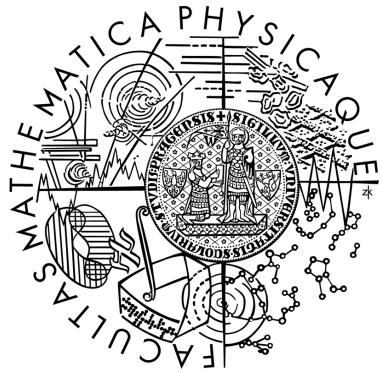
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Česlav Przywara

Metody extrakce víceslovných výrazů z textu

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: **Mgr. Pavel Pecina**

Studijní program: **Programování**

2008

Rád bych poděkoval vedoucímu práce Mgr. Pavlu Pecinovi za návrh zajímavého tématu a za čas, který mi při přípravě práce věnoval. Také bych chtěl poděkovat rodičům — za jejich trpělivost.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Těrlicku dne 3. 8. 2008 Česlav Przywara

Obsah

Název práce: Metody extrakce víceslovných výrazů z textu

Autor: Česlav Przywara

Katedra (ústav): Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. Pavel Pecina

Email vedoucího: pecina@ufal.mff.cuni.cz

Abstrakt: Cílem této práce je efektivní implementace metod (automatické) extrakce víceslovných spojení z textu, tak aby výsledný program dokázal zpracovat rozsáhlé textové korpusy o velikosti v řádu až miliard slov. Další důležitou funkcí programu je možnost ukládání kontextu pro všechny extrahované N-gramy. Pro účely práce je výsledný program implementován speciálně pro extrakci kolokací ze závislostní strukturou z Pražského závislostního korpusu (PDT), ale návrh programu umožňuje jeho snadné rozšíření.

Klíčová slova: kolokace, automatická extrakce kolokací, Pražský závislostní korpus

Title: Methods of multiword expression extraction from text

Author: Česlav Przywara

Department: Institute of Formal and Applied Linguistics

Supervisor: Mgr. Pavel Pecina

Supervisor's email address: pecina@ufal.mff.cuni.cz

Abstract: The goal of this thesis is an effective implementation of the methods of multiword expression extraction from text, so that designed program would be capable of processing large textual corpora containing up to billions of words. Additional function of the program is context tracing of extracted N-grams. For thesis purposes the program implementation is specially adjusted for collocation extraction from The Prague Dependency Treebank, but the program is designed in such manner that allows an easy future extensibility.

Keywords: collocation, automatic collocation extraction, The Prague Dependency Treebank

1 Úvod

Víceslovné výrazy jsou **přirozenou součástí jazyka**. Každý den je používáme tak běžně a intuitivně, že většina z nás se nikdy nepozastaví nad takovým „sémantickým nesmyslem“, za jaký lze považovat spojení „bílé víno“. Každý přece dobře ví, že bílé víno má zlatou barvu. Stejně tak, když se ve večerních zprávách objeví informace, že v nejmenovaném supermarketu prodávali prošlé uzeniny, nikdo si nepředstaví paní prodavačky, jak nutí uzenou kuřecí šunku pochodovat po mrazicím boxu, protože je každému samozřejmé, že prošlé uzeniny nejsou prošlé od chození. Obecněji řečeno, všichni dobře ví, že **bílé víno** a **prošlé uzeniny** jsou slovní spojení (víceslovné výrazy), která mají svůj specifický význam.

Jak je z uvedených příkladů patrné, jsou víceslovné výrazy zajímavým lingvistickým jevem. Z praktického hlediska se pak velice zajímavým stávají zejména v dnešní době fulltextových vyhledávačů a různých textově zaměřených indexovacích systémů. Při stále rostoucím objemu dostupných dat se možnost přesné specifikace parametrů hledání stává stále důležitější a zde hraje velkou roli (ne)schopnost systému správně interpretovat víceslovná spojení.

O praktickém významu **kolokací** (jak se také víceslovné výrazy označují) se lze dočíst např. v práci Pavla Peciny a Martina Holuba [1, kap. 2.4]. O této práci se hned v úvodu zmiňuji cíleně, protože se zabývá kolokacemi a jejich automatickou detekcí v českém jazyce a je de facto teoretickým základem, na kterém stojí návrh a implementace programu. V dalším textu na ni proto bude ještě několikrát odkazováno, zejména v úvodních, teoreticky zaměřených kapitolách.

Jak již bylo zmíněno v abstraktu, je cílem této práce **efektivní implementace** metod extrakce víceslovných výrazů pro použití na korpusech o velikosti v řádu miliard slov. Už z formulace zadání je zřejmé, že se bude jednat o efektivitu zejména **paměťovou** (a to často na úkor efektivity časové). Zde si dovolím ocitovat jednoho vyučujícího programování na MFF, jehož výrok tento případ nádherně ilustruje: *„Času Váš program může mít vždycky ještě dvakrát tolik, ale paměti ne.“*

Než se ovšem dostaneme ke specifikům implementace, nejdříve si v kapitole 2 uvedeme teoretické poznatky o víceslovných výrazech, které poté v kapitole 3 doplníme o teoretický pohled na metody jejich automatické detekce. V kapitole 4 popíšeme vlastnosti a chování programu z hlediska uživatele. Kapitola 5 je

věnována teoretické analýze programu, za ní pak následuje kapitola 6, jenž popisuje složitější detaily implementace. V kapitole 7 jsou nastíněny možnosti dalšího rozšíření práce a uvozují tak kapitolu 8, která shrnuje celou práci.

