maximálnej presnosti a nepresnosť analýzatoru spočíva hlavne v neznámych slovách alebo v slovách s nepravidelnými slovnými tvarmi. [22]

Český morfologický analyzátor

Pre morfológiu českého jazyka bol prvý morfologický analyzátor vyvinutý v Prahe na základe PDT - Prague Data Treebank. Tento morfologický analyzátor sa následne využíval vo viacerých českých označovačov častí reči, ako napríklad COMPOST v rámci projektu Morče. [14]

MorphoDiTa

Neskôr boli vyvinuté ďalšie morfologické analyzátori, ako napríklad MorphoDiTa, ktorá využíva modelu založeného na slovách. Využíva databázu lem a ich slovných tvarov, konkrétne Morflex CZ [15] vyvinutý v Českom morfologickom analyzátore, viď. sekciu 2.2.3. Záznamy z databáze zhlukuje do šablón na základe koncoviek. Jednotlivé lemy sú následne uložené v stromovej štruktúre, na ktorú sú napojené šablóny. V slovníku potom vyhľadáva lemu na základe prefixu slova a morfologické kategórie na základe sufixu slova. Ukážku systému je vidno na obrázku 2.2. [27]

```
Loading tagger: done
Tagging: done, in 0.1 deconds
znát        VB-S---1P-AA---
krišťálový  AAFS4----1A----
studánka    NNFS4-----A----
kde         Db-------------
hluboký     AAIS1----3A----
být         VB-S---3P-AA---
les         NNIS1-----A----
,           Z:-------------
tam         D-b------------
```

Obr. 2.2: Ukážka programu MorphoDiTa, kde na ľavej stranej sú jednotlivé spracované slová a na pravej strane sú získané údaje vyjadrené sadou značiek.

Majka

Ďalším českým morfologickým analyzátorom je Majka, a jej predchodza Ajka, na základe ktorej bol vytvorený aj morfologický analyzátor využívaný na FIT VUT. Svoj morfologický slovník vytvára zo slovníkov obsahujúcich lexémy nasledovaných lemou a morfologickými kategóriami. Lexémy a ich lemy následne zakóduje pomocou K-endings, a uloží ich do dvoch slovníkoch. V jednom je slovný tvar reprezentovaný ako slovný tvar , “:“, K-endings, “:“ a anotácia, v druhom je slovný tvar reprezentovaný ako lema, “:“, KEndings, “:“ a anotácia, viď. obrázok 2.3. [26]

KEndings kóduje morfologické operácie ako odstránenie n bajtov alebo symbolov od konca a nahradenie reťazcom, kde n je reprezentované ako symbol $A+n$, viď. obrázok 2.4

```
ježko:A:k1gMnSc1                $ echo test | majka -f input.w-lt
ježka:Cek:k1gMnSc2              test:k1gInSc1
ježka:Cek:k1gMnSc4              test:k1gInSc4
krtek:A:k1gMnSc1                test:k1gMnSc1
krtka:Cek:k1gMnSc2              testa:k1gFnPc2
```

Obr. 2.3: Na ľavo je ukážka slovníka a formátu uložených dát. Na pravo je ukážka výstupu systému Majka.

Vstup		**KEndings**	**Výstup**
slovenský+slovenskými	=>	Ami	slovenský+Ami
slovenskými+slovenský	=>	C	slovenskými+C
slovenský+slovenskému	=>	Bému	slovenský+Cému
slovenskému+slovenský	=>	Dý	slovenskému+Eý

Obr. 2.4: Ukážka zakódovania slovného tvaru a lemy pomocou KEndings. V treťom a štvrtom prípade je zakódované ako Cému, Eý namiesto Bému a Dý, keďže sa ráta podľa bajtov a *é* a *ý* vychádzajú na dva bajty.

2.2.4 Anotácia morfologických kategórií

Ako je vidno na ukážkach systémov Majka, obrázok 2.3, a MorphoDiTa, obrázok 2.2, označovanie jednotlivých morfologických kategórií je zákodované do anotácie pre ušetrenie priestoru a zvýšenie rýchlosti spracovania. V češtine sa využívajú dva systémy, jeden prevzatý z Ajky, druhý z PDT, oba reprezentujú jednotlivé kategórie postupnosťou symbolov, kde oba sú rovnako efektívne.

2.3 Extrakcia informácií a spracovanie pomenovaných entít

2.3.1 Extrakcia informácií

Zaoberá sa získaním štrukturovaných informácií z neštrukturovaného alebo semištrukturovaného textu. Z počiatku sa výskum extrakcie informácií zameriaval na vypĺňovanie preddefinovaných šablón alebo ručné vytváranie pravidiel na základe lingvistických vzorov a diagramov. Toto bolo časovo náročné a väčšina vytvorených šablón a pravidiel bola neprenosná medzi rôznymi platformami ako napríklad webové stránky alebo noviny. Na základe toho boli definované nové ciele pre extrakciu informácií, nezávislé na šablónach a platformách ako napríklad spoznávanie pomenovaných entít.

V súčasnosti extrakcia informácií rieši niekoľko základných úloh ako spoznávanie pomenovaných entít, extrakciu vzťahov medzi entitami a zjednoznačovanie sémantického významu. Pri spoznávaní sa hľadajú pomenované entity, ako sú napríklad osobnosti alebo lokality viď. obrázok 2.5, extrakcia vzťahov získava informácie o vzťahu medzi entitami napríklad *prezidentČoho*, viď. obrázok 2.5 a zjednoznačovanie rieši nejednožnačnosť jazyka a snaží sa určiť správne priradenie entít ku ich slovným reprezentáciam. [17]

Andrej Kiska sa stal prezidentom Slovenska po voľbách roku 2014.

Obr. 2.5: Na obrázku sú zelenou označené pomenované entity, ktoré môže systém na spoznávanie pomenovaných entít nájsť, a červeným kontext, ktorý pomáha pri extrakcií vzťahov, ako aj pri zjednoznačovaní. Napríklad, ak by existovalo viac známych Andrejov Kiskov, pomocou slov *prezident* a *2014* sa dokáže určiť, že sa hovorí o prezidentovi Slovenska. Pri extrakcii vzťahov sa získa vzťah *prezidentČoho(Andrej Kiska, Slovensko)*

2.3.2 Spoznávanie pomenovaných entít v texte

Pomenovaná entita

Pomenovaná entita je sekvencia slov, ktorá označuje nejakú entitu z reálneho sveta., napríklad *Slovensko*, *Vysoké učení technické v Brne* alebo *Dennis Ritchie*, viď. obrázok 2.5.

Spoznávanie pomenovaných entít

Hlavnou úlohou spoznávania pomenovaných entít — named entity recognition, alebo NER — je identifikovať pomenované entity z neštrukturovaného textu a zatriediť ich do jednej z preddefinovaných tried, ako napríklad *lokalita*, *organizácia* alebo *osoba*.

Spoznávanie pomenovávaných entít je jedna z najzákladnejších úloh v obore extrakcií informácií. Extrakcia komplexnejších štruktúr z textu, ako vzťahy medzi entitami a udalosťami, je závislá na presnom spoznávaní pomenovaných entít ako predzpracovávacom kroku. Spoznávanie pomenovaných entít je využívané aj v iných úlohách a aplikáciách. Napríklad, pri odpovedaní otázok, sú časti kandidátnej odpovede pomenované entity, ktoré musia byť najprv extrahované a klasifikované. Alebo pri entitne orientovanom prehľadávaní je spoznávanie pomenovaných entít v texte a vo vyhľadávacej požiadavke prvým krokom, ku dosiahnutiu čo najrelevantnejšieho výsledku. [17]

2.3.3 Vývoj spoznávania pomenovaných entít

Spoznávanie pomenovaných entít začalo už začiatkom 90. rokoch 20. storočia, aj keď úloha bola oficiálne formulovaná až v roku 1995 na šiestom MUC — MUC-6 — ako podúloha extrakcie informácií. Odvtedy bolo spoznávanie pomenovaných entít objektom viacerých štúdií a aj niekoľkých vyhodnocovacích programov napríklad CoNLL.

Najštudovanejším typmi pomenovaných entít sú osoba, organizácia a lokalita, ktoré boli prvé definované na MUC-6. Tieto typy sú dostatočne všeobecné, aby boli užitočné pre veľké množstvo aplikačných platforiem. Extrakcia dátumov, časov, penažných ohodnotení a percent, ktoré boli definované zároveň s typmi ako osoba na MUC-6, je tiež študovaná ako súčasť spoznávania pomenovaných entít, aj keď medzi ne technicky nepatrí. Okrem týchto všeobecných typov pomenovaných entít sú ďalšie typy pomenovaných entít zvyčajne definované pre špecifické platformy a aplikácie. [17]

2.3.4 Metódy pre spoznávanie pomenovaných entít

Pre identifikáciu jednotlivých pomenovaných entít sa využíva široká škála metód, napríklad porovnávanie reťazcov oproti slovníkom, metódy založené na pravidlách a metódy založené na štatistickom učení ako *hidden Markov models* alebo *Maximum-entropy Markov models*.

Metódy založené na pravidlách

Metódy založené na pravidlách používajú pravidlá, buď definované manuálne, alebo automaticky naučené, ktoré používajú na klasifikáciu textu reprezentovaného tokenmi s vlastnosťami. Pravidlo pozostáva zo vzoru a akcie. Ak tokeny z textu súhlasia so vzorom, vykoná sa špecifická akcia. Akcia može byť napríklad označenie entity alebo vloženie začiatku označenia entity.

Manuálne vytváranie pravidiel pre spoznávanie pomenovaných entít je ako obvykle nežiadúce, keďže je časovo náročné a je potrebné kvalifikovaná expertíza v obore. Pre automatické učenie pravidiel boli navrhnuté rôzne metódy, kategorizovateľné do dvoch tried, zhora-dole a zdola-hore. Pri oboch typoch metód je potrebný set trénovacích dokumentov s označenými pomenovanými entitami. Pri zhora-dole metódach sú najprv definované pravidlá pokrývajúce čo najviac testovacích inštancií. Potom systém definuje špecifickejšie pravidlá prekrývaním všeobecných pravidiel. Pri zdola-hore prístupe, špecifické pravidlá sú vytvárané na základe inštancií, ktoré ešte nie sú pokryté žiadnými pravidlami. [17]

Metódy založené na štatistickom strojovom učení

Metódy založené na štatistickom strojovom učení riešia problém spoznávania pomenovaných entít ako problém označenia sekvencie. Problém označenia sekvencie je všeobecný problém strojového učenia a bol použitý na modelovanie množstva úloh zo spracovania prirodzeného jazyka, ako napríklad označovanie časti reči.

Dostaneme sekvenciu pozorovaní, označenú ako $x = (x_1, x_2, ..., x_n)$. Zvyčajne je každé pozorovanie reprezentované ako vektor vlastností. Chceme priradiť označenie y_i každému pozorovaniu x_i. Obvykle by sa dalo predikovať označenie y_i iba na základe x_i, ale pri označovaní sekvencie sa predpokladá, že označenie y_i nezávisí iba na korešpondujúcom pozorovaní x_i, ale aj na ostatných pozorovaniach a ostatných označeniach. Táto závislosť je ale zväčša limitovaná na označenia a pozorovania v blízkosti pozície i.

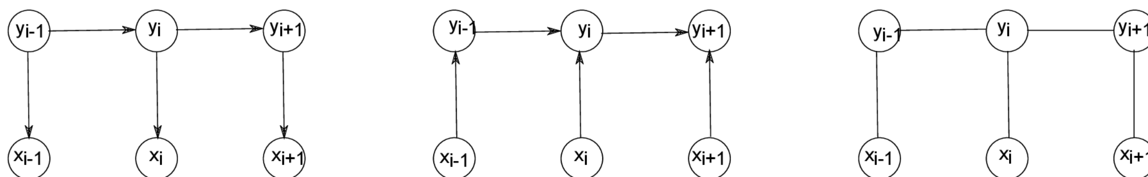
Pre namapovanie spoznávania pomenovaných entít na problém označenia sekvencie, bereme každé slovo ako pozorovanie. Označenie musia jasne indikovať hranice aj typ pomenovaných entít v sekvencii, pričom sa zväčša používa notácia BIO.

- Skryté markovove modely – Hidden Markov Models, HMD – v pravdepodobnostnom poňatí je najlepšia sekvencia označení $y = (y_1, y_2, ..., y_n)$ pre sekvenciu pozorovaní $y = (x_1, x_2, ..., x_n)$, teda tá, ktorá maximalizuje podmienenú pravdepodobnosť p(y|x), alebo ekvivalentne maximalizuje združenú pravdepodobnosť p(x,y). Združenú pravdepodobnost môžeme modelovať pomocou Markovho procesu, kde vytváranie označenia alebo pozorovania je závislé iba na niekoľkých predošlých označeniach a pozorovaniach. Potom ak budeme brať y ako skryté stavy, tak dostaneme Skrytý Markovov model. [31]

- Markovove modely maximálnej entropie – Maximum Entropy Markov Models, MEMM - na rozdiel od HMD, čo sú vytváracie modely, je MEMM diskriminatívny model. Výzkum ukazuje, že ak sú dostupné dostatočné trénovacie dáta tak diskriminatívne modely majú menšiu chybu predikcie. [17]

- Podmienené náhodné polia – Conditional Random Fields, CRF– Podmienené náhodné polia sú diskriminatívnym modelom pre problém označovania sekvencií. Boli predstavené ako možné riešenie pre problém extrakcie informácií. Rozdiel medzi CRF a MEMM je to, že CRF pri označovaní aktuálneho pozorovania neberie do úvahy iba

predošlé označenia, ale aj nasledujúce, viď. obrázok 2.6. CRF je k tomu neorientovaný grafický model, kým HMM a MEMM sú orientované, viď. obrázok 2.6. CRF sú od ich predstavenia používané v širokej škále aplikácií na zpracovanie prirodzeného jazyka. [21]



Obr. 2.6: Grafická reprezentácia lineárneho reťazca HMM, MEMM a CRF[17].

Systémy zjednoznačovania založené na vedomostnej bázi - NED, NERD

Využivájú slovníky s veľkým množstvom inštancií, v ktorých sú uložené pomenované entity a snažia sa identifikovať pomenovanú entitu v texte vyhľadávaním v slovníkoch.

Tento prístup už bol pôvodne skúšaný a zavrhnutý z viacerých dôvodov. Tieto slovníky museli byť manuálne vytvárané a anotované, čo bolo časovo náročné. Neskôr prestali byť rýchlo aktuálne a boli špecifické pre jednotlivé platformy. Navyše, aby slovníky boli obsahovo-vyčerpávajúce, museli zaberať veľké úložné miesto. Následkom čoho sa spomaľovalo vyhladávanie v slovníkoch.

Nový prístup využíva automatické vytváranie slovníkov z lingvistických zdrojov a encyklopédií ako WordNet a Wikipédia, kde sa za posledné roky urobil veľký pokrok smerom k spracovaniu, roztriedeniu a získaniu štrukturovaných informáciích, ako aj vzťahov medzi jednotlivými inštanciami. Na ukladanie slovníkov sa začali využívať špecializované štruktúry ako trie, viď. sekcia 3.1.1, alebo orientované grafy, pomocou ktorých sa dá aj v obrovských štruktúrach, ako sú tieto slovníky o desiatkach miliónov inštancií, efektívne a rýchlo vyhľadávať. [20]

V tomto prístupe sa na presné určenie pomenovanej entity využívajú systémy pre zjednoznačnovanie, ktorých úlohou je po tom ako sa z textu získa potencionálna entita, určiť na ktorú konkrétnu inštanciu v slovníku sa potencionálna entita odkazuje. Napríklad *Slovenka* môže byť žena slovenskej národnosti, časopis alebo firma. Využívajú pritom vyextrahované informácie v slovníkoch, ako vzťahy s inými entitami a vyhľadávajú v kontexte kľúčové slová relevanté pre danú entitu. [24]

Wiki extrakcia

Je špeciálnym prípadom spoznávania pomenovaných entít, keďže sa zaoberajú extrakciou pomenovaných entít a vzťahov medzi nimi z lingvistických a encyklopedických zdrojov, z ktorých najväčším a najdôležitejším zdrojom je Wikipédia.

Wikipédia je najväčším zdrojom vedomostí a najväčšia webová encyklopédia, ktorá obsahujé *wikistránky* dedikované špecifickým topikom a hyperlink odkazy na iné *wikistránky*.