Přílohy k práci obsahují příklad formátu vstupních dat, příklad formátu morfologického filtru pro extrakci kolokací, statistiky výkonnostních testů programu na výpočetním stroji a také přehled obsahu přiloženého CD.

2 Víceslovné výrazy

Již z úvodu je patrné, že pojem *víceslovný výraz*[1] má mnoho synonym, jako např. *ustálené slovní spojení* nebo *kolokace*. Přesné vymezení těchto pojmů není cílem této práce a bývá navíc často dost problematické [1, kap. 2.1]. V této kapitole uvedeme pouze, co se pod pojmem *víceslovný výraz* míní v tomto textu, jaké další pojmy jsou použity a v jakých souvislostech. Dále také popíšeme některé vlastnosti víceslovných výrazů důležité pro implementaci programu.

2.1 Definice víceslovných výrazů a souvisejících pojmů

Na tomto místě si uvedeme přehled důležitých definic:

Víceslovný výraz (*multiword expression*) — víceslovný, významově nedělitelný celek, jehož rozbitím by se význam původního výrazu ztratil.

Kolokace (*collocation*) — jiný termín pro označení „víceslovný výraz". Je v rámci práce často používán jednoduše proto, že je kratší.

N-gram — označení pro obecnou skupinu N slov, většinou n-prvkovou podposloupnost delší posloupnosti (textu). Pro konkrétní hodnoty N = 1, 2 a 3 se používají názvy **unigram**, **bigram** a **trigram**.

Kolokující N-gram — n-tice slov $w_1, w_2, \ldots, w_n$, jenž patří do *kolokačního kontextu*[2] stejného slova (nejčastěji je to nějaké slovo w_i z dané n-tice). Jinými slovy každou n-tici slov ze vstupního textu, jenž **potenciálně** tvoří kolokaci, označujeme jako kolokující N-gram. V další části textu se však hovoří pouze o kolokujících N-gramech, proto jsou zkráceně označovány jen jako N-gramy.

2.2 Charakteristické vlastnosti víceslovných výrazů

Víceslovné výrazy mají mnoho zajímavých vlastností [1, kap. 2.2]. Pro implementaci programu jsou důležité zejména tři vlastnosti: *vnitřní struktura*, *kolokační kontext* a *sémantičnost kolokací*.

1 Paradoxně i pojem „víceslovný výraz" je vlastně víceslovný výraz.
2 Pojem „kolokační kontext" bude definován v následující podkapitole.

Vnitřní struktura

Vnitřní struktura je definována syntaktickými závislostmi mezi jednotlivými členy kolokace, ačkoli ne všechny kolokace ji musí mít — zde záleží na definici kolokace, zda v sobě podmínku o syntaktické závislosti členů obsahuje. V důsledku ovšem platí, že k detekci kolokací můžeme využít výsledky syntaktické analýzy vstupních textů. Protože program slouží pro detekci kolokací, jenž mají vnitřní strukturu, kolokace bez vzájemné syntaktické závislosti členů neuvažujeme.

Sémantičnost kolokací

Hned v úvodu bylo zmíněno, že víceslovné výrazy mají svůj specifický význam (sémantiku). Z gramatického hlediska jsou proto víceslovné výrazy tvořeny zejména *autosémantickými slovními druhy*: podstatnými jmény, přídavnými jmény, slovesy, číslovkami a příslovci. Této vlastnosti se dá úspěšně využít pro návrh jednoduché heuristiky pro zlepšení výsledků metod pro automatickou detekci kolokací.

Kolokační kontext

Jak uvádí Pecina s Holubem [1, str. 8]: *„Kolokační kontext je kontext slova, ve kterém mohou ležet slova, která s ním tvoří kolokaci.“*

Protože je kolokační kontext **vždy omezený** (někdy velmi výrazně), jsou kolokace jevem lokálním, což má dvá podstatné důsledky:

1. Nemá smysl hledat kolokace napříč různými dokumenty.
2. Slova, jenž tvoří kolokaci, se v textu často vyskytují společně.

První důsledek se možná na první pohled nejeví jako důležitý, ale velmi usnadňuje zpracování vstupních dat.

Druhý důsledek lze využít pro návrh metod automatické detekce kolokací, jenž jsou obsahem následující kapitoly.

3 Automatická detekce víceslovných výrazů

Automatická detekce víceslovných výrazů je proces, který zahrnuje několik fází [1, kap. 3.2]: předzpracování vstupních textů, extrakci kolokujících N-gramů, aplikaci metod detekce kolokací, hodnocení kolokací a analýzu výsledků. Program implementuje dvě z nich a to: **extrakci kolokujících N-gramů** a **aplikaci metod detekce kolokací**.

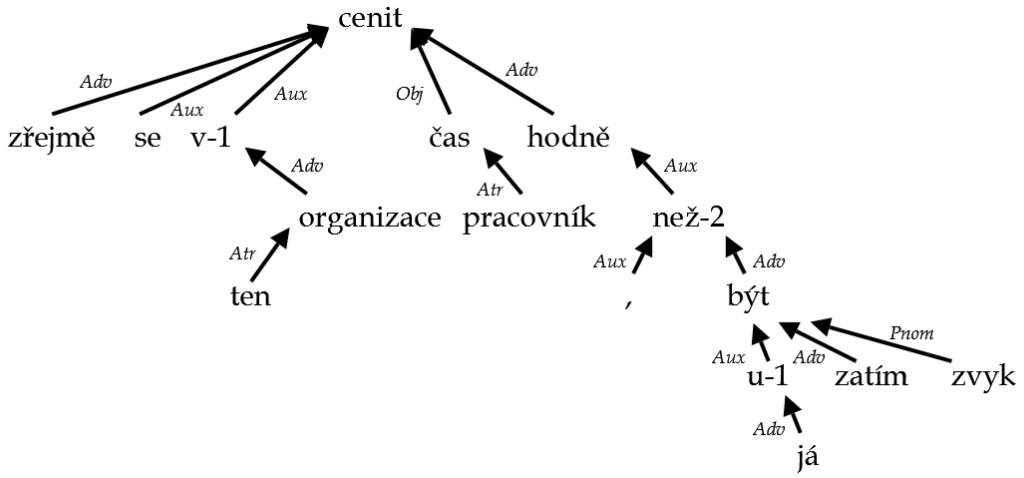
3.1 Metody extrakce kolokujících N-gramů

Z teoretického hlediska může kolokaci tvořit libovolná n-tice slov, z praktického hlediska je ovšem zřejmé, že kolokací nemůžou tvořit slova, jenž se nikdy spolu nevyskytují v nějakém omezeném kontextu. Tato myšlenka je základem pro návrh několika metod extrakce kolokujících N-gramů [1, kap. 4.4]. Zde bude blíže popsána jedna z nich (implementována v rámci programu) a to metoda využívající syntaktické závislosti.

3.1.1 Syntaktická závislost

Máme-li k dispozici textový korpus se syntaktickou anotací, můžeme se pokusit hledat kolokující N-gramy na základě syntaktické závislosti jednotlivých slov. Složky víceslovných výrazů jsou totiž ve většině případů syntakticky závislé (viz kap. 2.2).

V syntakticky anotovaném korpusu můžeme každou větu (tj. její členy a syntaktické závislosti mezi nimi) znázornit pomocí tzv. *závislostního stromu*. **Závislostní strom věty** je strom, jehož uzly tvoří základní tvary slov a hrany reprezentují syntaktickou závislost a jsou ohodnoceny typem této závislosti (viz obrázek 1). Z obrázku lze vypozorovat, že relace závislosti je asymetrická. Každý uzel (slovo) má pouze jednoho předka a tento předek se označuje jako **řídící slovo** závislosti. Řídícím slovem celé věty je pak kořen stromu (nejčastěji je to uzel odpovídající přísudku).



Obrázek 1: Příklad závislostního stromu věty: „Zřejmě si v té organizaci cenili času pracovníků více, než je u nás zatím zvykem."

Závislostní strom věty může obsahovat více stromů (grafů). Velmi často se tečka na konci věty definuje jako uzel bez řídícího slova a tvoří tak samostatný strom. V takovém případě koncová tečka tvoří pouze unigram a nestane se součástí žádného kolokujícího N-gramu.

Zcela jistě je v nejrůznějších situacích mnohem jednodušší pracovat pouze se souvislými stromy, proto existuje jiný (rozšiřující) pohled na závislostní stromy, který definuje tzv. „*abstraktní kořen*". Abstraktní kořen je rodičem všech „reálných" kořenů a není s ním spojeno žádné slovo věty.

Extrakce závislostních N-gramů

Přímočarý postup pro získání množiny všech kolokujících N-gramů z dané věty spočívá v extrakci všech **souvislých podstromů** velikosti N ze závislostního stromu věty. Takto získané N-gramy mají obdobné vlastnosti jako závislostní strom — jejich závislostní (pod)stromy jsou souvislé a právě jeden prvek je řídícím slovem celého N-gramu.

Tuto metodu je možné ještě „tranzitivně" rozšířit a to tím způsobem, že se extrahují i nesouvislé podstromy takové, že vzdálenosti všech jejich uzlů v rámci závislostního stromu věty nepřekračují určitou konstantu. Implementace programu ovšem pracuje pouze s přímou závislostí.

3.2 Metody detekce kolokací

Předchozí podkapitola popisovala způsoby, jakými lze získat kandidáty na kolokace. Cílem této kapitoly je popsat, jakými metodami tyto kandidáty rozdělit na ty více perspektivní a ty méně perspektivní (z hlediska jejich možnosti „býti kolokací“).

3.2.1 Počítaní frekvencí

Nejjednodušším způsobem pro určování kolokací, je počítání jejich výskytů v textu. Je zřejmé, že pokud se stejná skupina slov vyskytuje v textu opakovaně, je pravděpodobné, že má svůj vlastní význam, jenž se dá odvodit pouze z významu celku.

Nevýhodou této metody je, že nerozpozná kombinace často používaných, avšak sémanticky neúplných slov, která málokdy tvoří víceslovný výraz. Tento nedostatek se ovšem dá minimalizovat použitím tzv. *slovnědruhového filtru*, o němž se zmíníme v jedné z následujících kapitol.

3.2.2 Testování hypotéz

Vysoká frekvence výskytu nějakého N-gramu ještě nemusí znamenat, že jeho komponenty se spolu nevyskytují náhodně. U frekventovaných slov typu *nový, dobrý, každý apod.* je pravděpodobné, že se nezávisle v textu vícekrát vyskytují spolu s jiným slovem. Ne všechny často se vyskytující N-gramy jsou proto zajímavé pro hledání kolokací.

Bude nás proto zajímat, zda je společný výskyt dané N-tice slov náhodný nebo naopak zda komponenty N-gramu na sobě nějak závisí. Určování (ne)závislosti jevů je klasická statistická disciplína a obvykle se řeší tzv. *testováním hypotéz* [1, kap. 5.3].