Hlavnými problémami pre extrakciu užitočných informácií z wikistránok je ich rôzna štruktúra, úroveň kvality a možné konfliktné a nesprávne informácie. Niektoré wikistránky obsahujú už štrukturovanú informáciu, ako meno, dátum narodenia, národnosť pre osoby, iné naopak, sú len voľným textom, z ktorého je nutné túto informáciu získať. Konfliktné a

nesprávne informácie vznikajú už z princípu fungovania Wikipédie, kde každý môže editovať stránky a je len obmedzené množstvo editorov zaručujúcich kvalitu príspevkov. [4]

Aj napriek týmto problémom je Wikipédia najväčším a najkvalitnejším zdrojom vedomostí, ktorý je neustále aktutalizovaný pomocou crowdsourcingu, čím sa stáva ideálnym zdrojom pre vytváranie slovníkov pre spoznávanie pomenovaných entít. Boli navrhnuté rozdielne metódy a existuje niekoľko funkčných systémov pre extrakciu infromácií z Wikipédie, na ktorých boli vybudované systémy pre spoznávanie pomenovaných entít, ktoré dosiahli porovnateľné výsledky ako metódy založené na štatistickom strojovom učení. [24]

2.4 Aplikácie pre spoznávania pomenovaných entít a ich porovnanie

Aktuálne je vyvinutých a implementovaných niekoľko rôznych systémov na spoznávanie pomenovaných entít a systémov pre zjednoznačňovanie, pričom každý využíva rozličné prístupy, metódy a algoritmy, ako ukažku viď. obrázok 2.7, a každý dosahuje nejakú úroveň presnosti a pokrytia na rôznych datasetoch. Je preto ťažké vybrať najlepšie systémy pre spoznávanie pomenovaných entít, takže nasledujúce vymenovanie má za úlohu ukázať niektoré implementácie vyššie zmienených metód.

Obr. 2.7: Ukážka implementácie systému na spoznávanie entít. Toto je obrázok dema aplikácie systému TAGME, ktorá sa dá nájsť na http://tagme.di.unipi.it. Ukazuje nájdené entity v texte a informácie, o nich uložené vo vedomostnej bázi, v tomto prípade vo Wikipédií.

- Wikipédia Miner — vyvinutý v roku 2008, je založený na využívaní rôznych faktov, ako relatívnosť kontextu a kvalita, ktoré sú potom skombinované klasifikátorom. Bol vyskúšaný na AQAINT datasete. [29]
- AGDISTIS — je systém len pre zjednoznačňovanie. Využíva podobnosť reťazcov, rozpínavú heuristiku na označenia a grafovo-založený HITS algoritmus. [29]

- AIDA2 — je založená na budovaní grafu súdržnosti a algorimoch pre husté subgrafy, pričom využíva vedomostnú bázu YAGO2. Systém bol testovaný na ručné anotovanom subsete z 2003 CoNLL. [29]

- NERD-ML — vyvinutý na extrakciu pomenovaných entít z tweetov. Je založený na strojovom učení typu entít, s tým, že potrebuje: vektor vlastností vytvorený zo setu lingvistických vlastností, výstup natrénovaného CRF klasifikátoru a výstup extraktorov na spoznávanie pomenovaných entít podporovaných NERD rozhraním. Testované na sete mikropostov a novinových stránok. [29]

- WAT — je nasledovníkom TagMe. Pri vývoji WAT bola redizajnovaná väčšina komponentov pôvodného TagMe, ako pozorovateľ, zjednoznačovač a pruner. Boli uvedené dve nové zjednoznačovacie kategórie: grafovo-založené algoritmy pre koletívne spájanie entít a volebne-založené algoritmy pre zjednoznačnovanie lokálnych entít. Pozorovateľ a pruner môžu byť ladené pomocou SVM lineárnych modelov. Môže byť použitý aj ako čisto zjednoznačovač, ak dostane nutné dáta. [23]

- DEXTER — je open-source implementácia systému a rozhrania pre zjednoznačovanie entít. Systém bol implementovaný na zjednoznačenie implementácie spájania entít, a dovoľuje nahrádzať jednotlivé časti procesu. V rámci implementácie bolo použitých niekoľko rôznych zjednoznačovacích metód, ako napríklad pôvodná TagMe zjednoznačovacia metóda. [29]

- DBpedia Spotlight — je jeden z prvých sémantických prístupov. Jeho framework kombinuje NER a NED – named entity disambiguation - prístupy založených na DBpedii. Využíva reprezentáciu entít vo vektorovom priestore a používa cosinúsovú podobnosť. [29]

- Secapi — je systém na spracovanie pomenovaných entít a ich zjednoznačenie vyvíjaný na FIT VUT. Využíva vedomostnú bázu, ktorá vznikla extrakciou a následným zlúčením relevantných informácií o pomenovaných entitách z viacerých zdrojov ako Wikipédia, Freebase, Geonames alebo Getty ULAN. Na spoznávanie pomenovaných entít v texte sa využíva nástroj FIGA - Fit Gazeteer – ktorý je založený na konečných automatoch.

- KEA NER/NED — využíva kontextový model, ktorý bere do úvahy heterogénne zdroje, ako aj automatickú multimediálnu analýzu. Zdrojové texty môžu mať rozdielne úrovne presnosti, kompletnosti, jemnosti a spoľahlivosti, čo vplýva na zistenie súčasného kontextu. Nejednoznačnosť je vyriešená vybraním entitných kandidátov s najväčšou pravdepodobnosťou vzhľadom na vopred určený kontext. Využíva DBpediu, konkrétne využíva n-gram analýzu, a potom vyhľadáva všetky potencionálne entity v DBpedii. [29]

- Babelfy — využíva princíp náhodných prechádzaní a algoritmus hustého podgrafu na riešenie problému nejednoznačnosti a spájania entít v multilinguálnom prostredí. Bol otestovaný na 6 datasetoch: dve AIDA, tri SemEval úlohy a Senseval. [29]

- Stanford Named Entity Recognizer — je java implementácia NER, ktorá poskytuje možnosť extrakcie vlastností a veľa možností pre definovanie extraktorov vlastností. Standford NER je CRF klasifikátor, je implementovaný ako model sekvencie lineárne zreťazených CRF, čiže pomocou trénovacích dát je možné vybudovať modely pre NER alebo iné úlohy. [11]

- Targeted Hypernym Discovery – využíva metódy založené na pravidlách, kde pravidlá definuje pomocou JAPE gramatiky. Pomocou nej potom vo vstupnom texte vyhľadáva kandidátne entity, pre ktoré pomocou THD algoritmu vyhľadá zodpovedajúce wiki stránky, z ktorých sú extrahované hypernymy, ktoré sú potom zlinkované DBpediou a vrátené v NIF formáte. [10]
- DBpedia — slúži ako vedomostná báza pre mnoho systémov pre spoznávanie pomenovaných entít alebo zjednoznačovanie. Extrahuje štrukturované informácie z wikipédie a je konštantne automaticky aktualizovaná. DBpedia ponúka rozhranie pre požiadavky typu SQL pre vyhľadávanie entít. DBpedia má vlastný formát dát a obrovské množstvo typov entít, ktoré sú definované v DBpedia Ontology. Jej cieľom je ponúknuť základ pre vývoj sémantického webu. [4]

2.4.1 Systémy na testovanie a vyhodnocovanie systémov pre extrakciu informácií

Väčšina uvedených systémov bola testovaná na rôznych datasetoch, či už špecifických pre nejakú sútaž alebo konferenciu, alebo špeciálne pre nich vytvorené datasety ako napríklad v prípade NERD-ML. Najčastejšie využívané datasety boli datasety vytvorené pre systém AIDA, založené na datasetoch z konferencie CoNLL, ale aj tieto neboli využité ani v polovici prípadov.

Získanie porovnateľných výsledkov ostáva závažným problémom, kde viac ako polovica času výzkumu je strávená pripravovaním dát na experimenty. Hlavným dôvodom tohto problému v systémoch na extrakciu informácií je neexistencia štandardu a rozdielne datové reprezentácie v jednotlivých datasetoch. Z týchto problémov vyplynulo, že pri výzkume sa používali hlavne datasety spĺňajúce dve podmienky. Po prvé museli byť dostupné a po druhé implementácie parseru a vyhodnocovacého systému museli byť čo najjednoduchšie a časovo najefektívnejšie. Na základe toho sa začali vyvíjať systémy s rozhraniami, ktorých úlohou bolo pomocou rozhrania ľahko otestovať a vyhodnotiť jednotlivé systémy, príkladom porovnávajúceho systému je napríklad GERBIL alebo GATE. [29]

2.4.2 Formát výstupneho hodnotenia

Výsledkami hodnotení sú tzv. F-measures, alebo F-merania, ktoré sú štatistickou mierou presnosti pri binárnej klasifikácií. Používa sa pri tom harmonický priemer, v ktorom sa využíva pomer presnosti, tj. koľko získaných entít bolo naozaj entitami, a citlivosti, čiže koľko z celkového počtu entít bolo identifikovaných. Existujú rôzne varianty F- meraní, kde každá dáva iný dôraz na citlivosť a presnosť, poprípade využijú inú štatistickú metódu ako harmonický priemer.

Kapitola 3

Efektívne vyhľadávanie v slovníkoch

Vo väčšine súčasných implementácií systémov na spoznávanie pomenovaných entít sa využívajú nejaké druhy vedomostných bázi, ktoré obsahujú niekoľko desiatok až stoviek miliónov inštancií, v ktorých treba vyhľadávať kandidátne pomenované entity na základe vstupného textu. To isté platí pre morfologické analyzátori, ktoré využívajú rozsiahle morfologické slovníky vygenerovaných slov a ich tvarov, oproti ktorým porovnávajú vstupný text. Aby toto bolo možné v reálnom čase, bolo treba začať ukladať tieto slovníky v špecializovaných štruktúrach, ktoré podporujú rýchle a efektívne vyhľadávacie algoritmy. Ďalším problémom, ktorý sa musí riešiť pri využívaní rozsiahlych slovníkoch je aj ich veľkosť, ktorá môže ovplyvňovať rýchlosť vyhľadávania. [20]

3.1 Štruktúry pre efektívne vyhľadávanie

Tento problém je typický pre väčšinu sekcií spracovania prirodzeného jazyka alebo spracovania textu. Praktické štruktúry boli vymyslené pri riešení problému hľadania vzoru alebo reťazca v texte. Navrhnuté riešenie pracovalo vytvorením stromu sufixov, kde sa všetky možné sufixy postupne počas sekvenčného spracovania textu uložili do stromovej štruktúry. Princíp efektívneho uloženia je, že ak dva sufixy majú spoločný prefix, tak maju aj spoločnú cestu od koreňového uzla po nejaký uzol, kde cesta reprezentuje daný prefix. Daný reťazec alebo vzor sa vyhľadával ako cesta vo vytvorenom strome. Táto štruktúra bola známa ako sufix trees alebo sufix trie.

Výskum sa ďalej zaoberal hlavne na urýchlenie vytvorenia stromovej štruktúry, ale tieto štruktúry môžu byť použité pri tzv. „offline“ hľadaní, kedy sa môže predspracovať text a vytvoriť stromová štruktúra, pomocou ktorej sa potom môžu vyhľadávať reťazce a vzory. Toto dovolovaľo, aby časová zložitosť hľadania nebola závisla na dĺžke textu v ktorom sa vyhľadáva, ale iba na dĺžke hľadaného reťazca. Toto je ideálne pre slovníky, ktoré stačí iba raz spracovať a potom sa v nich dá rýchlo vyhľadávať bez ohľadu na ich veľkosť. [2]

3.1.1 Trie

Nech je daný set kľúčov K, kde pre všetky klúče k patriace do setu K platí, že sú reťazcami symbolov z abecedy Σ. Ďalej existuje znak \$ nepatriaci do abecedy Σ. Trie je potom reprezentáciou setu K, pre ktorú platí: trie je stromová štruktúra, kde každa cesta z koreňa do listu reprezentuje jeden klúč v reprezentovanom sete.

Oblúky v trie sú označené symbolmi c a pre každý uzol platí, že z neho ide maximálne jeden oblúk označený symbolom c, kde c je z množiny $\Sigma \cup \{ \$ \}$. Oblúky v trie korešpondujú s jednotlivými symbolmi kľúčov v sete. Uzol, ku ktorému vedie symbol \$, je označený ako koncový list a nevedú z neho žiadne oblúky. Ku každému kľúču môže byť priradená hodnota, ktorá je uložená v koncovom liste. [1]

Využíva sa v rámci vyhľadavacích algoritmov, kde umožňuje vyhľadávanie v čase *O(k)*, kde k je dĺžka hľadaného kľúča. Ďalej sa využíva ako dokonalá hešovacia funkcia alebo v sieťových aplikáciach, napríklad na ukladanie pravidiel pre filtrovanie paketov.

3.1.2 Redukovaná trie

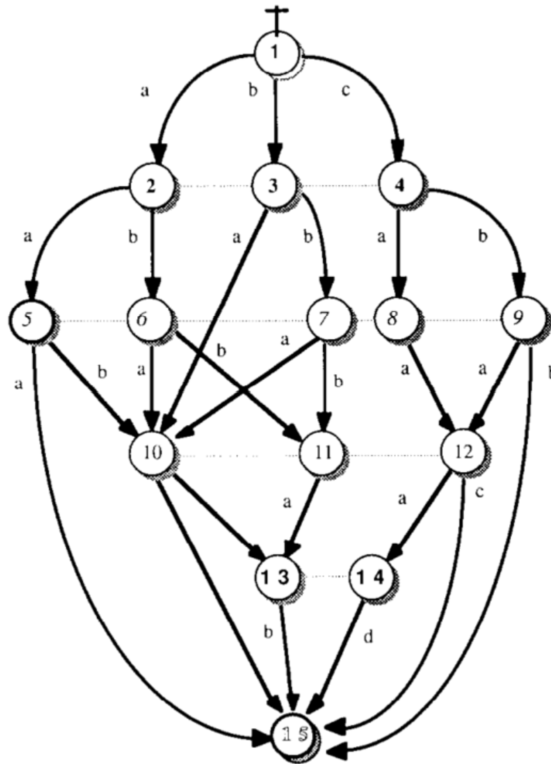
K je set klúčov reprezentovaný trie. Trie je zložená z uzlov a oblúkov. Označenie oblúkov pozostáva zo symbolov. Oblúk označený a z uzlu n do m je reprezentovaný notáciou $g = (n, a) = m$. Pre klúč k v sete K, je uzol m, kde $g = (n, a) = m$, *separovaný uzol*, ak a je dostatočné na rozlíšenie klúča k od všetkých ostaných v sete K. Strom vytvorený iba z oblúkov medzi koreňom a separátnymi uzlami pre všetky klúče v sete K, sa nazýva *redukovaná trie*, viď. obrázok 3.1. [1]

Obr. 3.1: Ukážka transformácie trie na redukovanú trie

3.1.3 Deterministický acyklický graf slov

Deterministický acyklický graf slov – Deterministic Acyclic Word Graph, DAWG - je koncový automat, ktorý príjma všetky slová patriace do slovníka. Príjme abecedu ako svoju vstupnú abecedu. Možeme ho chápať ako trie, kde jednotlivé stavy sú uzly, ale jeho hlavná výhoda oproti trie je, že spoločné sufixy sú reprezentované jedinou sekvenciou stavov, viď. obrázok 3.2.

Nech $w = \{a_1, ..., a_n | a_1, ..., a_n \in \sum\}$ je slovo v $\sum^*$. Nech $w = \{xyz\}$, čiže x je prefix, y je podslovo a z je sufix slova w. Pre každé neprázdne $y \in \sum^*$, $end\!-\!set$ pre y vo w je definované ako $end - set_w(y) = \{i : y = a_{i-|y|+1} ..., a_i\}$. Zvlášť pre $end - set_w(\lambda) = \{0, 1, 2, ..., n\}$. Povieme, že x a y v $\sum^*$ sú end-equivalent (na w), ak $end - set_w(x) = end - set_w(y)$. [7, 6]



Obr. 3.2: Ukážka DAWG nad abecedou pozostávajúcou z *a,b,c,d*. Stav 15 určuje koniec slova, a jednotlivé cesty z uzlu 1 do uzlu 15 predstavujú rozdielne zakódované kľúče. [25]

3.2 Slovníkový problém a stručné štruktúry

Slovníkový problém je teoretickým vyjadrením vyššie zmieneného problému, a snaží sa vyriešiť, ako čo najefektívnejšie uložiť dáta, čiže čo najbližšie spodnej teoretickej hranici. Dá sa ním popísať veľké množstvo situácií, kde je podstatný rýchly prístup k veľkému množstvu dát, napríklad suffix polia alebo IP vyhľadávacie trie.