Vstupní text pro účely testování hypotéz budeme brát jako náhodnou **posloupnost N-gramů**, přičemž se ovšem nejedná o posloupnost v striktním matematickém pojetí, protože na vzájemném pořadí N-gramů v textu nezáleží. Oproti tomu u **posloupnosti slov**, jenž N-gram tvoří, je pořadí významné: w_1w_2 a w_2w_1 jsou dva různé bigramy.

Jako **nulovou hypotézu** H_0 v našem případě označíme tvrzení, že mezi komponentami N-gramu není žádná souvislost a tedy jejich společný výskyt je zcela náhodný. **Alternativní hypotéza** H_1 bude negací nulové hypotézy.

Zkoumat budeme následující jevy:

- $w_1w_2…w_n$: výskyt kolokujícího N-gramu tvořeného n-ticí[3] slov $w_1w_2…w_n$. Zde můžeme měřit četnost výskytů $C(w_1w_2…w_n)$ a maximální věrohodný odhad pravděpodobnosti výskytu dané n-tice $w_1w_2…w_n$ (kde $C(*)$ je počet všech N-gramů v textu[4]):

$$P(w_1 w_2 ... w_n) = \frac{C(w_1 w_2 ... w_n)}{C(*)}$$

- w^i: výskyt slova w jako i-té komponenty N-gramu. Zajímat nás bude četnost $C(w^i)$ takových výskytů ve všech N-gramech v textu a maximální věrohodný odhad pravděpodobnosti, že náhodně vybrané slovo na i-té pozici N-gramu je w:

$$P(w^i) = \frac{C(w^i)}{C(*)}$$

Zajímat nás také budou i jevy doplňkové:

- $\neg w^i$: výskyt všech ostatních slov jako i-té komponenty kolokujícího N-gramu kromě slova w. Zde opět můžeme spočítat četnost výskytů $C(\neg w^i) = C(*) - C(w^i)$ a maximální věrohodný odhad pravděpodobnosti, že náhodně vybrané slovo na i-té pozici není w:

$$P(\neg w^i) = \frac{C(\neg w^i)}{C(*)} = \frac{C(*)-C(w^i)}{C(*)} = 1-\frac{C(w^i)}{C(*)} = 1-P(w^i)$$

- $(\neg w_i | w_j)^*$: výskyt n-tic takových, že pro některá $i \in \{1,…,n\}$ slovo w_i **není** členem n-tice, pro některá $j \in \{1,…,n\}$ slovo w_j **je** členem n-tice, $\|w_i\|+\|w_j\|=n$ a navíc pro všechna i a j platí $i \neq j$.

Nulovou hypotézu jsme definovali tak, že popisuje situaci, kdy zkoumaný N-gram netvoří kolokaci. Tzn. že společný výskyt jeho komponent je náhodný a

3 Pojem n-tice je zde použit ve smyslu **posloupnosti n prvků**.

4 V případě unigramů (resp. bigramů) je počet všech kolokujících N-gramů v textu roven (resp. přibližně roven) počtu všech slov v textu.

tedy odhad pravděpodobnosti společného výskytu slov $w_1 \ldots w_n$ je roven součinu odhadů pravděpodobnosti výskytů jednotlivých slov na daných pozicích N-gramu:

$$H_0: \; P(w_1 w_2 \ldots w_n) \; = \; P(w_1^1) P(w_2^2) \ldots P(w_n^n)$$

Nesmíme ovšem zapomenout, že takto definovaný model je **zjednodušený**, protože ve skutečnosti nelze říct, že text v přirozeném jazyce je posloupností náhodných slov.

Statistických testů existuje velké množství, jejich detailní popis může zájemce nalézt např. ve skriptech Jiřího Anděla [2]. Pro detekci kolokací se používají zejména: **t test**, **Z score**, **χ^2 test** a **poměr pravděpodobností**.

χ^2 test a **poměr pravděpodobností** při výpočtu statistik využívají tzv. *kontigenčních tabulek*. **Kontigenční tabulka** pro skupinu N jevů je N-rozměrná. Každý rozměr tabulky odpovídá jednomu jevu a obsahuje 2 sloupce (nebo chcete-li řádky): jeden pro případy, kdy jev **nastává** a druhý pro případy, kdy **nenastává**. Tabulka pak vzniká vzájemnou kombinací sloupců všech dimenzí. Pro N = 2 se dá zobrazit jako tzv. *čtyřpolní tabulka* (viz tabulka 1). V našem případě jevy odpovídají výskytům slov, takže v čtyřpolní tabulce sloupce odpovídají případům w_1^1 a $\neg w_1^1$ a řádky případům w_2^2 a $\neg w_2^2$. Všimněme si, že součet hodnot ze všech polí uvedené kontigenční tabulky je roven počtu všech bigramů (uvedený fakt platí pro libovolné N).

$\|\|w_1 w_2\|\|$	$\|\|\neg w_1 w_2\|\|$
$\|\|w_1 \neg w_2\|\|$	$\|\|\neg w_1 \neg w_2\|\|$

Tabulka 1: Kontigenční tabulka pro N=2.