Nech $S = \{s_1, ..., s_2\}$ je zoradený set n položiek z univerza $U = \{0, 1, ...u - 1\}$ veľkosti u, ciže $i < j$ implikuje $s_i < s_j$. Chceme reprezentovať S v *stručnej* - succint - forme tak, aby sme mohli robiť základné slovníkove vyhľadávania nad jej kompresovanou formou. [13]

Formálnejšie, definujeme základné funkcie nad setom dát. Nech $a \in U$.

- $rank(S, a)$ funkcia vráti počet položiek v S, ktoré sú menšie alebo rovné a.

  $rank(S, a) = |\{s_i | s_i \leq a\}|$

- $select(S, i)$ funkcia vracia i-tú najmenšiu položku zo setu S, pre i rovné 1 až n.

  $select(S, i) = s_i$

- $member(S, a)$ funkcia indikuje, či sa a vyskytuje v sete S.

  $member(S, a) = 1$ ak $a \in S, 0$ inak.

- $prank(S, a)$ je rank funkcia, ale iba pre položky z S.

  $prank(S, a) = rank(S, a)$ ak $a \in S, -1$ inak.

- $pred(S, a)$ funkcia vracia predchodcu a, najväčšiu položku x v S, takú že $x < a$.

  $pred(S, a) = max s_i | s_i < a$ ak $rank(S, a-1) > 0, -1$ inak.

Jacobson, autor štručných datových štruktúr, argumentoval, že *rank* a *select* funkcie sú efektívnejšie ako zvyšné, a ukázal, že tieto funkcie dokážu zložitejšie vyhľadávania než *member* a *pred*. Výsledkom toho je, že väčšina neskoršieho výzkumu brala *rank* a *select* ako fundamentálne operácie nad slovníkovými štruktúrami. [13].

- Indexovateľný slovnik(IS) reprezentuje subset $S \subseteq U$ a podporuje funkcie $prank(S, a)$ a $select(S, i)$.

- Plne indexovateľný slovník(PIS) reprezentuje subset $S \subseteq U$ a podporuje funkcie $rank(S, a)$ a $select(S, i)$.

Plne indexovateľný slovník dokáže riešiť vyhľadávanie predchodcov, čím je skvelo aplikovateľný na problémy ako IP vyhľadávacie štruktúry alebo indexovanie kompresovanáho textu.

Nech pre každý set S o n položkách, každá položka s_i má tiež asociované satelitné dáta d_i. Na zaistenie rýchleho získania satelitných dát pre zadanú položku, môžeme vytvoriť set dvojíc S' formátu (klúč, dáta), kde $S = \{(s_1, d_1), (s_2, d_2), ..., (s_n, d_n)\}$, a vybudovať slovník nad setom S'.

- $lookup(S', a)$ ak $a = s_j$ tak $d_j, null$ inak .

- Vyhľadávací slovník(LS) reprezentuje set S' a podporuje funkciu $lookup(S, a)$.

Nech $A = d_1 d_2..d_n$ je bitový vektor dĺžky $|A| = \sum_i |d_i|$ bitov, kde satelitné dáta d_i sú konkatenované dohromady. Ak sú satelitné dáta d_i fixnej dĺžky r, tak ich dokážeme uložit do poľa o dĺžke $n \times r$ bitov. Vieme zkonštruovať IS nad S, tak že pre každú položku s_i, *prank* vráti pozíciu v A, kde sú uložené satelitné dáta danej položky. [13]

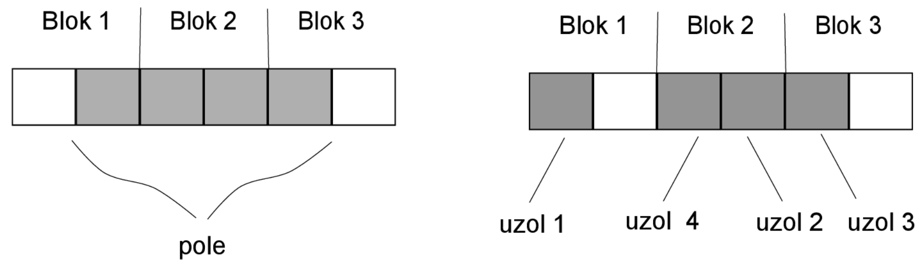
Funkcia *select* býva väčšinou implementovaná pomocou funkcie *rank*, presnejšie ako binárne vyhľadávanie nad *rank*. Týmto sa *rank* stáva najdôležitejšou časťou problému, keďže od jeho implementácie závisí rýchlosť vyhľadávania, ako aj veľkosť slovníka. Rôzne praktické implementácie sa líšia hlavne štruktúrami, ktoré využívajú na rýchlu implementáciu funkcie *rank*, čo väčšinou zahŕňa rôzličné vyhľadávacie tabuľky a pomocné polia.

3.3 Hardvérovo efektívne algoritmy

Výpočet teoretickej zložitosti algoritmov nebere do úvahy mechanické obmedzenia alebo vlastnosti hardvéru. Rôzne algoritmy s rovnakou zložitosťou, a dokonca aj rôzne implementácie rovnakého algoritmu môžu byť rádovo rýchlejšie ak sú tzv. "mechanicky sympatickejšie". Mechanicky sympatické znamená, že algoritmus využíva vlastnosti hardvéru a vyhýba sa jeho obmedzeniam. Pre slovníky, a štručné datové štruktúry používané v algoritmoch pre vyhľadávanie v nich, je dôležitá tzv "cache oblivous" vlastnosť, čiže nedbajúcnosť na vyrovnávaciu pamät, kde algoritmy sú nezávislé na parametroch využívanej pamäte.

3.3.1 Nedbajúcnosť na vyrovnávajúcu pamäť

Väčšina výpočtovej techniky je von Neumanovského typu, kde dôležité pre túto vlastnosť je existencia procesoru, ktorý je napojený na pamäť. Využíva sa hlavne hierarchická pamäť,



Obr. 3.3: Na ľavo je ukážka poľa uloženého v pamäti a najhorší možný prípad pri jeho sekvenčnom prechádzaní. Na pravo je ukážka najhoršieho možného prípadu pre prechádzanie trie, kde každý uzol má samostatne alokovanú pamäť a tieto alokované pamäte su náhodne porozhadzované po pamäti počítača. V oboch prípadoch má vyrovnávacia pamäť veľkosť jedného bloku. Pre sekvenčný prechod poľa sú to tri prístupy a pre prechádzanie trie sú to štyri prístupy, aj keď pristupujú k rovnakému počtu prvkov.

kde existuje niekoľko rôznych pamätí, kde prístupová doba do pamäte je priamo úmerná veľkosti pamäte. V hlavnej pamäti sú všetky dáta a má najdlhšiu dobu prístupu. Keď počítač pracuje s dátami tak si ich vytiahne z hlavnej pamäte a uloží do menších vyrovnávacích pamätí s menšou dobou prístupu. Dáta sa vyťahujú po blokoch, ktoré sú následne celé uložené do menšej vyrovnávajúcej pamäte. [8]

Tu dochádza k obmieňaniu uložených blokov dát kvoli obmedzeniu veľkosti pamäte. Môže nastať prípad, kde už raz načítaný blok dát sa musí znovu načítať, keďže bol vymenený za iný blok dát, a musí sa znova pristupovať do hlavnej pamäte.

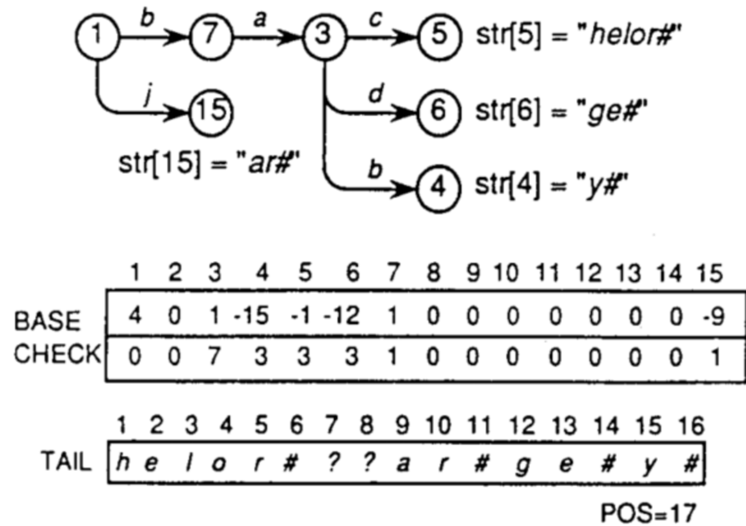
Nedbajúcnosť na vyrovnávaciu pamäť je vlastnosť, kedy algoritmy minimalizujú počet prístupov do hlavnej pamäte a hlavne minimalizujú znovu načítanie dát bez ohľadu na parametre pamätí. Medzi parametre pamäte patri veľkosť pamäte a veľkosť bloku. Ak pre algoritmus platí nedbajúcnosť na vyrovnávajúcu pamäť nad dvojúrovňovou hierachiou pamätí, tak táto nedbajúcnsoť na vyrovnávajúcu pamäť platí pre ľubovolne hlbokú hierarchiu pamätí.

Algoritmus nedbajúci na vyrovnávajúcu pamäť je napríklad algoritmus, ktorý prechádza sekvenčne blok pamäte, napríklad pole, počet prístupov do pamät bude maximálne veľkosť pola delená veľkosťou bloku plus jedna. viď. obrázok 3.3. Algoritmus bez tejto vlastnsoti je napríklad prechádzanie trie, kde každý uzol má samostatne alokovanú pamäť, a uzly môžu byť každy v inom bloku pamäte. A aj keď môžu byť uzly v tom istom bloku, kľudne sa môže načítať raz pre jeden uzol, a kým sa dostane ku druhému uloženému uzlu v bloku, je tento blok nahradený iným a musí sa znova načítať z hlavnej pamäte, viď. obrázok 3.3. [9]

3.4 Implementácie stručných datových štruktur

Jednotlivé implementácie sa líšia teoretickou implementáciou *rank* funkcie a kódovou reprezentáciou štruktúr. Ale aj implementácie využívajúce rovnakú teoretickú funkciu dosahujú rozličných výsledkov, keďže správna implementácia stručných datových štruktúr je náročná a nemusí vždy správne efektívne fungovať. Ďalej novšie implementácie väčšinou pridávajú menšie alebo väčšie zmeny v snahe dosiahnuť či už lepšiu rýchlosť, veľkosť využívanej pamäte alebo pridať ďalšiu funkcionalitu.

Väčšina praktických implementácií systémov pre spoznávania pomenovaných entít má vedomostnú bázu s pomenovanými entitami a ich kontextom uloženú samostatne, a v štruk-



Obr. 3.4: Ukážka reprezentácie redukovanej trie pomocou dvojpoľovej trie. Napríklad, j má hodnotu 11, čiže zoberieme hodnotu uzlu, v ktorom práve sme, čo je počiatočný uzol 1, zoberieme jeho hodnotu v $BASE$, čo je 4, sčítame s hodnotou j, čo je 11, dostaneme 15. Pozreme sa do $BASE[15]$, kde je -9, kde záporné číslo indikuje koniec slova a odkazuje nás do poľa $TAIL$, kde získame ar, a zložením získame slovo *jar*. [1]

túrach sú uložené iba samotné pomenované entity a index do uložených slovníkoch, čo uľahčuje implementáciu takýchto štruktúr pre uloženie vedomostných bází, kde každá inštancia pomenovanej entity má priradenú hodnotu, ktorá je reprezentovaná nejakým štandardným typom.

3.4.1 Dvojpoľové trie

Dvojpoľová trie - Double-Array trie - je technika na kompresiu trie do dvoch jednodimenzionálnych polí $BASE$ a $CHECK$, nazývaných dvojpole. V dvojpoli sú neprázdne pozície uzla n namapované pomocou poľa $BASE$ do poľa $CHECK$ tak, že dve neprázdne lokácie v ľubovoľných uzloch nie sú namapované na tú istú pozíciu v poli $CHECK$. Každý oblúk v trie môže byť získaný z dvojpoľa v čase $O(1)$, čiže v najhoršom prípade, časová zložitosť získania klúča je $O(k)$, kde k je dĺžka daného klúča. V dvojpoľovej trie sú uložené len nutné uzly, a tak koniec klúčov je uložený v samostatnom poli $TAIL$. [1] Vzťah medzi redukovanou trie a dvojpoľovou trie je nasledujúci:

- Ak existuje oblúk $g(n,a) = m$ v redukovanej trie, tak $BASE[n] + a = m$ a $CHECK[m] = n$.
- Ak uzol m je separátny uzol taký, že jeho chvostový reťazec $str[m] = b_1, b_2, .., b_k$ tak $BASE[m] < 0$ a platí: $p = -BASE[m]$, $TAIL[p] = b_1$, $TAIL[p+1] = b_2$, ..., $TAIL[p+k-1] = b_k$.

Praktické implementácie Dvojpoľových trie

- Darts je C++ knižnica implementujúca Dvojpoľovú trie, ktorú je možno použiť ako slovník alebo hash. Jej rozhranie je použité ako základ rozhraní ďalších knižníc pre

stručné štruktúry. [18]

- Cedar je C++ knižnica implementujúca efektívnu dynamickú dvojpoľovú trie, založenú na efektívnom vkladaní a odstranovaní dát, čo dovoľuje rýchlu aktualizáciu trie.

  Na rozdiel od originálnej dvojpolovej trie, získanie potomkov uzla sa vypočíta pomocou XOR namiesto sčítania ako c = *BASE[n] XOR* a . Pole $CHECK$ uchováva adresu rodiča a používa sa na overenie validity.

  Pridávanie nového uzla vytvára konflikt, kedy sa môže daný uzol namapovať na už namapované miesto. V takom prípade treba realokovať všetky súrodenecké uzly, buď nového alebo starého uzla a realokovať ich vetvu na nové nepoužívané miesto. Práve táto operácia je časovo najzložitejšia.

  Na vykonanie pridávania uzlu rýchlo, boli pridané dve nové jednorozmerné polia, nazývané $NLINK$ - node link - a $BLOCK$. Pre každý uzol, $NLINK$ uloží označenie potrebné na dosiahnutie prvého potomka a označenie nutné na dosiahnutie susedného súrodeneckého uzla z rodičovského uzla. Toto dovoľuje rýchle prechádzanie súrodencov pri realokovaní.

  $BLOCK$ udržiava informácie o voľných adresách vo vnútri 256 bytového odielu nazývaneho blok v poliach $BASE$ a $CHECK$. Každý blok je klasifikovaný do troch kategórií *plný*, *zatvorený* a *otvorený*. Plné bloky nemajú voľné adresy a nerealokujú sa do nich žiadne nové uzly. Zatvorené bloky majú, buď iba jednu volnú adresu, alebo počet neúspešných realokácií do bloku prekročil stanovenú hranicu. Otvorené bloky sú všetky ostatné bloky, čiže bloky s väčším množstvom voľných adries.[30]

  Schopnosť rýchlej aktutalizácie slovníka ale mierne spomaľuje vyhľadávanie a hlavne niekoľko násobne zväčšuje pamäťové požiadavky pre uloženie slovníka.

3.4.2 Reprezentácie vybalancovanými zátvorkami

Reprezentácia stromu balancovými zátvorkami - balanced parentheses representation, BP - bola navrhnutá autorom stručných štruktúr Jacobsonom, pričom neskôr bola zlepšená a dosahuje konštantne časy.

Stavia sa pomocou prechádzania stromu, zapisujúc otvárajúce zátvorky pri príchode do uzlu po prvý raz, a zatváracie zátvorky pri prechode na predchodcu, po prejdení podstromu s koreňom v danom uzle. Dostaneme sekvenciu $2n$ vybalancovaných zátvoriek. Každý uzol je reprezentovaný zodpovedajúcou dvojcou zátvoriek '(' a ')', pričom uzol je identifikovaný jeho otvárajúcou zátvorkou. Podstrom uzlu x obsahuje uzly, ktorých reprezentácia je medzi jeho párom zátvoriek. [3]

Konkrétne implementácie BP

Nech $P[1, 2n]$ je sekvencia n párov balancovaných zátvoriek, reprezentovaných bitmi 0 a 1. Nech $excess(i) = rank((i) - rank)(i)$. Potom $findclose(i)$ je najmenše $j > i$, také že *excess(j) = excess(i - 1)*. Taktiež $enclose(i)$ je najväčšie $j < i$, také že $excess(j) = excess(i) - 1$. [3]

- Hešovo založená heuristika - rozdelí zátvorky do 3 kategorií, *close*, *near* a far. Nech b = *log n* a s = *b log n*. Potom *close* zátvorky sú tie, ktorých vzdialenosť od seba je

maximálne b. $Near$ zátvorky sú tie, ktoré sú od seba vzdialené $b+1$ až s, a far sú zvyšné, ktorých vzdialensoť je väčšia ako s.

V hešovacích tabuľkách sú uložené informácie o *near* a far zátvorkách, ktoré sa využívajú pri implementácií $findclose$, zároveň s rozdelním tabuľky na kusy bitov, po ktorých sa vyhľadávajú zátvorky[3].

- Range Min-Max-Tree založené reprezentácie - RMM– využívajú $rangemin - max$ stromy. Tieto sú vybudované nad virtuálnym poľom $excess(i)$ hodnôt. Sekvencia P je rozdelená na bloky $b = n/2$ zátvorok, pričom minimálny a maximálny *excess* z bloku je uložený. Následne je k blokov uložených do superbloku. Následne je k superblokov uložených do ďalších superblokov atd., až kým nie je postavená kompletná k-nárna hierarchia. Nad touto štruktúrou sa následne dajú ľahko implementovať základné operácie. [3]

Praktické implementácie BP

- SDSL-lite - knižnica implementujúca širokú škálu stručných štruktúr, medzi ktoré patri aj reprezentácia balancovanými zátvorkami. Implementuje Range min-max strom implementáciu a Hierarchickú štruktúru s pionerskymi zátvorkami. [12]

3.4.3 Level-ordered unary degree sequence, LOUDS

Je podobná reprezentácií balancovanými zátvorkami, prechádza strom po úrovňach, pri vstupe do uzla pridá i otváracích zátvoriek a jednu zatváraciu zátvorku, kde i je počet detí uzla. Výsledná frekvencia je balancovaná, ak sa na začiatok pridá jedná otváracia zátvorka. Potom pomocu operácií *rank* a *select* implementuje všetky základné operácie. [3]

Praktické implementácie LOUDS

- Tx-trie — priamo implementuje LOUDS. Najnovšia implementácia výrazne znížila veľkosť využívanej pamäte, a tým dokáže ukladať množstvo klúčov, okolo miliardy.
- Marisa-trie — využíva Patricia trie, typ Sufix trie, založených na LOUDS reprezentácií.

3.4.4 Deterministický acyklický graf slov

Praktické implementácie DAWG

- Darts-clone je klon knižnice Darts, ktorá využíva dvojpoľové trie. Oproti Darts má niekoľko výhod, po prvé na uloženie každej jednotky využíva iba 4 bajty namiesto 8 bajtov využívaných v Darts. Po druhé využíva vyššie zmienený deterministický acyklický graf, čím znižuje počet jednotiek nutných na reprezentáciu trie.
- Knižnica fsa p. Daciuka je knižnica využívajúca štruktury DAWG a vyvinutá pre ukladanie slovníkov a efektívne hľadanie v nich. Jej implementácia je pomalšia oproti novším implementáciam, ale zachováva si efektívnu veľkosť slovníka. Pomalosť knižnice je daná aj neefektívnym rozhraním, kde je možné prechádzať triu iba po jednom symbole, a pre nájdenie špecifického oblúka sa musia po jednom prehľadať potomkovia rodičovského uzla.

3.4.5 Porovnanie implementácií

Autori *cedaru* spravili aj porovnania s ostatnými implementáciami štruktúr pre ukladanie slovníkov, viď. obrázok 3.5, medzi ktorými sú aj DAWG alebo LOUDS. Toto poslúžilo ako referenčné štatistiky pri porovnávaní jednotlivých knižních počas výberu knižnice pre implementáciu systému na spoznávanie pomenovaných entít.

Software	Data Structure	Space [MiB]	Size [MiB]	Build [ns/key]	Lookup [ns/key]
cedar	Double-array trie	832.82	816.54	183.57	38.95
cedar `ORDERED=false`	Double-array trie	832.81	816.54	171.38	39.07
cedar	Double-array reduced trie	782.46	642.38	176.86	43.48
cedar `ORDERED=false`	Double-array reduced trie	773.89	642.38	**166.46**	43.51
cedar	Double-array prefix trie	**490.61**	488.38	211.33	38.88
cedar `ORDERED=false`	Double-array prefix trie	**490.59**	488.35	221.87	39.07
libdatrie 0.2.8	Double-array prefix trie	1229.12	644.97	209955.04	124.66
libtrie 0.1.1	Double-array two-trie	2312.11	654.39	5401.59	181.95
dary	Double-array trie	897.75	895.54	51144.92	57.90
doar 0.0.13	Compacted double-array trie	1937.25	*334.59	990.51	48.00
Darts 0.32	Double-array trie	4306.02	858.93	2387.87	40.89
Darts-clone 0.32g	Directed-acyclic word graph	2311.39	409.17	1339.14	**36.39**
Darts-clone 0.32e5	Compacted double-array trie	2779.10	**309.31**	1011.92	59.42
DASTrie 1.0	Compacted double-array trie	2626.16	383.37	92634.88	85.02
tx-trie 0.18	LOUDS trie	1791.10	*113.11	626.90	972.32
ux-trie 0.1.9	LOUDS two-trie	2223.80	*92.39	1229.11	1975.28
marisa-trie 0.2.4	LOUDS nested patricia trie	2036.49	***87.27**	698.76	194.87
			log		log

Obr. 3.5: Referenčé štatistiky z http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar, ukazujú efektivity jednotlivých implementácií pre rôzne vlastnosti, napríklad malá veľkosť *LOUDS*, ale zároveň ich veľké vyhľadávacie časy. Ako najrýchlejšia implementácia sa ukazuje $Darts - clone$, ale len nepatrne oproti *cedar* a *doar*.

Kapitola 4

Návrh a implementácia NER na základe FIGA

Súčasťou práce je návrh a implementovanie systému pre spoznávanie pomenovaných entít, s využitím stručných dátových štruktúr. Ako predloha sa mohol využiť predtým vyvíjaný systém na spoznávanie pomenovaných entít FIGA s tým, že sa zároveň zachová pôvodné rozhranie pôvodného systému.

4.1 FIT GAzeteer - FIGA

Systém FIGA využíva implementáciu stručnej dátovej štruktúry typu DAWG, konkrétnejšie knižnicu *fsa* p. Daciuka. FIGA je schopná spoznávať entity v texte v reálnom čase, má funkciu automatického doplňovania, ktorá vracia, buď dané množstvo potencionálnych entít z nájdeného možného fragmentu, alebo všetky možné potencionálne pomenované entity. Ďalšou funkciou je automatické opravovanie pravopisu, ktorý dokáže spoznávať entity v texte aj v prípade, že sa v nich vyskytujú preklepy, ako napríklad chýbajúce písmeno alebo písmeno navyše.

4.1.1 Knižnica Fsa p. Daciuka pre prácu s uloženými slovníkmi

Knižnica *fsa* p. Daciuka využívaná vo FIGA je implementovaná ako C knižnica a kolekcia skriptov patriacich k nej. Knižnica ponúka rozhranie pre prácu so slovníkmi uložených v spracovanej podobe v súboroch.

Skripty vytvárajú konečné automaty zo slovníkov, ktoré sú následne spracované a ukladáné do súborov. Je niekoľko rôznych skriptov pre rôzne druhy slovníkov. Jednotlivé druhy slovníkov sú špecializované pre jednotlivé úlohy ako automatická oprava pravopisu alebo automatické doplňovanie textu. Jednotlivé skripty môžu využívať ešte dodatočné súbory s vydolovanými informáciami o slovníkoch a použitom jazyku, ako je frekvencia slov alebo znakov v slovníku, poprípade abeceda jazyka a vymeniteľné slabiky.

Pre prechádzanie slovníkov ponúka iba jednu funkciu *next_node* spoločne s *forallnodes*, pomocou ktorej sa dajú prechádzať potomkovia zadaného uzla, kde v každom uzle je uložený symbol oblúku vedúcemu k nemu. Ďalej sa dá prejsť na potomka a nastaviť ho ako nový rodičovský uzol. Teoreticky treba prejsť všetkých potomkov uzlu pre nájdenie správneho podstromu alebo vylúčiť existenciu daného podstromu.

4.1.2 Rozhranie FIGA

Systém FIGA spracováva vstup, buď zo štandardného vstupu alebo zo súboru, a má nasledujúce parametre:

- parameter *-h* vypíše stručný popis programu, jeho verziu a popis jeho parametrov,
- povinný parameter *-d SÚBOR* berie súbor pomenovaný *SÚBOR*, v ktorom je uloženým slovník pomenovaných entít,
- parameter *-f SÚBOR* berie súbor pomenovaný *SÚBOR* ako vstupný súbor obsahujúci text pre spracovanie, ak nie je zadaný berie sa vstup zo štandardného vstupu,
- parameter *-p* umožnuje vypisovanie výsledkov na štandardný výstup, ktorý má nasledujúci formát viď. 4.1:

  začiatok riadku, všetky získané indexy oddelené ; , poradové číslo počiatočného symbolu reťazca, poradové číslo koncového symbolu reťazca, a následne reťazec z textu identifikovaný ako pomenovaná entita.

  Dané indexy odkazujú do vedomostnej báze na kandidátne pomenované entity,
- parameter *-b* určuje typ poradového čisla, ak je zadaný počíta poradové číslo počiatku reťazca v bajtoch, ak nie je, tak počíta v symboloch,
- parameter *-o* dovoľuje prekrývanie pomenovaných entít, viď. obrázok 4.2,
- parameter *-q* dovoľuje apostrofom a úvodzovkám slúžiť ako hranice pre pomenovanú entitu,
- parameter *-a* spúšta funkciu automatického doplňovania potencionálnych pomenovaných entít,
- parameter *-m ČÍSLO* definuje maximálny počet vrátených potecionálnych pomenovaných entít ku jednému fragmentu na *ČÍSLO*, a funguje iba v kombinácií s parametrom *-a*,
- parameter *-x* určuje, že sa pri automatickom doplňovaní majú vrátiť všetky potencionálne pomenované entity, a funguje iba v kombinácií s parametrom *-a*,
- parameter *-s* spúšta funkciu automatického opravovania pravopisu.

index začiatok koniec reťazec
./ner -d vedomostna_baza -f vstupny_text -p
131335 1 7 Charles
79894;129672 9 19 H. Franklin

Obr. 4.1: Ukážka výstupu systému NER. Na začiatku idú indexy do vedomostnej báze, nasledujú počiatočné a koncové offsety a na konci je reťazec zo vstupného textu, ktorý je spoznaní ako entita.

Slovnik pomenovaných entít
Jan 11111,22222
Jan Zelený 11111
Jan Modrý 22222
Peter Jan 33333

Text:
Peter Jan Zelený

Identifikované entity
Peter Jan Zelený

Obr. 4.2: Ukážka prekrývania entít. Systém spozná zeleno a červeno ohraničenú pomenovanú entitu. Dôležité je si všimnúť, že nespozná entitu Ján, ktorá je prefixom dlhšej entity.

4.2 Návrh nového systému NER

Pred návrhom nového systému bolo treba preskúmať dostupné knižnice pre stručné štruktúry, zistiť ich možnosti na preskúmavanie uložených slovníkov a určiť najlepší formát uloženia klúčov a asociovaných hodnôt do slovníka a ďalej zvážiť využitie pôvodného kódu. Následne na základe získaných informácií navrhnúť základnú funkcionalitu systému.

4.2.1 Knižnica Cedar a Darts-clone

Knižnica *cedar* má rozhranie pre ukladanie a načítavanie slovníkov zo súborov. Funkcia na vytváranie slovníkov má niekoľko parametrov, prvý a povinný je ukazateľ na reťazce charov končiace znakom s hodnotou 0. Tieto reťazce sú jednotlivé kľúče slovníka. Ďalším dôležitým parametrom je ukazovateľ na pole hodnôt, kde index klúča korešponduje s indexom jeho asociovanej hodnoty. Tento parameter je nepovinný, a v prípade, že nie je zadaný, sa automaticky vygenerujú asociované hodnoty identické s indexom kľúčov.

Cedar ponúka niekoľko funkcií pre preskúmavanie slovníkov, pričom každa môže byť spustená z ľubovoľného uzla, ktorý potom slúži ako koreňový uzol. Prvá je *exactMatchSearch*, ktorá umožnuje exaktné porovnanie reťazca oproti kľúčom v slovníku, a vráti, či sa daný reťazec nachádza ako klúč v slovníku.

Funkcia *commonPrefixPredict* vracia počet unikátnych sufixov pre prefix zadaný ako vstupný reťazec, a tiež vracia pole štruktúr, v ktorom sú uložené údaje potrebné k získaniu jednotlivých sufixov pomocou pomocnej funkcie $suffix$. Maximálny počet vrátených štruktúr je daný parametrom, a pole musí byť vopred alokované s dostatkom priestoru na uloženie zadaného počtu štruktúr.

Poslednou funkciou je *traverse*, ktorá ponúka prechádzanie trie po častiach reťazca. Kde ako vstup berie reťazec a počiatočný uzol, a potom overí, či z daného uzla ide sekvencia oblúkov daná reťazcom. Vracia tri údaje, prvým je návratová hodnota, druhým je dosiahnutý uzol a tretím je počet spracovaných znakov z reťazca. Návratová hodnota je buď asociovaná s klúčom, alebo hodnota *NO_VALUE*, či hodnota *NO_KEY*.

- Hodnota *NO_VALUE* znamená, že existuje cesta z daného uzla pomocou daného reťazca, ktorý je súčasťou prefixu nejakého kľúča, ale dosiahnutý uzol nie je koncový pre žiadny klúč.

- Ak vráti hodnotu asociovanú s nejakým kľučom, to znamená, že existuje cesta z daného uzla pomocou daného reťazca, ktorý je súčasťou prefixu nejakého kľúča, a zároveň je dosiahnutý koncový uzol pre nejaký kľúč.

- Hodnota *NO_KEY* znamená, že z daného uzla neexistuje pre daný reťazec cesta, a

počet znakov z reťazca, pre ktoré existuje cesta, je vrátený ako počet spracovaných znakov.

Ďalej ponúka pomocné funkcie *begin*, *next* a *suffix*, ktoré zoberú zadaný uzol ako koreňový, a prechádzajú celý podstrom po jednotlivých koncových uzloch a vracajú korešpondujúce klúče. Konkrétne:

- funkcia *begin* zobere daný uzol, hĺbku daného uzla v strome a vráti jeho najpravejší koncový uzol, jeho hĺbku a jeho asociovanú hodnotu,
- funkcia *next* zobere koncový uzol, jeho hĺbku, koreňový uzol a vráti najpravejšieho ľavého suseda pre daný uzol, jeho hĺbku a asociovanú hodnotu, alebo *NO_KEY*, ak neexistuje ďalši ľavý sused,
- funkcia *suffix* potom zobere koncový uzol a dĺžku, a vráti sufix klúča pre daný uzol o zadanej dĺžke.

Rozhranie pre *darts-clone* je iba podmnožinou rozhrania knižnice *cedar*, kde *darts-clone* nepodporuje funkciu *commonPrefixPredict* a ani pomocné funkcie ako *begin*.

4.2.2 Formát dát v slovníku

Dáta uložené v slovníku nemohli byť jednoducho uložené ako klúč a asociovaná hodnota, keďže pre kľúč môže existovať viacero hodnôt, napríklad v prípade rovnakých názvov, alebo v prípade názvu popisujúceho viacero objektov, ako mesto, okres a kraj. Pre tento problém boli možné dve riešenia:

- vytvoriť samostatnú datovú štruktúru, najlepšie pole, kde sa budú ukladať indexy do vedomostnej bázy, a do slovníka sa uloží index ukazujúci do poľa. Má to výhodu jednoduchého vytvárania slovníku, ale aj niekoľko závažných nevýhod. Musí sa vytvárať, udržiavať, ukladať a predávať ďalšia datová štruktúra, čo by znamenalo zmenu existujúceho rozhrania, a naimplementovať knižnicu alebo triedu pre prácu s ňou. Znamenalo by to tiež pridanie ďalšej vrstvy odkazov medzi klúčom a kontextovými dátami.
- uložiť každú hodnotu ako samostatný klúč, pričom tieto bonusové kľúče sa budú líšiť sufixom, pozostávajúcim z nevytlačiteľných znakov, ktoré nepatria do používanej abecedy slovníka. Má niekoľko nevýhod, ako vytváranie nových kľúčov, zložitejšie získavanie všetkých hodnôt pre daný kľúč, a praktické znefunkčnenie niektorých funkcií z rozhrania knižnice *cedar*, ako *commonPrefixPredict*, *begin*, *next* a *suffix*.