Nyní stručně popíšeme výhody a nevýhody některých uvedených testů a také uvedeme vzorce pro výpočet testových statistik na základě **frekvencí pozorovaných jevů** [3]. Vzorce používají následující symboly:

- N — **počet všech N-gramů** (jednodušší zápis pro *C(*)*).
- f — **zjištěná frekvence** výskytů N-gramu. Zde rozlišujeme dva případy: *f(xy)* je běžná frekvence (bigramu), zatímco f_{ij} je hodnota z kontigenční tabulky (indexy jsou pouze pro zdůraznění, že se v daném případě jedná o frekvence z kontigenční tabulky, jejich přesná interpretace není důležitá).

- $\hat{f}$ – **očekávaná frekvence** výskytů N-gramu. Podobně jako výše rozlišujeme dva případy: $\hat{f}(xy)$ a $\hat{f}_{ij}$. Očekávaná frekvence N-gramu se spočítá na základě maximálních věrohodných pravděpodobností jeho komponent.

  Pro bigram w_1w_2 je to:

$$\hat{f}(w_1w_2) = P(w_1^1) * P(w_2^2) * N = \frac{C(w_1^1)}{N} * \frac{C(w_2^2)}{N} * N$$

  Obdobně, pro pole třírozměrné kontigenční tabulky odpovídající „trigramu“ $\neg w_1 w_2 \neg w_3$ je:

$$\hat{f}_{ij}(\neg w_1 w_2 \neg w_3) = \frac{C(\neg w_1^1)}{N} * \frac{C(w_2^2)}{N} * \frac{C(\neg w_3^3)}{N} * N$$

t test

Poměrně často používaným testem pro detekci kolokací je **t test**. Má ovšem jeden důležitý nedostatek: počítá s tím, že četnosti výskytu slov mají přibližně normální rozdělení, což obecně není pravda [4].

$$\frac{f(xy) - \hat{f}(xy)}{\sqrt{f(xy)(1-(f(xy)/N))}}$$

Z score

Z score se používá jako alternativa k t testu.

$$\frac{f(xy) - \hat{f}(xy)}{\sqrt{\hat{f}(xy)(1-(\hat{f}(xy)/N))}}$$

χ^2 test

Výhodou **χ^2 testu** je, že nevyžaduje normálnost rozdělení četnosti výskytů jako t test. Jak již bylo zmíněno, pro jeho výpočet je třeba použít kontigenční tabulky, vzorec je následující:

$$\sum_{ij} \frac{(f_{ij} - \hat{f}_{ij})^2}{\hat{f}_{ij}}$$

Poměr pravděpodobností

Další metodou založenou na testování hypotéz je **poměr pravděpodobností**. Jeho velkou výhodou je, že výsledek se dá velmi hezky interpretovat: říká nám, o kolik pravděpodobnější je hypotéza, že N-gram je kolokací než hypotéza, že kolokací není [5, kap. 5.3.4]. V případě bigramu w_1w_2 se hypotézy definují následovně:

- **Hypotéza 1**: $P(w_2^2|w_1^1) = p = P(w_2^2|\neg w_1^1)$
- **Hypotéza 2**: $P(w_2^2|w_1^1) = p_1 \neq p_2\ P(w_2^2|\neg w_1^1)$

Hypotéza 1 je formálním vyjádřením nezávisloslosti výskytu slov w_1 a w_2. Hypotéza 2 naopak popisuje případ, kdy vzájemný výskyt těchto slov není náhodný a tedy tato slova můžou tvořit kolokaci (přitom předpokládáme, že $p_1 >> p_2$). Pokud symbolem $L(H_1)$ (resp. $L(H_2)$) označíme pravděpodobnost platnosti hypotézy 1 (resp. hypotézy 2), pak poměr pravděpodobností λ se jednoduše vyjádří jako:

$$\lambda = \frac{L(H_1)}{L(H_2)}$$

Tuto metodu lze použít i pro testování hypotéz. Pokud je λ poměr pravděpodobností pro určitý N-gram, pak hodnota $-2\log\lambda$ má přibližně χ^2 rozdělení [6]. Navíc je výsledek v případě malých frekvencí slov spolehlivější než χ^2 test.

S využitím zjištěných a očekávaných hodnot z kontigenční tabulky lze hodnotu $-2\log\lambda$ spočítat podle následujícího vzorce [3]:

$$2\sum_{ij}(f_{ij}\log\frac{f_{ij}}{\hat{f}_{ij}})$$

3.2.3 Vzájemná informace

Jinou metodou detekce kolokací je metoda, jenž vychází z teorie informace, měření tzv. *vzájemné informace* (*mutual information*). Přesněji řečeno se metodě, která nás zajímá, říká *bodová vzájemná informace* (*pointwise mutual information*), protože původní obecnější název se dnes používá pro trochu jinak definovanou metodu [5, kap. 5.4].

Co tedy vyjadřuje (bodová) vzájemná informace mezi dvěma konkrétními událostmi x a y? Definice je následující:

$$I(x,y) = \log_2 \frac{P(xy)}{P(x)P(y)} = \log_2 \frac{P(x|y)}{P(x)P(y)} = \log_2 \frac{P(y|x)}{P(x)P(y)}$$

Událostmi x a y jsou v případě detekce kolokací výskyty dvou konkrétních slov a vzájemná informace nám říká (zjednodušeně), kolik informace nám výskyt jednoho slova říká o výskytu slova druhého.