Nakoniec som sa rozhodol pre implementovanie druhej možnosti, keďže je jednoduchšia na implementáciu a údržbu, aj keď znefunkčnením niektorých funkcií knižnice *cedar* sa niektoré funkcie NER musia implementovať komplikovanejšie. Ďalej je kompatibilná s pôvodným rozhraním na rozdiel od prvej možnosti. Pre podrobnejší popis viď. obrázok 4.3.

4.2.3 Využitie pôvodného systému FIGA

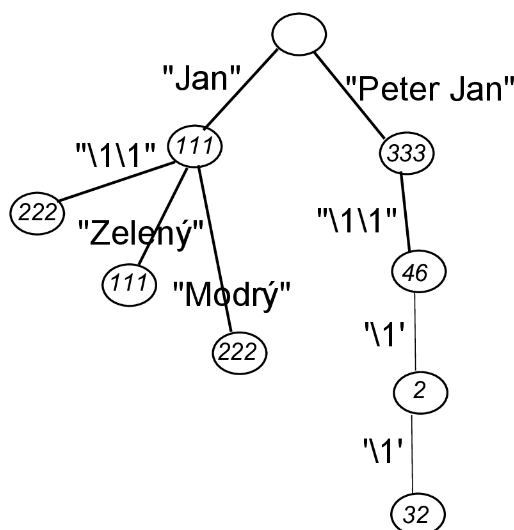
Aby sa znova nemuselo vymyslieť koleso, zvážil som aj reimplementáciu pomocou úpravy pôvodného kódu, a potom som zvážil aj na akej úrovni abstrakcie kódu. Nakoniec som sa rozhodol, že zachovám iba pôvodné rozhranie a to z viacerých dôvodov.

Slovník pomenovaných entít

Jan	111;222
Jan Zelený	111;
Jan Modrý	222;
Peter Jan	333;46;2;32

Formát pre vybudovanie slovníka

Pole reťazcov	Pole hodnôt
Jan	111
Jan'\1\1'	222
Jan Zelený	111
Jan Modrý	222
Peter Jan	333
Peter Jan'\1\1'	46
Peter Jan'\1\1\1'	2
Peter Jan'\1\1\1\1'	32



Obr. 4.3: Ukážka formátu slovníka. Vľavo hore je slovník v textovej podobe, v ktorej je uložený v čase spracovania. Vľavo dole je formát spracovaných dát, ktoré sa predajú funkcii na vytvorenie slovníka. Vpravo je výsledná reprezentácia pomocou trie. Hrubé čiary so slovami v dvojtých úvodzovkách reprezentujú postupnosť uzlov a oblúkov. Plná čiara so symbolom v úvodzovkách reprezentuje oblúk. Hodnoty v uzloch sú asociované hodnoty ku kľúčom. Daná reprezentácia je potom zakódovaná do binárnej podoby do polí a následne do súboru, a súčasne využívaná pre vyhľadávanie.

Úprava iba knižničných volaní na funkcie pracujúce zo slovníkmi nebola možná, kvôli absolútne rozdielnym spôsobom prechádzania slovníkov. Tieto dva spôsoby sa nedali vymeniť jeden za druhý bez, buď úplného prepísania kódu, alebo napísania zložitého rozhrania medzi nimi, ktoré by nakoniec neprinieslo žiadne zlepšenie rýchlosti novej implementácie.

Úprava pôvodného kódu by bola moc zložitá a zabralo by čas naštudovať pôvodný kód a vymyslieť minimálne zásahy pre maximalizovanie rýchlosti. A podrobné štúdium pôvodného kódu pre reimplementáciu není nutné, keďže požadované funkcie na nový systém sú dobre špecifikované. Ďalej pôvodný systém využíva viacero rozdielnych slovníkov a špeciálnych reprezentácií, ktoré v novej implementácií nie sú nutné.

4.2.4 Rozhranie nového systému

Aj keď nový systém má všetky časti pôvodného rozhrania, boli k nemu pridané ďalšie časti, špecifickejšie pre vytváranie binárnych reprezentácií slovníkov z textovej formy. V pôvodnom systéme toto bolo riešené pomocou skriptov, ktoré boli súčasťou distribúcie knižnice *fsa* p. Daciuka. Rozhranie bolo rozšírené o dva parametre:

- *-n* , zadávaný spoločne s parametrom *-d SUBOR*, a určuje, že *SUBOR* neobsahuje binárnu reprezentáciu, ale textovú reprezentáciu slovníka vo formáte:

  kľúč medzera index1;index2; … ;indexi koniec riadku,

- *-w SUBOR* určuje, že použitý slovník sa zapíše v binárnej forme do súboru *SUBOR*.

4.2.5 Návrh systému

Nový systém má implementovať už zmienené automatické doplňovanie, a automatickú opravu pravopisu a funkcie jednotlivých parametrov. Ďalšou požiadavkou je podpora pre UTF-8 v rámci všetkých funkcií, čo je hlavne dôležité pre funkcie automatickej opravy pravopisu.

Ďalšou požiadavkou na implementáciu bolo implementovať systém tak, aby bolo možné použiť ľubovoľnú knižnicu využívajucu rozhranie knižnice *DARTS*, ktorá je základom rozhrania knižnice *cedar*, čo znamená použitie iba 3 základných funkcií knižnice *cedar* a nie pomocných funkcií, ktoré nepatria do pôvodného rozhrania. Toto je dosiahnuté pomocou využitia šablón, kedy stačí predať ako parameter triedu reprezentujúcu knižnicu podporujúcu funkciu *traverse*. Stačí využívať funkciu *traverse* vďaka zvolenému formátu uloženia asociovaných hodnôt do slovníka.

Systém využíva frontu, kde ukladá spracovávané slová, aby bol schopný backtrackingu, keďže sa musí snažiť nájsť najdlhší možný reťazec z textu odpovedajúci nejakému kľúču zo slovníka. Pre efektívne spracovanie textu si ukladá medzivýsledky do kontextu slova, aby ich mohol znovu použiť pri opätovnom spracovávaní slova počas backtrackingu.

4.3 Implementácia systému pre spoznávanie pomenovaných entít

Systém pre spoznávanie pomenovaných entít je implementovaný ako dve časti, prvou je rozhranie a to je implementované v súbore *main*, kde sa spracovávajú vstupné parametre, a druhou je knižnica *figa_cedar*, obsahujúca triedu *figa_cedar*, ktorá implementuje funkcionalitu systému.

4.3.1 Dátové štruktury

Implementácia okrem štandardných datových štruktúr, ako vektor a list, využíva ďalšie dve špeciálne datové štruktúry, kde prvá reprezentuje frontu spracovávaných slov a súčasný kontext a druhá spracovávané slovo s jeho kontextom.

```
typedef struct context{
    int value;
    size_t from;
    unsigned int start;
    unsigned int end;
    bool capital;
    string word;
    vector<size_t> spell_from;
}t_context;
```

Obr. 4.4: Definícia štruktúry, ktorá reprezentuje slovo s jeho kontextom, kde ako kontext sa berie napríklad poradie vo vstupnom texte a dosiahnutý uzol v trie získaný po aplikovaní slova.

- definicia štruktúry reprezentujúcej slovo je na obrázku 4.4. Položky *start* a *end* určujú poradové číslo a *capital* uchováva informáciu, či je prvé písmeno kapitálne, aby sa to nemuselo opakovane zisťovať.

  Položka *word* uchováva spracovávané slovo a položka *value* uchováva návratovú hodnotu funkcie *traverse* pre dané slovo.

  Položka *from* je id uzlu, kam vedie slovo, a položka *spell_from* uchováva id uzlov, v prípade, že bola prevedená automatická oprava pravopisu, a existuje niekoľko alternatívnych ciest cez trie. Tieto posledné dve položky sa využívajú pri získavaní všetkých asociovaných hodnôt, po identifikovaní pomenovanej entity.

```
typedef struct res{
    list<t_context> value;
    size_t from;
    bool load;
    bool found;
    vector<size_t> spell_from;
    bool spell;
}t_status;
```

Obr. 4.5: Definícia štruktúry reprezentujúca frontu spracovávaných slov, ktorá definuje frontu štruktúr reprezentujúcu slovo a jeho kontext, aktuálnu pozíciu v trie a aktuálny stav spracovania textu.

- definícia štruktúry reprezentujúcu frontu spracovávaných slov je na obrázku 4.5.

  Položka *from* udržiava súčasnú pozíciu v trie, konkrétne id uzlu, položka *value*, trieda list zo štandardnej C++ knižnice, udržiava práve spracovávané slová a ich kontext. Položka *load* určuje, či sa majú načítavať nové slová do fronty, alebo znovu spracovať slová vo fronte.

  Položka *found* reprezentuje, či sa našla potencionálna pomenovaná entita. Toto je využívané v rámci automatickej opravy pravopisu. Položka *spell* určuje, či bola prevedená automatická oprava pravopisu, a položka *spell_from* udržiava súčasné pozície v trie po prevedení automatickej opravy pravopisu.

4.3.2 Spracovanie parametrov

V prípade zadania parametra *-n* sa načítajú dáta z textovej formy slovníka, upravia sa na nový formát špecifikovaný v 4.2.2, následne sa vytvorí slovník používaný v rámci programu.

V prípade, že nie je zadaný parameter -n, sa z parametru *-d SUBOR* otestuje reťazec SUBOR na koncovku, kde koncovka ".ct" znamená *cedar tree* a daný slovník je uložený do binárnej formy pomocou knižnice *cedar*, a koncovka "dct" znamená *darts clone tree* a daný slovník je uložený do binárnej formy pomocou knižnice $darts - clone$. Toto umožňuje jednoduché udržiavanie koherencie v prípade vylepšenia systému ďalšou knižnicou kompatibilnou s rozhraním DARTS. Následne sa súbor zo slovníkom otvorí pomocou adekvátnej knižnice a načíta do programu.

Parameter *-w SUBOR* určuje zapísovací súbor pre slovník, a znova na základe koncovky v reťazci *SUBOR* je určená knižnica použitá pre zapísanie slovníka do súboru.

Parameter *-f SUBOR* určí, či sa otvorí vstupný stream zo súboru *SUBOR* alebo zo štandardného vstupu.

Následne sa vytvorí inštancia triedy *figa_cedar*, kde sa konštruktoru predajú jednotlivé parametre určujúce zapnutie špecifických funkcií systému a zavolá sa vyhľadávacia funkcia triedy *figa_cedar*, ktorej sa predá načítaný slovník a vstupný stream ako parametre.

4.3.3 Algoritmus pre spracovanie textovej formy slovníka

Textová forma slovníka sa spracováva po riadkoch, ktoré sú načítané, a následne ak je na riadku iba jeden index, sa oddelí, uloží do poľa asociovaných hodnôt a výsledný reťazec sa uloží do poľa pre uloženie do slovníka. Ak je na riadku viacero indexov, sú oddelené od riadku, a následne sa po jednom vytvárajú jednotlivé reťazce, kde prvý je pôvodný reťazec, a následne reťazce majú ako sufix vždy o jeden symbol s ordinálnou hodnotou 1 navyše, so základným sufixom troch takých symbolov.

Nakoniec sa obe polia zotriedia na základe reťazcov, keďže knižnica *darts-clone* vyžaduje pri budovaní slovníka zotriedené kľúče.

4.3.4 Algoritmus pre spracovanie textu

Algoritmus začína inicializáciou kontextu a fronty spracovávaných slov. Následne pozostáva z while cyklu, ktorý skončí po spracovaní všetkých slov na vstupe.

Vo while cykle má algoritmus dve fázy, ktoré sa priebežne striedajú. V prvej fáze sa načítavajú nové slova na koniec fronty a priebežne sa spracovávajú. V druhej fáze sa znovu spracovavajú slova od začiatku fronty, kým sa nespracujú všetky slova vo fronte a následne sa pokračuje znova prvou fázou.

Počas načítavania slova zo štandardného vstupu sa inkrementuje poradové číslo, a počiatočné a koncové poradové číslo slova sa uložia do kontextu pre slovo. Slová sa berú ako postupnosť alfanumerických znakov, pričom sa oddelujú diaktritikou a bielymi znakmi.

Počas spracovania slova v prvej fáze sa zobere slovo a uzol uložený v súčasnom kontexte, a pomocou funkcie *traverse* aplikuje na slovník, kde sa návratová hodnota a novo dosiahnutý uzol uloží do kontextu slova, a následne sa zistí, či cesta v slovníku pokračuje medzerou. Ak áno, pokračuje sa v prvej fázi. Ak nie, spustí sa algorimtus pre spracovanie výsledkov a po ňom sa prejde do druhej fázy.

Opakované spracovanie počas druhej fázy je podobné, ale iba sa upravuje návratová hodnota a uzol uložené v kontexte pre každé slovo. V druhej fáze sa pokračuje, kým sú vo fronte nespracované slová.

4.3.5 Algoritmus pre spracovanie textu s automatickou opravou pravopisu

Toto je variácia algoritmu pre spracovanie textu, s úpravou spracovania slova, ktoré sa rozdelilo na dve fázy a pridanie kontroly pravopisu.

Algoritmus ma jeden hlavný predpoklad, a to, že vstup je pravdepodobne správny, čiže pravopis sa automaticky nekontroluje pre každé spracovávané slovo. Algoritmus is uchováva údaj o začiatočnom písmene potencionálnej pomenovanej entity zloženej z postupnosti slov a údaj, či pre nejaký prefix práve spracovavanej postupnosti sa nenašiel odpovedajúci kľúč v slovníku. Algoritmus automatickej kontroly pravopisu sa potom spustí len nad postupnosťou začínajúcou veľkým písmenom a bez prefixu odpovedajúcemu nejakému kľúču v slovníku a neexistencii cesty v trie pomocou práve spracovávaneho slova.

Spracovanie slova v algoritme je rozdelené na dve fázy, prvá odpovedá spracovaniu slova v pôvodnom algoritme pre spracovanie textu, a druhá fáza je upravená verzia prvej, ktorá prebieha, ak bola nad práve spracovávanou postupnosťou slov prevedená automatická oprava pravopisu. Druhá fáza spočíva v aplikovaní slova na viacero uzlov, ktoré reprezentujú možné pozície v trie po automatickej oprave pravopisu. Ak existuje cesta v trie z daných uzlov, vrátené uzly sa uložia do kontextu slova ako nové východzie uzly. V prípade vratenia asociovaných hodnôt sa uloží údaj o ich existencii do kontextu slova a použije sa pri spracovaní výsledkov.

4.3.6 Algoritmus pre spracovanie výsledkov a ich výpis

Algoritmus pre spracovanie výsledkov sa zavolá v prípade, že sa pri spracovaní textu nenájde cesta v trie pomocou spracovávaného slova. Potom sa na základe parametrov programu zavolá jedna z troch variacií algoritmu, základná variácia, variácia pre automatické opravenie pravopisu a variácia pre automatické doplnenie.

V základnej variácií algoritmus prechádza frontu spracovávaných slov odzadu, a hľadá prvé slovo s asociovanou hodnotou a prepisuje návratové hodnoty slov na unikátnu zarážkovu hodnotu, až kým na ňu nenarazí. Následne získa všetky asociované hodnoty pre uzol uložený s asociovanou hodnotou, zotriedi ich a odstráni duplikáty a následne vypíše. Potom vypíše poradové čísla. Potom začne algoritmus prechádzať frontu od začiatku a vypisovať jednotlivé slová a prepisuje ich návratovú hodnotu na zarážkovu, až kým nenarazí na slovo zo záražkovou hodnotou, ktoré už nevypíše, ale jeho hodnotu prepíše na nezáražkovu hodnotu. Následne vymaže prvé slovo z fronty. Ak nie je povolené prekrývanie pomenovaných entít, tak sa následne odstránia aj všetky slova od začiatku fronty po prvé slovo s nezáražkovu hodnotou.