Měření vzájemné informace je poměrně spolehlivá metoda pro určování nezávislosti, ale pro určování závislosti je její použití problematické. V případě **dokonalé závislosti** totiž platí:

$$I(x,y) = \log_2 \frac{P(xy)}{P(x)P(y)} = \log_2 \frac{P(x)}{P(x)P(y)} = \log_2 \frac{1}{P(y)}$$

Proto pro dokonale závislé bigramy platí, že čím je jejich frekvence výskytů menší, tím je jejich vzájemná informace větší a tedy výsledek měření je závislý na počtu výskytů daného bigramu, což je nežádoucí. Můžeme tento jev trochu potlačit tím, že bigramy s nízkou frekvencí automaticky vyřadíme, ale tímto se zbavíme pouze nejvíce zkreslených výsledků, nedostatek metody tímto nijak neopravíme.

V případě **dokonalé nezávislosti** platí:

$$I(x,y) = \log_2 \frac{P(xy)}{P(x)P(y)} = \log_2 \frac{P(x)P(y)}{P(x)P(y)} = \log_2 1 = 0$$

Pro hodnoty vzájemné informace kolem 0 se tedy můžeme s velkou pravděpodobností spolehnout na to, že komponenty hodnoceného bigramu jsou na sobě nezávislé.

3.2.4 Filtrace N-gramů

V kapitole 2 bylo zmíněno, že kolokace tvoří především slova, která patří mezi autosemantické slovní druhy[5]. Této vlastnosti se využívá pro návrh speciální heuristiky, tzv. *slovnědruhového filtru* [1, kap. 5.1].

Slovnědruhový filtr tvoří seznam pravidel určující kombinace slovních druhů, které jsou pro víceslovné výrazy typické. Kolokující N-gramy, které neodpovídají

5 Pro připomenutí — jsou to: podstatná a přídavná jména, slovesa, číslovky a příslovce.

žádnému z pravidel, jsou vyřazeny ze zpracování. K filtraci výsledků na základě slovnědruhového filtru je samozřejmě nutné znát slovní druhy slov ve zpracovávánem textu, což ovšem bývá základní informace u anotovaných korpusů.

Příklady vzorů pro slovnědruhový filtr a jim odpovídajících kolokací jsou v tabulce 2 (A je zkratka pro přídavné jméno, N pro podstatné jméno a P pro předložku). Tento filtr navrhli Justenson a Katz [7] a je určen pro anglický jazyk. Pro češtinu je až příliš restriktivní, proto pro české texty navrhli Pecina s Holubem rozšířený filtr, jenž navíc obsahuje i slovesa (V), číslovky (C) a příslovce (D) (viz příloha B). Navíc lze pro češtinu, jako morfologicky bohatý jazyk, definovat ještě specifičtější pravidla (vzory) zahrnující i slovní poddruhy.

Slovnědruhový filtr je velice universální heuristika — dá se použít ke zlepšení výsledků všech metod zmíněných v této kapitole.

vzor	**příklad**
AN	kanadské bodování
NN	král střelců
AAN	regulérní hrací doba
ANN	útočné pásmo hostů
NAN	vítěz základní části
NNN	titul mistra světa
NPN	rolba na led

Tabulka 2: Vzory pro slovnědruhový filtr navržené Justensonem a Katzem a příklady víceslovných výrazů, které jim vyhoví.

4 Popis programu

Jak již bylo zmíněno v předchozí kapitole, program implementuje dvě fáze automatické detekce kolokací: **extrakci kolokujících N-gramů** a **aplikaci metod detekce kolokací**. Kolokující N-gramy jsou extrahovány na základě vzájemné syntaktické závislosti složek N-gramu, aplikace metod detekce kolokací pak spočívá ve výpočtu testových statistik. Ty ovšem poskytují „pouze" relativní ohodnocení (neurčují kolokace se stoprocentní přesností).

4.1 Vstupní data

Program je navržen speciálně pro detekci kolokací v **Pražském závislostním korpusu** (neboli PDT[6]) ve verzi 2.0. PDT 2.0 je textový korpus se syntaktickou, sémantickou a morfologickou anotací obsahující velké množství český textů [8].

4.1.1 Formát vstupních dat

PDT data jsou uložena v textových souborech **v řádkovém formátu**. Každý řádek obsahuje záznam pro jedno slovo. Věta je uložena jako posloupnost řádků, jednotlivé věty jsou odděleny prázdným řádkem.

Formát řádku (záznam o slově) je následující:

```
order occurrence lemma tag parent dependency
```

Jednotlivé položky jsou oddělené **tabulátorem** a jejich význam je následující:

- **order** je pořadové číslo slova ve větě. Program detekuje špatně očíslované věty a automaticky je vyřazuje ze zpracování.
- **occurrence** je tvar slova, v jakém se dané slovo vyskytuje v povrchovém slovosledu věty.
- **lemma** je základní tvar slova.
- **tag** je speciální, 15-ti znaková morfologická značka určující morfologické kategorie slova. Každá pozice značky odpovídá nějaké morfologické kategorii, pořadí kategorií a významy znaků na jednotlivých pozicích jsou uvedeny v dokumentaci k PDT. Zde se hodí uvést jenom tolik, že když pro nějaké slovo

6 Prague Dependency Treebank 2.0 (PDT 2.0) – viz: http://ufal.mff.cuni.cz/pdt2.0/

nemá daná morfologická kategorie význam, obsahuje relevantní pozice pomlčku. Příklady několika morfologických značek jsou v tabulce 3.

- **parent** je pořadové číslo řídícího slova (rodiče v závislostním stromě). V případě řídícího slova celé věty (kořene závislostního stromu) má tato položka hodnotu nula.
- **dependency** je informace o typu syntaktické závislosti. Přehled všech různých typů syntaktické závislosti je uveden v dokumentaci k PDT. Pro implementaci programu je důležitá informace, že všech různých typů je cca 130 a tedy lze pro jejich reprezentaci použít číselný index o velikosti jednoho bajtu.

Morf. značka	Význam
N-FS6----------	N – podst. jm., F – ženský rod, S – jednotné číslo, 6 – 6. pád
Cr--4----------	C – číslovka, r – řadová číslovka, 4 – 4. pád
V-Y----1P------	V – sloveso, Y – rod mužský, 1 – 1. osoba, P – přítomný čas

Tabulka 3: Příklady jednoduchých morfologických značek a jejich význam.

Krátký příklad obsahu datového souboru PDT 2.0 je uveden v příloze A.

4.1.2 Zpracování vstupních dat

Textové korpusy jsou velmi obsáhlé a zřídka jsou uloženy v jediném souboru, proto je program navržen tak, aby umožňoval **sériové zpracování** vstupních souborů.

Na výsledky extrakce N-gramů mají vliv pouze hranice vět, ne hranice souborů (resp. dokumentů). Hranice dokumentu je ovšem důležitá v případě zjišťování kontextu (viz podkapitola 4.6). Implementace programu počítá s tím, že jeden vstupní datový soubor odpovídá vždy jednomu dokumentu.

4.1.3 Kódování vstupních dat

Program klade minimální nároky na kódování vstupních dat. Jednotlivé datové položky jsou z pohledu programu jenom posloupností bitů a jedinými znaky interpretovanými na základě jejich (bitové) reprezentace ve zdrojovém kódu jsou **tabulátor**, **konec řádku**, **hvězdička** (*) a **pomlčka** (-). Zdrojové kódy jsou ve formátu ASCII, proto program bude fungovat s daty v jakémkoli kódování, jenž je

nadmnožinou ASCII[7]. Pro bezchybnou extrakci je však nutné, aby všechny vstupní datové soubory byly ve stejném kódování.

4.2 Ekvivalence N-gramů

Chceme-li počítat výskyty identických objektů v rámci nějaké skupiny objektů potřebujeme pro libovolné dva objekty dokázat rozhodnout, zda jsou si identické. Ne jinak je tomu v případě počítání frekvencí jednotlivých N-gramů v textu.

Identita N-gramu závisí na jeho složkách, proto si uveďme, jaké položky obsahuje **každá** složka. Jsou nimi:

- lemma
- pořadové číslo řídícího slova
- typ závislosti
- morfologická značka

Vidíme, že povrchový tvar slova je pro detekci kolokaci nezajímavý. Lemma se ukládá beze změny, tak jak je uvedeno ve vstupních datech. Pořadové číslo řídícího slova se pro každou složku pozmění tak, aby korespondovalo s pořadím slov v N-gramu. Typ závislosti zůstává stejný pro každé slovo, kromě řídícího slova celého N-gramu — tomu je přiřazen speciální typ závislosti „*Head*“. Morfologická značka je podmnožinou morfologické značky uvedené ve vstupních datech. Signifikantní pozice pro daný běh programu určuje jeden z parametrů, pro každou značku jsou ovšem stejné.

Jak jsme již zmínili dříve, pořadí složek N-gramu je signifikantní.

Definice 4.1 (Ekvivalence N-gramů)

> Dva N-gramy jsou považovány za **identické**, právě když je počet jejich složek stejný a odpovídající si složky obou N-gramů mají identické **všechny** svoje položky.

Je jistě zřejmé, že nemá smysl porovnávat bigram a trigram. Proto je hodnota N jedním ze základních parametrů programu a v rámci jednoho běhu program dokáže extrahovat pouze N-gramy pro dané N.

7 Např. UTF-8 nebo ISO-8859-2 (kódování používaná v rámci PDT).

Dále je dobré si uvědomit, že uvedená definice ekvivalence N-gramů zajišťuje, že tvary závislostních stromů identických N-gramů se musí shodovat (včetně ohodnocení jednotlivých hran).

4.3 Filtrace extrahovaných N-gramů

V kapitole 3 byla popsána jednoduchá heuristika pro zlepšení automatické detekce kolokací, tzv. *slovnědruhový filtr*. Program umí filtrovat nalezené N-gramy na velmi podobném principu, v zásadě se jedná o zobecnění principu slovnědruhové filtrace, tzv. *morfologický filtr*.

Morfologický filtr filtruje N-gramy na základě údajů obsažených v morfologických značkách slov, jenž N-gram tvoří. Jedním z těchto údajů je také slovní druh, ale filtrovat lze podle libovolné kategorie.

Morfologický filtr je tvořen sadou pravidel, jenž jsou postupně aplikovaná na každý nalezený N-gram. Aplikace pravidel se zastaví, když je nalezeno vyhovující pravidlo nebo už nezbývají žádná další pravidla. Pokud nebylo nalezeno vyhovující pravidlo, je N-gram vyřazen z dalšího zpracování (jeho výskyt není započítán). Filtr tedy funguje na principu *whitelistu*.

Pravidla se načítají z textového souboru — každé pravidlo se uvádí na samostatném řádku a skládá se z N částí, jenž se oddělují jednou nebo více mezerami. Jednotlivé části odpovídají jednotlivým složkám N-gramu. Formát každé části je stejný jako formát morfologické značky — na každé pozici lze tedy uvést znak v jednom z následujících významů:

a) Definovat (jedinou!) vyhovující hodnotu pro danou morfologickou kategorii (pomocí znaku, jenž tuto hodnotu indikuje).

b) Použít zástupný znak „-“ (pomlčka), jenž vyhoví jakémukoliv znaku morfologické značky.