Variácia pre automatické opravenie pravopisu je v princípe rovnaká ako základná variácia, ale líši sa získavaním asociovaných hodnôt. Tu sa namiesto získavania hodnôt z jedného uzlu získavajú všetky asociované hodnoty zo všetkých uzlov uložených v kontexte slova.

Variácia pre automatické doplňovanie sa líši viac, pri prechádzaní fronty spracovávaných slov odzadu je rovnaká, ale zastaví sa aj na návratovej hodnote *NO_VALUE* a nielen na asociovanej hodnote. Pri prechádzaní zpredu sa slová nevypisujú, ale spájajú pomocou medzery a uloží sa kontext posledného slova, ako poradové číslo a návratoví hodnota. Ak návratová hodnota nie je asociovaná hodnota, spustí sa algoritmus automatického doplňovania textu, ktorý do poľa uloží všetky rozšírenia postupnosti spracovávaných slov aj s ich asociovanými hodnotami. V prípade, ak návratová hodnota je asociovaná hodnota, do poľa sa uloží iba pôvodný reťazec a jeho všetky asociované hodnoty. Následne sa vypíšu všetky položky z poľa a vyčistí sa fronta ako v základnej variácií.

4.3.7 Algorimtus automatického doplňovania textu

Spustí sa v rámci algoritmu pre spracovanie výsledkov, ak je povolené automatické doplňovanie textu. Je to jednoduchý rekurzívny algoritmus, ktorý zobere vstupný uzol a iteruje cez všetky char symboly s ordinálnou hodnotou väčšsou ako 29 a snaží sa pomocou nich najsť oblúk z daného uzla. Ak nájde, skopíruje vstupné slovo, pridá k nemu daný symbol a zavolá znovu algoritmus s nájdeným novým uzlom a upraveným slovom ako parametrami. V prípade, že pre nové slovo je kľúč zo slovníka, tak sa získajú všetky jeho asociované hodnoty a dané slovo aj s hodnotami sa uloží do poľa, ktoré je nakoniec vrátené ako výstup.

4.3.8 Algoritmus automatického opravovania pravopisu

Algoritmus automatického opravovania pravopisu sa využíva na posledné spracovávané slovo, pre ktoré nebola nájdená cesta v trie. Spočíva v aplikovaní jednoduchých základných operácií a to vloženia, výmeny a odstránenia symbolu. Týmto sa nájdu všetky potencionálne cesty v trie, ktoré sa líšia od slova na vstupe o hammingovu vzdialenosť jedna. Ďalej sa pokúsi vymeniť susedné písmená, čo je jednoduchá a častá chyba s hammingovou vzdialenosťou dva.

Operácie výmeny a vloženia symbolu fungujú generovaním všetkých možných symbolov a postupným skúšaním každého z nich.

Ak počas aplikovania týchto operácií sa nájdu asociované hodnoty, dané uzly sa uložia do poľa, ktoré sa uloží do kontextu slova. Po každom nájdení možnej cesty sa zistí, či z daného uzlu ide oblúk zo symbolom medzery. Ak áno, daný uzol sa uloží do štruktúry kontextu spracovania textu.

Daný algoritmus sa zopakuje pre každý potencionálny počiatočný uzol predaný algoritmu s tým, že sa dané operácie neaplikujú na všetky symboly v slove. Aplikujú sa len po pozíciu, kde bola nájdená chyba, čo je zistené pomocou *traverse*. Ak sa aplikovalo celé slovo na uzol a našla sa cesta v trie, tak to znamená, že sa nenašla nasledujúca medzera, tak sa pridá koncový uzol slova ako možné pokračovanie, čím sa dosiahne možnosť spojiť dve nasledujúce slová.

4.3.9 Podpora UTF-8

Väčšina programu funguje bez špecialnej podpory UTF-8, keďže algoritmy pracujú s charovými symbolmi s ľubovoľnou hodnotou, a tým technicky dokážu podporovať ľubovoľné kódovanie s podmienkou, že slovník aj vstupný text majú rovnaké kódovanie.

Špecialne podpora pre UTF-8 je nutná iba v dvoch prípadoch. V rámci počítania poradového čísla po symboloch, kedy sa nemôže jednoducho inkrementovať počítadlo s každým načítaným symbolom, ale musia sa dekódovať. Druhý prípad je automatická oprava pravopisu, kde sa v rámci operácií vloženia a nahradenia generujú všetky možné symboly, čo by ale v prípade UTF-8 znamenalo niekoľko miliónov, čo je nepraktické. Namiesto toho sa UTF-8 symboly generujú po bajtoch a ak sa pre nejaký bajt nenájde cesta, negenerujú sa žiadne jeho naväzujúce bajty, čím sa počet vygenerovaných symbolov pre jeden UTF-8 symbol zníži približne na 300.

Kapitola 5

Návrh a implementácia morfologického analyzátoru

Súčasťou práce je návrh a implementovanie morfologického analyzátoru, s využitím stručných datových štruktúr. Ako predloha sa mohol využiť predtým vyvíjaný morfologický analyzátor.

5.1 Návrh morfologického analyzátora

Systém má prevádzať morfologickú analýzu nad vstupným textom a na základe parametrov urobiť morfologickú analýzu každého slova. Systém využíva knižnicu *cedar*, a jej funkcie $traverse$ a pomocné funkcie *begin*, *next* a *suffix*, viď. sekciu 4.2.1.

5.1.1 Pôvodný morfologický analyzátor

Pôvodný morfologický analyzátor je rozdelený na dve časti, knižnicu *libma*, ktorá implementuje funkcionalitu morfologickej analýzy, a fasádu, ktorá spracováva parametre, parsuje text a vypisuje výsledky.

Knižnica *libma* využíva knižnicu p. Daciuka fsa, a je rozdelená na niekoľko tried, ktoré implementujú jednotlivé tvaroslovné operácie, a triedu pre modifikáciu výsledkov do preurčených datových štruktúr.

Libma využíva viacero rozdielnych slovníkoch, každý špecifický pre jednu alebo viacero tried. Každý obsahuje iné inštrukcie pre iné operácie zo slovami, ako skloňovanie, tvaroslovie pomocou derivácie alebo tvaroslovie pomocou spájania.

Princíp jednotlivých tried je podobný, ale líšia sa hlavne postupnosťou spracovania údajov, čo je dané použitým typom slovníka, a formátom výstupu. Hlavnou úlohou každej triedy je zobrať vstupné slovo, nájsť ho v slovníku, upraviť a vrátit upravené slovo spolu s morfologickými kategóriami.

5.1.2 Využitie pôvodného systému

Pri návrhu a implementácií nového morfologického analyzátoru sa znova objavila otázka využitia pôvodného kódu ako pri návrhu systému pre spoznávanie pomenovaných entít. V tomto prípade som sa rozhodol pre zachovanie väčšej časti kódu a logiky a prepísať iba jednotlivé volania knižnicu fsa.

Rozhodol som sa zachovať implementáciu fasády, keďže bola úplne oddelená od implementácie logiky a funkcionality morfologickej analýzy a nemusel som ju nijak upravovať, len zachovať jednoduché rozhranie, ktoré spočíva vo verejných funkciách jednotlivých tried knižnice *libma*, ktorým sa predáva reťazec spracovávaneho slova, poprípade ďalšie reťazce reprezentujúce parametre vyhľadávania. Takže implementácia spočívala v úprave alebo reimplementácií knižnice *libma*.

Rozhodol som sa zachovať väčšinu pôvodnej logiky a algoritmov, keďže z dokumentácie sa nedala zistiť plná funkcionalita systému, a pri reimplementácií som chcel zachovať jej plnú funkčnosť. Ďalej prechádzanie pomocou knižnice *fsa* bolo implementované tak, že sa dalo ľahko emulovať pomocnými funkciami knižnice *cedar*, takže väčšsina algoritmov pôvodného systému bola kompatibilná.

Rozhodol som sa neupraviť logiku iba pomocou výmeny *fsa* knižnice, z rovnakého dôvodu ako pri systéme na spoznávanie pomenovaných entít, čo znamená rýchlosť.

5.1.3 Formát dát v slovníkoch

Využívajú sa štyri rôzne slovníky, kde každý má vlastný formát dát. Informácie sú uložené priamo kľúčoch a nemajú žiadne asociované hodnoty. Slovníky sa vytvárajú pomocu skriptov z viacerých súborov, napríklad súbor obsahujúci dvojice lexém a lem. Tento štýl nie je závislý na jazyku a funguje pre väčšinu skloňovacích jazykov. Súbory potrebné k vytváraniu slovníkov sa dajú získať napríklad z lingvistických inštitúcií daných jazykov.

V rámci morfologických slovníkov sa využíva morfologická anotácia, čo je zakódovanie morfologických kategórií do symbolov. Zakódované sú informácie ako rod, číslo, pád, čas, typ slova, napríklad podstatné meno a podobne. Ďalej sa využíva KEndings na zakódovanie morfologických operácií.

Používajú sa nasledujúce typy slovníkov:

- derivačný slovník, ktorý obsahuje dva druhy záznamov, záznam o derivácií slova z koreňa a reverznú operáciu zo slova na koreň. Ak záznam v slovníku začína na @, tak daný tvar slova musí mať predponu. Má nasledujúci formát :

  slovný tvar "+" vzor slovného tvaru "+" typ odvodenia "+" anotácia väzby "+" Kendings "+" vzor nového slovného tvaru.

  Pre ukážku viď. obrázok 5.1.

  tvar + vzor + smer + typ + kód + vzor
  Absolon+filosof+D+2.2.0.0+Aúv+otcúv
  Absolonúv+otcúv+B+2.2.0.0+C+filosof

  Obr. 5.1: Ukážka formátu dát v derivačnom slovníku.

- slovník morfologických tvarov obsahuje slovný tvar a zakódovanú operáciu pre zmenu na lemu. Má nasledujúci formát:

  slovný tvar "+" KEndings "+" anotácia morfologických kategórií "+" poznámka "+" vzor lemy.

  Pre ukážku viď. obrázok 5.2.

- slovník lem obsahuje lemy a zakódovanú operáciu pre zmenu na slovný tvar. Má nasledujúci formát:

tvar + Kendings + anotacia + pozn. + vzor
Absolón+A+k1gMnSc1++filozof
Absolónama+D+k1gMnPc7wH+sA+filozof
Absolóna+B+k1gMnSc2++filozof
Absolóna+B+k1gMnSc4++filozof
Absolóne+B+k1gMnSc5++filozof

Obr. 5.2: Ukážka formátu dát v slovníku morfologických tvarov.

lema "+" anotácia morfologických kategórií "+" poznámka ["!"predpona] "+" KEndings "+" vzor tvaru slova.

Pre ukážku viď. obrázok 5.3.

lemma + anotacia + [! predpona] pozn. + Kendings + vzor
krásný+k2eAgMnSc2d1++ Béhо+nový
krásný+k2eAgMnPc1d2+sA+Bější+nový
krásný+k2eAgMnPc1d3+!nej+Bější+nový
krásný+k2eNgMnSc1d3+!nejne+Bější+nový

Obr. 5.3: Ukážka formátu dát v slovníku lem.

- slovník čisloviek má jedinečný tvar s tým, že je rozdelenný na tri slovníky, slovník tokenov, gramtický slovník pravidiel a slovník koncoviek. Využíva sa na spracovanie čísloviek.

  Slovník tokenov má formát:

  T + token "#" terminál "#" akcia

  Gramatický slovník pravidiel pre vytváranie čísloviek má formát:

  G + terminál "#" terminál "#" .. "#" typ číslovky.

  A slovník koncoviek má formát:

  E + typ číslovky + akcia "#" sufix "+" KEndings "+" anotácia morfologických kategórií "+" vzor slova

  Pre ukážku viď. obrázok 5.4.

5.1.4 Použité knižnice

V rámci implementácie sa nedala použiť knižnica *darts-clone*, keďže nemá funkcionalitu na prechádzanie celého podstromu, a jedinou možnosťou je vyskúšať všetky možné symboly v každom uzle, čo je neefektívne a niekoľko násobne pomalšie ako pomocné funkcie knižnice *cedar*.

Pôvodný morfologický analyzátor využíva upravenú knižnicu *fsa*, ktorá je optimalozovaná a nieje schopná ukladať hodnoty ku kľúčom, čo nieje problém, keďže informácie sú zakódované ako cesta v trie. Toto dovoľuje extrémnu kompiláciu slovníka, ale je to za cenu rýchlosti prehľadávania, kde má tú istú nevýhodu ako knižnica *fsa* použitá vo FIGA, viď sekciu 4.1.1.

T token # terminál # akcia
Tdva#T#+2
Ta#H#N

G terminál#terminal#..# typ
GT#sc
GT!H.T#sc

E typ akcia # sufix + KEndings + anotácia + vzor
EscN#+A+k5eN+nový
Esc*10#dsať+A+k5eN+nový

Obr. 5.4: Ukážka slovníku pre spracovanie čísloviek. Pomocou tohto slovníka sa dá spracovať čislo dva alebo číslo dvaadvadsať.

Oproti tomu *cedar* ma obrovské pamäťové nároky, čo ale vyvažuje rýchlosťou prechádzania trie. Otázkou je ako veľmi bude pamäťová náročnosť spomalovať systém.

5.2 Implementácia morfologického analyzátoru

Implementácia sa skladala hlavne z úpravy knižnice *libma* tak, aby nevyužívali knižničné volania z knižnice *fsa*, ale *cedar*. K tomu bolo treba, čo najmenej zasahovať do logiky pre zachovanie funkčnosti a zároveň odstrániť prebytočný kód.

Knižnica *libma* má triedu fsa, ktorá slúži ako prototyp pre ostatné triedy, a definuje spoločné premenné, ako list slovníkov, globálne premenné a rôzne premenné pre parametre. Knižnica obsahuje ďalšie dve základné triedy *deriv_fsa* a *morph_fsa*, ktoré sú obe odvodené od triedy *fsa*. Obsahuje tiež pomocné triedy a knižnice ako *nstr* pre prácu s reťazcami, *morhp_config* pre nastavenie konfigurácie alebo *xstr* pre transformáciú dát do formy pre fasádu. Ďalej obsahuje triedy odvodené z *morph_fsa* a to *get_form_fsa*, *numbers_fsa* a *all_words_fsa*. Všetky tieto triedy spolu s *deriv_fsa* a *morph_fsa*, s výnimkou *numbers_fsa*, pracujú principiálne na základe rovnakého algoritmu.

Algoritmus je prakticky rovnaký pre nový systém ako pre pôvodný systém. Líši sa hlavne implementačnými podrobnosťami. Algoritmus zobere vstupné slovo a aplikuje ho na počiatočný uzol v slovníku. Ak sú zadané ďalšie parametre, tak sa aj tie následné aplikujú na vrátený uzol a pokúsi sa nájsť cesta v trie. Ak sa nájde cesta v trie, tak sa následne získajú všetky kľúče dosiahnutelné z daného uzla. Tieto kľúče sa následne rozparsujú, spracujú a aplikujú sa na vstupné slovo a vytvorí sa nový reťazec, ktorý sa predá knižnici *xstr* a transformuje sa na datový formát pre fasádu morfologického analyzátora.

5.2.1 Trieda deriv_fsa

Deriv_fsa je trieda, ktorá pracuje s derivačným slovníkom, dokáže odvodené slovo previesť na koreňové slovo a koreňové slovo previesť na odvodené slovo. Ako rozhranie využíva funkciu *get_word*, ktorá má štyri vstupné parametre, a to vstupné slovo, hľadaný vzor, hľadaná anotácia väzby a smer operácie.

Trieda podobne ako v algoritme aplikuje slovo na počiatočný uzol. Ak je zadaný vzor, tak ho aplikuje na vrátený uzol. Následne sa postupne preberú všetky kľúče, do ktorých sa dá dostať z vráteného uzla.To isté sa spraví, ak nie je zadaný hľadaný vzor, ale zobere sa pôvodný uzol. Tieto kľúče sú ďalej spracovávané po sekciách oddelných symbolom "+", a v prípade, že kľúč nevyhovuje nejakej podmienke, tak sa daný kľúč ignoruje.

Najprv sa porovná smer odvodenia, keďže sú v slovníku dva druhy záznamov. Ak je zadaná hľadaná anotácia väzby, tak sa porovná ďalšia časť kľúča s hľadanou anotáciou, ak vyhovuje, tak sa uloží a pokračuje sa v zpracovávaní. Ak nie je zadaná hľadaná anotácia, tak sa sekcia iba uloží a pokračuje sa. Z ďalšej sekcie sa spracuje KEndings kódovanie a odstráni sa sufix hľadaného slova a pripojí zvyšok kľúča. Nakoniec vráti reťazce v tvare:

vzor vstupného slova + smer odvodenia + anotácia väzby + odvodenné slovo + vzor odvodeného

5.2.2 Trieda morph_fsa

Morph_fsa je trieda, ktorá pracuje s morfologickým slovníkom a dokáže skloňovať a časovať lemy. Ako rozhranie využíva funkciu *morph_word*, ktorá má jediný parameter a to hľadané slovo. Ďalej ponúka rozhranie pre odvodené triedy pre spoločné funkcie a to spracovanie KEndings a vytvorenie reťazca pre predanie na transformáciu do knižnice *xstr*.

Algoritmus je prakticky identický so základným algoritmom. Nájde slovo v slovníku, zobere všetky kľúče, ktoré ho majú ako prefix, a spracuje ich. Konkrétne v prvej sekcii zobere KEndings, odstráni požadovane dlhý sufix hľadaného slova a nahradí ho zvyškom kľúča. A následne vráti všetky možné lemy slova.

5.2.3 Trieda all_words_fsa

All_words_fsa je trieda, ktorá pracuje s morfologickým slovníkom a pre hľadané slovo vygeneruje všetky možné tvary slova. Ako rozhranie využíva funkciu *all_words*, ktorá má jediný parameter a to hľadané slovo. Algoritmus je rovnaký ako v triede *morph_fsa*.

5.2.4 Trieda get_form_fsa

Get_form_fsa je trieda, ktorá pracuje so slovníkom koreňov a pre hľadané slovo nájde všetky slovné tvary odpovedajúce hľadaným morfologickým kategóriam zadaných pomocou anotácie morfologických kategórií. Ako rozhranie využíva funkciu *get_form*, ktorá má päť vstupných parametrov a to hľadané slovo, povolená anotácia kategórií, zakázaná anotácia kategórií, povolené poznámky a zakázané poznámky.

Algoritmus je podobný algorimtu z *deriv_fsa*. Nájde hľadané slovo v slovníku a po jednom prejde a spracuje všetky kľúče v slovníku, ktorým je hľadané slovo prefixom. Spracuje kľúč po sekciach, spracuje anotáciu morfologických kategórií a zistí, čí vyhovuje zadaným parametrom, potom spracuje poznámky, a ak všetko vyhovuje, vytvorí nový reťazez pomocou KEndings a uloží ho pre transformáciu knižnicou *xstr*.

5.2.5 Trieda numbers_fsa

Numbers_fsa je trieda, ktorá pracuje so slovníkom čísloviek, a dokáže určiť hodnotu reprezentovanú slovom, ak nejakú reprezentuje. Ako rozhranie využíva funkciu *morph_word*, ktorá ma jediný parameter a to hľadané slovo.

Využíva iný algoritmus ako ostatné triedy v knižnici *libma*. Používa tri slovníky, uložené ako jeden, kde ich oddeľuje prefix. Kľúče tokenov majú prefix “T“, kľúče s gramatickými pravidlami majú prefix “G“ a kľúče, ktoré obsahujú číslovky s akciou a morfologickými informáciami má prefix “E“.

Algoritmus iteratívne parsuje hľadané slovo, ktoré je rozparsované na tokeny, až kým nespracuje celé slovo. K tomu využíva funkciu *morph_word_in_gram*, ktorá má ako parametre sufix slova na zpracovanie, pôvodné slovo, slovník, vstupný uzol, 2 pomocné parametre a 3 parametre predávajúce hodnotu čislovky pomocu hodnoty, hodnoty počítadla a hodnoty prefixu.

Tokeny parsuje po symbole, a aplikovaním symbolu do slovníka tokenov zisťuje, či dané slovo obsahuje nejaký token. Zo slovníku sa potom ziskajú všetky sufixy kľúčov s prefixom obsahujúci token nasledovaný “#“. Sufix má v prvej sekcii terminál, nasledovaný symbolom “#“ a akciou zakódovanou do sufixu. Ak sa typ terminál z prvej sekcie dá aplikovať na uzol v gramatickom slovníku, tak sa uloží dosiahnutý uzol. Následne sa odskúšajú možné symboly spojujúce terminály v slovníku, kde každý symbol reprezentuje iný typ následného spracovania. Existuje 5 symbolov vyjadrujúce typ následného spracovania. Konkatenácia tokenov, získaný token je prefix nasledujúceho tokenu alebo token je prefix zlomku, alebo nasledujúci token bude zlomok, alebo sufix zlomku alebo získaný token je možný koniec čísla.

- “. “ označuje konkatenácia tokenov, znamená načítanie ďalšieho tokenu, čiže napríklad v slove “dvaadvadsať“ sa načíta “dva“ a potom “a“, ktoré sa preskočí a tokeny “dva“ a “dvadsať“ sa konkatenujú.
- “!“ označuje, že token je prefix iného tokenu, tak sa zavolá tá istá funkcia, kde sa uloží hodnota tokenu ako hodnota prefixu, hodnota sa nastaví na 0 a numerátor ostane rovnaký.
- “@“ označuje, že nasledujúci token bude zlomok, tak sa hodnota uloží ako hodnota počítadla, hodnota sa nastaví na 0 a hodnota prefixu ostane rovnaká.
- “/“ znamená, že token je prefix zlomku, čiže sa z hodnoty, hodnoty počítadla a hodnoty prefixu vypočíta nová hodnota, a predá sa novému volaniu funkcie, a zvyšné dve hodnoty sa dajú na 0.
- “#“ označuje jednoduchú separáciu, napríklad čislo “16“, sa načíta ako “1“ a “6“.
- “#“ označuje, že token je posledným tokenom v čisle a zavolá sa funkcia na kontrolu zvyšku slova.

Ešte sa v rámci algoritmu spracovávajú akcie pre tokeny, a je ich celkom päť:

- NOP, znamená nič nerobiť, napríklad pre token bodky alebo inej spojky ako a
- ADD, sčíta súčasnú hodnotu zo súčasným tokenom
- MULT, zvyšuje radix
- FRACT, vypočíta zlomok, ako hodnota počítadla deleno hodnota
- INDEFINITE, nedefinované , da hodnotu na NAN

Kontrola zvyšku slova extrahuje zvyšný sufix, kde je typ číslovky, ktorým prejde v slovníku koncoviek, načíta akciu a pomocou nenačítaného zvyšku vstupného slova nájde podstrom, v ktorom sú uložené možné lemy slova Následne sa prevedie morfologická analýza a získajú sa všetky lemy a hodnota vstupného slova sa uloží, a je možné k nej prístúpiť pomocou funkcie *get_value*.

Kapitola 6

Testy a experimenty

Cieľom práce je implementácia systému pre spoznávanie pomenovaných entít a morfologický analyzátor za využitia stručných datových štruktúr. Systém pre spoznávanie pomenovanných entít má byť rýchlejší a efektívnejší ako pôvodne používaný systém FIGA. Pre implementáciu nového systému bolo dôležité zachovanie fukncionality, efektívne ukladanie slovníka, rýchla práca zo slovníkom a schopnosť pracovať zo slovníkom, ktorý má viac ako desať milónov záznamom.

Cieľom práce je tiež implementácia rýchlejšieho a efektívnejšieho morfologického analyzátoru. Pre morfologický analyzátor je podstatné efektívne ukladanie slovníka a rýchlosť práce s ním.

6.1 Testy potencionálnych knižníc

Pre priestorovo efektívnejšiu a rýchlejšiu implementáciu som mal vybrať nové knižnice z potencionálnych knižníc. Počas implemtácie som vykonal niekoľko testov a experimentov na zistenie veľkosti využívaného miesta a rýchlosti knižníc pre porovnanie s pôvodnou knižnicou *fsa*, ako aj pre porovanie s experimentami prevedenými autormi knižnice *cedar*.

6.1.1 Rozhrania knižníc

Najprv som musel určiť, ktorú implementáciu nový systém môže využiť. Po stiahnutí jednotlivých knižníc, ich nainštalovaní a vytvorení jednotlivých rozhraní som urobil jednoduché testy na ich vlastnosti, viď. tabuľku 6.1.2, kde som ako potencionálnu knižnicu vylúčil knižnicu *marisa 2*, keďže nemá dostatočné rozhranie a funkcionalitu. Vylúčil som aj ďalšie knižnice, ktoré nebolo možné ani preložiť.

6.1.2 Rýchlosť a efektivita knižníc

Zvýšné knižnice som potom porovnal a zistil ich rýchlosť pri budovaní slovníka a vyhľadávaní v ňom, viď. obrázok 6.1.2. Z týchto testov vyšly dva konečný kandidáti, a to knižnice *cedar* a *darst-clone*.

Knižnica *cedar* dosahovala najrýchlejšie časy pri budovaní slovníka a konkurenčné časy s knižnicou *darts-clone* pri vyhľadávaní v slovníku. Knižnica *darts-clone* dosahovala najlepšie časy pri vyhľadávaní a využíva aj najmenej pamäte pri práci zo slovníkom, preto som sa rozhodol vytvoriť implementáciu podporujúcu obe knižnice, čo bolo uľahčené tým, že majú skoro rovnaké rozhrania.

knižnica	Úprava slovníka	ExactMatch	PrefixMatch	NextChildren	Traverse
marisa 2	Nie	Nie	Nie	Nie	Nie
dawgic	Úplne prebudovanie	Áno	Áno	Nie	Áno
darts-clone	Úplne prebudovanie	Áno	Áno	Nie	Áno
cedar	Áno	Áno	Áno	Nie	Áno
fsa	Úplne prebudovanie	Nie	Nie	Áno	Nie

Tabuľka 6.1: Štatistika vlastností jednotlivých knižníc. *Marisa 2* má nedostatočné rozhranie a preto bola vylúčená z ďalších testov.

knižnica	Použitá pamäť	Vybudovanie	Najdenie	Nenájdenie	Mix
dawgic	551904 kB	14,4 M tikov	107 K tikov	1299 tikov	62 K tikov
darts-clone	497000 kB	7,4 M tikov	63 K tikov	548 tikov	38 K tikov
cedar	997000 kB	2,1 M tikov	93 K tikov	348 tikov	51 K tikov

Tabuľka 6.2: Porovnanie veľkostí a rýchlosti vyhľadávania pre implementácie s jednotlivými knižnicami. Kde *darts-clone* je najrýchlejší vo vyhľadávaní a *cedar* pri budovaní, podobne ako to vychádza v štatistikách autorov *cedaru*, viď. sekciu 3.4.5. Rýchlosť bola meraná v počte procesorových cyklov.

6.2 Prenositelnosť knižníc

Obe knižnice sú plne prenositeľné, keďže pracujú so základnými C funkciami a využívajú iba minimum štandardných knižnic.

6.3 Porovnanie efektivity a rýchlosti implementácií NER

Po implementovaní systému som spravil experimenty porovnávajúce nový systém oproti pôvodnému systému. Oproti pôvodnému systému bol nový systém schopný načítať slovník aj o 15 000 000 položkách a teoreticky môže načítať až niekoľko miliard položiek.

Veľkosť vytvorených slovníkov je vidno v tabuľke 6.5, kde je vidno, že knižnica *cedar* má väčšie pamäťové nároky pri ukladaní na disk ako aj pri práci zo slovníkom. Naopak knižnica *darts-clone* je najefektívnejšia pri ukladaní do binárnej reprezentácie slovníka, a dosahuje veľkosť menšiu než pôvodná textová reprezentácia.

Pri spracovaní textu a práci zo slovníkom všetky systémy využívajú iba pamäť o niečo väčšiu ako je uložená binárna reprezentácia použitého slovníka.

Knižnica *cedar* dosahuje najmenší čas pri vytváraní slovníkov, kedy potrebuje iba polovičný čas oproti *darts-clone*.

Ohľadom porovnania rýchlosti nového a pôvodného systému som urobil testy pomocou programov *time* a *gprof*, viď. obrázok 6.3.

knižnica	Čas - time	Čas - time s -pg -g pri preklade	Čas - gprof
darts-clone	2,19 s	10,5 s	3 s
cedar	2,45 s	10,8 s	3 s
fsa	6,4 s	29,7 s	10 s

Tabuľka 6.3: Porovnanie rýchlosti spracovania textového súboru so 750 000 entitami. Nový systém potrebuje iba 30 percent času oproti pôvodnému systému.

knižnica	750 000 entít	15 000 000 entít
darts	3,4 s	87 s
cedar	1,4 s	45 s
fsa	27,3 s + 30 min	-

Tabuľka 6.4: Porovnanie rýchlosti vybudovania slovníku so 750 tisíc entitami a slovníka s 15 miliónmi entít. *Cedar* dosahuje najlepší čas, a *darts-clone* má iba dva krát pomalší, fsa zaostáva potrebuje 20 násobne dlhší čas ako *cedar*. Do tohto času sa nepočíta predspracovanie súborov, kedy *cedar* a *darts-clone* potrebujú menšie predspracovanie, *fsa* potrebuje predspracovať ešte ďalšími skriptami, ktoré pre 750 tisíc entít trvali 30 minút.

knižnica	750 000 entít	15 000 000 entít
cedar	50 MB	996MB
darts	14 MB	369MB
fsa	14 MB	-
text	17 MB	394 MB

Tabuľka 6.5: Porovnanie veľkosti slovníkov pre rôzne knižnice, *cedar* je najmenej efektívny, pričom jeho binárna reprezentácia dosahuje trojnásobnej veľkosti pôvodného súboru. Pôvodná knižnica *fsa* je efektívnejšia a dosahuje skoro veľkosti ako knižnica *darts-clone*,ktorá je najefektívnejšia, a jeho výsledná reprezentácia je menšia ako pôvodný textový súbor.

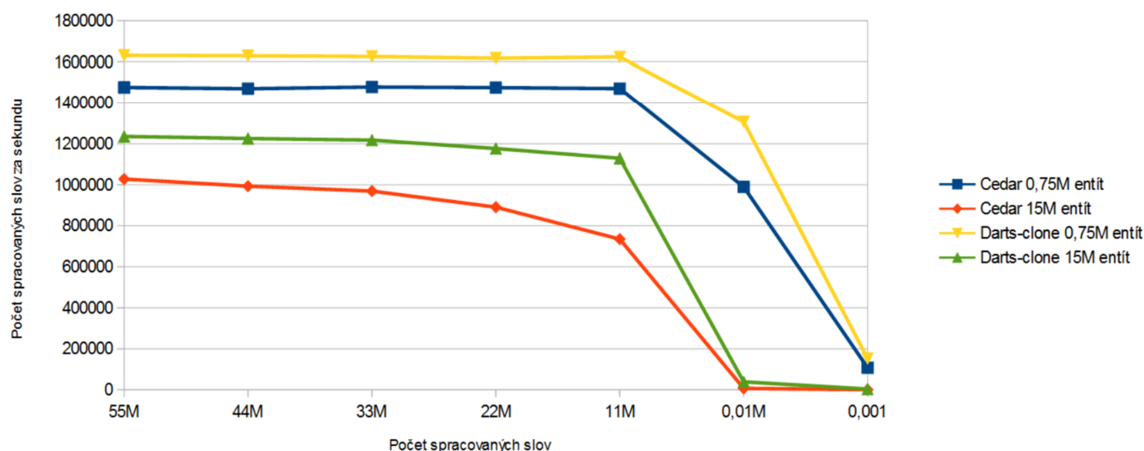
6.4 Rýchlosť systému NER

Rýchlosť nového systému je vidno na tabulke 6.6 a grafe 6.1. Je vidno, že veľkosť slovníka ovplyvňuje aj rýchlosť spracovania, a to hlavne pri menších vstupných textoch, kedy režia nahratia slovníka systémom je až 90 percent času programu. Keďže sa ale bude používať hlavne pre veľké množstvá dát, tak dopad takejto počiatočnej réžie na priemernú rýchlosť sa dá minimalizovať. Ďalej je viditeľný vplyv veľkosti binárnej podoby slovníka na rýchlosť, kde menšie slovníky dosahujú lepšie výsledky.

V tabuľke 6.7 je vidno, že aj pri vedlajších funkciách ako je automatické doplňovanie textu a automatická oprava pravopisu je nový systém rýchlejší, aj keď pomerovo o niečo spomalil, pravdepodobne kvôli neefektívnemu prehľadávaniu podstromu, kedy sa generujú všetky možné symboly.

knižnica	cedar		darts-clone	
velkosť slovníka	750 tisíc entít	15 miliónov entít	750 tisíc entít	15 miliónov entít
55 miliónov slov	37,22 s	53,48 s	33,7 s	44,5 s
44 miliónov slov	29,9 s	44,3 s	27 s	35,9 s
33 miliónov slov	22,3 s	34,05 s	20,3 s	27,1 s
22 miliónov slov	14,9 s	24,69 s	13,6s	18,7 s
11 miliónov slov	7,48 s	14,98 s	6,78 s	9,75 s
30 tisíc slov	0,037 s	5,1 s	0,028 s	0,94 s
3 tisíc slov	0,034 s	4,9 s	0,024 s	0,901 s

Tabuľka 6.6: Tabuľka časov potrebných na spracovanie daného množstva slov. Je vidno, že väčšie slovníky majú väčšiu réžiu a dosahujú pomalšie časy. Toto je nevýhodné hlavne pre knižnicu *cedar*, ktorá má najmenej efektívnu binárnu reprezentáciu slovníka, viď. tabuľku 6.5



Obr. 6.1: Graf ukazuje priemernú rýchlosť spracovania slov za sekundu, kde sa porovnávali systém s knižnicou *cedar* a s knižnicou *darts-clone*. Rýchlosť sa porovnávala aj s rôznymi veľkými slovníkmi, jeden slovník mal 15 miliónov položiek, druhý mal 750 tisíc položiek. Z grafu jasne vyplýva vplyv veľkosti binárnej podoby slovníka na rýchlosť systému.

knižnica	kontrola pravopisu	doplňovanie
darts-clone	6 s	4 s
fsa	16 s	12 s

Tabuľka 6.7: Porovnanie rýchlosti pri automatickom doplňovani textu a automatickej kontrole pravopisu. Nový systém je rýchlejší, ale je vidno, že pôvodný systém je v pomere o niečo rýchlejší ako pri jednoduchom vyhľadávaní. Hlavne vďaka efektívnejšej možnosti prehľadávať podstromi.

Z týchto testov jasne vyplýva, že *darts-clone* implementácia je lepšia ako *cedar*, keďže je rýchlejšia a využíva menej miesta čo výrazne pomáha pri obrovských slovníkoch, kedy *cedar* môže rýchlejšie obmedzený hardvérom. Výhodou *cedaru* sú ale pomocné funkcie pomocou ktorých, môže efektívnejšie robiť vedlajšie úlohy ako automatické doplňovanie a automatická kontrola pravopisu kde je potreba prechádza podstromov, čo ale nebolo implementované.

Ďalej sa ukázalo, že veľkosť slovníka ovplyvňuje rýchlosť prehľadávania a to nie len počiatočnou réžiou ale celkovo počas behu systému, aj keď sa využívajú veľmi efektívne algoritmy.

6.4.1 Porovnanie s inými implementáciami

Systém NER bol porovnaný so Standford Named Entity Tagger. Bol porovnaný pri spracovaní 350 tisíc slov, kde dosiahol približne 10-násobnú rýchlosť. Standford NET ale mal presnejšiu analýzu, kde dokázal určiť aj nezvyčajné výrazy, kde slovník zaostával a nedokázal správne identifikovať napríklad „New York FED and Treasury“, kde NER nedokázal identifikovať „FED“.

Porovnanie s ostanými systémami nebolo možné, keďže buď využívali Standford Named Entity Tagger ako AIDA, alebo ponúkali iba online demo alebo iba program so sieťovým rozhraním. Samotný NER nieje možné zapojiť do GERBIL alebo GATE bez zjednoznačovača významu a porovnania sa tak sa môžu uskutočniť až v rámci *secapi*.

6.5 Rýchlosť a efektivita morfologického analyzátoru

Keďže *darts-clone* nemala dostatočnú funkcionalitu pre morfologický analyzátor, porovnáva sa iba knižnica *cedar*.

Veľkosť binárnej reprezentácie slovníka dosahuje približne dvojnásobnú veľkosť oproti textovej reprezentácií, viď. tabulka 6.8, a ak sa zo slovníka odstránia vzory má približne rovnakú veľkosť ako pôvodná textová reprezentácia.

	28 miliónov položiek	1,5 milióna položiek
nový systém	1,9 GB	111 MB
starý systém	6,3 MB	375 KB
text	1,1 GB	60 MB
darts-clone	989 MB	52 MB

Tabuľka 6.8: Tabuľka veľkostí binárnych reprezentácií slovníkov a textovej reprezentácie. Starý systém využívajúci upravenú knižnicu *fsa* je jednoznačne najefektívnejší.

Rýchlosť pôvodného morfologického analyzátora a nového morfologického analyzátora ukazuje tabuľka 6.9, v ktorej je vidno, že nový systém dosahuje až trojnásobnej rýchlosti oproti pôvodnému.

veľkosť slovníka	28 miliónov položiek	1,5 milióna položiek
nový systém	68 s + 50 s	10 s + 10 s
starý systém	315 s + 37 s	36 s + 20 s
majka	7 s + 1 s	-
MorphoDiTa	23 s + 3 s	-

Tabuľka 6.9: Tabuľka rýchlosti systémov v tvare čas programu + čas systému, kde nový systém dosahuje až päťnásobne zlepšenie času, po zarátaní systémovej režie sa toto ale zníži iba na 2,5 násobnú rýchlosť. Iné analyzátory sú niekoľko krát rýchlejšie a keďže sa nedalo manipulovať s veľkosťou ich slovníka nedali sa porovnať s menším počtom položiek.

Pri porovnaní s inými morfologickými analyzátormi, ako je majka, je pomalší, hlavne kvôli veľkosti slovníka, viď. tabulka 6.9.

Majka dokáže slovník pre český jazyk uložit iba do niekoľkých megabajtov dát, s efektivitou 0,02 bajtu na položku. Pre porovnanie knižnica *cedar* dosahuje približne 70 bajtov na jednu položku, a knižnica darts-clone približne 36 bajtov na položku, bola skúšaná len na preklad slovníka. Knižnica *fsa* dosahuje podobné výsledky ako *majka*, kedy dosahuje približne 0,2 bajtu na položku, keďže vychádzajú z tej istej knižnice.

Efektivita pri vytváraní slovníka je daná spečializovanou knižnicou *majky*, ktorá je špecializovaná priamo na ukladanie morfologického slovníka, kde na rozdiel od knižnice *cedar* neukladá ku kľúčom žiadnu asociovanú hodnotu a využíva DAWG štruktúru a na jeden uzol potrebuje iba 1 bajt.

Ďalšou nevýhodou implementovaného morfologického analyzátora je fasáda a väčšia funkcionalita, ktorá spomaľuje spracovanie, kde majka ponúka iba dve funkcionality definované vstupným slovníkom.

Následne som urobil pokusnú implementáciu pomocou *darts-clone*, ktorá fungovala ako algoritmus pre automatické doplňovanie textu systému NER, a podľa očakávaní bola až 50-krát pomalšia. Potom som implementoval jednoduchý testovací algoritmus, založený na tom že *darts-clone* má v každom uzle hodnotu aj keď ku nemu nieje priradená, kedže priestor

pre ňu je vždy alokovaný. Vytvoril som slovník s tým, že som vytvoril prefixy kľúčov a uložil do nich hodnotu symbolu potomka a ďalšieho súrodenca. Ale aj takto optimalizovaný algoritmus dosahoval 8-krát horšie časy.

Následne som sa pokúsil upraviť knižnice tak, aby používali menej miesta odstránením hodnoty a pridať knižnici *darts-clone* funkcionalitu, ale toto bolo príliš zložité a nedosiahol som úspech, keďže asociované hodnoty sú priamo zakomponované do funkcionality, ktorá dovoľuje rýchle prechádzanie stromovej štruktúry.

6.6 Prenositeľnosť

Oba systémy sú prenositeľné medzi platformami, keďže využívajú iba základné funkcie a knižnice a nepracujú priamo so súborovým systémom. Názvy súborov dostávajú ako parametre čím sa môžu ošetrit prípadné problémy s cestami.

Kapitola 7

Záver

V diplomovej práci som sa zaoberal štúdiom a implementáciou systémov pre spracovanie textových dát a to konkrétne systémami pre spoznávanie pomenovaných entít a morfologickými analyzátormi. Cieľom mojej práce bolo naimplementovať efektívne a rýchle systémy schopné spracovávať veľké množstvá dát a zároveň pracovať zo slovníkmi a vedomostnými bázami o veľkosti desiatok miliónov položiek.

K dosiahnutiu optimálneho výsledku som využil knižnice implementujúce stručné štruktúry a rozhranie pre efektívnu prácu s nimi. Pre vybratie najoptimálnejšej knižnice som otestoval niekoľko rôznych knižníc a následne vybral najlepšie. Výsledky experimentov potvrdili výsledky dosiahnuté autormi knižníc, a na základe týchto zistení som vybral nasledujúce knižnice *darts-clone* a *cedar*, kvôli ich rýchlosti, veľkosti slovníkov, funkcionalite a prenositeľnosti.

Jednou častou mojej práce bola implementácia systému pre spoznávanie pomenovaných entít. Tento systém bol implementovaný na základe už implementovaného systému FIGA, s tým že si ponechal jeho rozhranie. Dokáže pracovať s knižnicou *cedar* ako aj *darts-clone* za využitia šablón. Využitie knižnice *darts-clone* je o trochu rýchlejšie ako *cedar* a obe potrebujú iba 30 percent času pôvodného systému pre spracovanie rovnakého počtu slov. Pri porovnávaní s inými systémami vyšli niekoľko násobne rýchlejšie.

Druhou častou mojej práce bola implementácia morfologického analyzátora, kde som použil iba knižnicu *cedar*, keďže knižnica *darts-clone* nemá dostatočnú funkcionalitu. Nový systém je efektívnejší a dosahuje až 3-násobné zrýchlenie oproti pôvodnému systému, ale stále zaostáva za inými morfologickými analyzátormi ako je *majka*, ktoré využívajú knižnice špecializované pre morfologický slovník, čím dosahujú menšiu veľkosť a lepšiu efektivitu a rýchlosť. Experimentovanie s optimalizáciou kódovania a optimalizáciou použitých knižníc bolo neúspešné.

Oba systémy sa využívajú v rámci systémov spracovania textu na FIT VUT a dosahujú lepšiu rýchlosť ako pôvodné systémy, ale pre morfologický analyzátor majú slabú efektívnosť pri ukladaní slovníkov. Pre budúci vývoj systémov by bolo najlepšie optimalizovať jednu z knižníc a špecializovať ju pre jednotlivé problémy. Poprípade využiť knižnicu dovoľujúcu efektívnejšie využitie miesta s tým, že zachová rýchlosť prechádzania štruktúry.

Systém pre spoznávanie pomenovaných entít sa môže efektívne zaradiť do systému *secapi*. Pre morfologický analyzátor sa môže zefektívniť kódovanie slovníka pre zhustenie záznamov za využitia vlastností knižníc, rozšíriť funkcionalitu napríklad o štatistický klasifikátor schopný rozlišovať nejednoznačnosti, a tým zlepšiť presnosť analýzy, alebo pre zlepšenie rýchlosti minimalizovať funkcionalitu špecializovaním systému na jeden druh analýzy a zjednodušiť prepojenia v rámci systému.

Literatúra

[1] Aoe, J.-I.; Morimoto, K.; Sato, T.: An efficient implementation of trie structures. *Software: Practice and Experience*, ročník 22, č. 9, 1992: s. 695–721.

[2] Apostolico, A.; Crochemore, M.; Farach-Colton, M.; aj.: 40 years of suffix trees. *Communications of the ACM*, ročník 59, č. 4, 2016: s. 66–73.

[3] Arroyuelo, D.; Cánovas, R.; Navarro, G.; aj.: Succinct Trees in Practice. In *ALENEX*, SIAM, 2010, s. 84–97.

[4] Auer, S.; Bizer, C.; Kobilarov, G.; aj.: *Dbpedia: A nucleus for a web of open data.* Springer, 2007.

[5] Beard, R.; Volpe, M.: *Lexeme-morpheme base morphology.* Springer, 2005.

[6] Blumer, A.; Blumer, J.; Haussler, D.; aj.: The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, ročník 40, 1985: s. 31–55.

[7] Crochemore, M.: Transducers and repetitions. *Theoretical Computer Science*, ročník 45, 1986: s. 63–86.

[8] Demaine, E. D.: Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, ročník 8, č. 4, 2002: s. 1–249.

[9] Demaine, E. D.: Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, ročník 8, č. 4, 2002: s. 1–249.

[10] Dojchinovski, M.; Kliegr, T.: Recognizing, Classifying and Linking Entities with Wikipedia and DBpedia. *WIKT 2012*, 2012: s. 41–44.

[11] Finkel, J. R.; Grenager, T.; Manning, C.: Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, Association for Computational Linguistics, 2005, s. 363–370.

[12] Gog, S.; Beller, T.; Moffat, A.; aj.: From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, s. 326–337.

[13] Gupta, A.; Hon, W.-K.; Shah, R.; aj.: Compressed data structures: Dictionaries and data-aware measures. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, IEEE, 2006, s. 213–222.

[14] Hajič, J.: Czech Morphological Analyzer v1. 2014, LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague. URL http://hdl.handle.net/11858/00-097C-0000-0023-4336-4

[15] Hajič, J.; Hlaváčová, J.: MorfFlex CZ 160310. 2016, LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague. URL http://hdl.handle.net/11234/1-1673

[16] Hirschberg, J.; Manning, C. D.: Advances in natural language processing. *Science*, ročník 349, č. 6245, 2015: s. 261–266.

[17] Jiang, J.: Information extraction from text. In *Mining text data*, Springer, 2012, s. 11–41.

[18] Karoonboonyanan, T.: An implementation of double-array trie. 2003.

[19] Liddy, E. D.: Natural language processing. 2001.

[20] Magnini, B.; Negri, M.; Prevete, R.; aj.: A WordNet-based Approach to Named Entities Recognition. In *Proceedings of the 2002 Workshop on Building and Using Semantic Networks - Volume 11*, SEMANET '02, Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, s. 1–7, doi:10.3115/1118735.1118744. URL http://dx.doi.org/10.3115/1118735.1118744

[21] McCallum, A.; Li, W.: Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, Association for Computational Linguistics, 2003, s. 188–191.

[22] Minnen, G.; Carroll, J.; Pearce, D.: Applied morphological processing of English. *Natural Language Engineering*, ročník 7, č. 03, 2001: s. 207–223.

[23] Piccinno, F.; Ferragina, P.: From TagME to WAT: a new entity annotator. In *Proceedings of the first international workshop on Entity recognition & disambiguation*, ACM, 2014, s. 55–62.

[24] Ponzetto, S. P.; Navigli, R.: Knowledge-rich Word Sense Disambiguation Rivaling Supervised Systems. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, s. 1522–1531. URL http://dl.acm.org/citation.cfm?id=1858681.1858835

[25] Revuz, D.: Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, ročník 92, č. 1, 1992: s. 181–189.

[26] Šmerk, P.: Fast morphological analysis of czech. In *Proceedings of Third Workshop on Recent Advances in Slavonic Natural Language Processing, Brno, Tribun EU*, 2009, s. 13–16.

[27] Straková, J.; Straka, M.; Hajič, J.: Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, Baltimore, Maryland: Association for Computational Linguistics,

June 2014, s. 13–18.
URL http://www.aclweb.org/anthology/P/P14/P14-5003.pdf

[28] Tsarfaty, R.; Seddah, D.; Goldberg, Y.; aj.: Statistical parsing of morphologically rich languages (SPMRL): what, how and whither. In *Proceedings of the NAACL HLT 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages*, Association for Computational Linguistics, 2010, s. 1–12.

[29] Usbeck, R.; Röder, M.; Ngonga Ngomo, A.-C.; aj.: GERBIL: General Entity Annotator Benchmarking Framework. In *Proceedings of the 24th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2015, s. 1133–1143.

[30] Yoshinaga, N.; Kitsuregawa, M.: A Self-adaptive Classifier for Efficient Text-stream Processing. In *Proc. COLING*, 2014, s. 1091–1102.

[31] Zhou, G.; Su, J.: Named entity recognition using an HMM-based chunk tagger. In *proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, Association for Computational Linguistics, 2002, s. 473–480.

Prílohy

Zoznam príloh

A Obsah CD **56**

Príloha A

Obsah CD

- ./xrajco00.pdf – technická správa
- ./doc — zdrojové súbory technickej správy
- ./work — zdrojové súbory systémov
- ./readme.txt — návod na inštaláciu