Je důležité si ještě uvědomit, že znak pomlčka v rámci morfologické značky nevyhoví libovolnému znaku v rámci morfologického filtru, pouze pomlčce!

Tabulka 4 obsahuje příklady pravidel a jejich aplikace na N-gramy pro lepší představu o fungování filtru.

N	Pravidlo	Morfologické značky složek validního N-gramu	Morfologické značky složek nevalidního N-gramu
1	---4C	N--4	N---C
2	C -	CD-- V---	AD-- V---
2	C--D- N---1	C--D2 N--D1	C--- A---1
3	A---3 N---- N---2	AF-N3--- N------I N--D2--K	AN---K N------ A---2---

Tabulka 4: Příklady pravidel pro morfologický filtr a jejich aplikace na N-gramy. Podtržené znaky označují pozice, které jsou signifikantní pro určení (ne)validity N-gramu. Pozn.: Morfologické značky zde uvedené jsou pouze pro ilustraci.

Jak je vidět z tabulky, části pravidel nemusí mít stejnou délku jako morfologické značky:

1. V případě, že je část pravidla **kratší**, jsou chybějící pozice interpretovány jako pomlčky, takže vyhoví všem znakům.
2. V případě, že je část pravidla **delší**, jsou přebytečné pozice ignorovány. Program umožňuje pracovat pouze s určitou podmnožinou morfologické značky, proto tento případ může nastat i v případě, že všechny části pravidel jsou kratší než 15 znaků.

Navíc části pravidel nemusí být ani stejně dlouhé v rámci jednoho pravidla. Vždy ale musí být uvedeny všechny části (třeba jen jako samostatná pomlčka), protože podle počtu částí program detekuje pro jaké N je pravidlo určeno. Je dovoleno mít ve stejném souboru pravidla pro různá N — program vždy načte pouze ty relevantní.

Použití morfologického filtru má i další výhodu. Protože je filtrace prováděna už během extrakce N-gramů, vyřazené N-gramy ani jejich frekvence se neukládají a díky tomu **se šetří** použitá paměť. Poměr ušetřené paměti není ovšem snadné přesně odhadnout, závisí totiž na vstupních datech, na pravidlech filtru i na hodnotě parametru N.

Program počítá statistiky aplikace jednotlivých pravidel v rámci morfologického filtru, tj. pro každé pravidlo si pamatuje počet N-gramů, které mu vyhověly. Tyto statistiky lze vypsat do zadaného souboru.

4.4 Výsledné statistiky

Kromě frekvence výskytu a očekávané frekvence výskytu program počítá následující testové statistiky:

- χ^2 test (chi square)
- poměr pravděpodobností (log likelihood ratio)
- bodová vzájemná informace (pointwise mutual information)
- Pearsonův korelační koeficient (Pearson's coefficient)
- t test (Student's t-test)
- Z score

χ^2 test a poměr pravděpodobnosti lze spočítat pro obecné N, zbytek testových statistik se dá vypočítat pouze pro bigramy (N = 2).

Statistiky se ukládají do výstupního souboru specifikovaného parametrem programu. Záznam pro jednotlivý N-gram obsahuje 3 části: definici N-gramu, příslušnou kontigenční tabulku a testové statistiky.

Definice N-gramu má N řádků, na každém řádku jsou data jedné složky N-gramu v následujícím pořadí:

1. index (pozice) dané složky v rámci N-gramu
2. lemma
3. morfologická značka (může být i prázdná)
4. index (pozice) řídícího slova (složky)
5. typ závislosti

Po definici N-gramu následuje řádek s údaji z kontigenční tabulky. Pořadí polí je dáno jejich typem[8] (pole jsou řazena vzestupně).

Jako poslední následuje řádek se všemi testovými statistikami a to ve stejném pořadí, v jakém jsou uvedeny v seznamu výše.

Hodnoty na řádcích jsou odděleny **tabulátorem**, záznamy pro jednotlivé N-gramy jsou odděleny jedním prázdným řádkem.

8 Definici tzv. smíšených N-gramů, jenž odpovídají polím kontigenční tabulky a také definici jejich typu obsahuje kapitola 5.

4.5 Filtrace vypisovaných výsledků

Další možností, jak omezit výstup programu, je filtrace vypisovaných výsledků. Na rozdíl od morfologického filtru, se tato filtrace provádí až po extrakci N-gramů a jediným jejím efektem je tak zmenšení objemu výstupních dat. Pro vhodně nastavené kritické hodnoty to ovšem může znamenat výrazné zvětšení kvality výstupu.

Filtrace vypisovaných výsledků se dělí na dva stupně, které se dají označit jako *pre-evaluační* a *post-evaluační*.

Pre-evaluační filtrace

K pre-evaluační filtraci dochází ještě před výpočtem testových statistik. Testové statistiky se vypočítávají na základě skutečné a očekávané četnosti výskytů. Pro tyto hodnoty lze definovat požadované minimum. Pokud skutečná nebo očekávaná četnost N-gramu je menší než příslušné minimum, je N-gram vyřazen z dalšího zpracování.

Minimum pro skutečnou četnost se uvádí jako **přirozené** číslo, minimum pro očekávanou četnost jako kladné **reálné** číslo.

Post-evaluační filtrace

K post-evaluační filtraci dochází až po výpočtu testových statistik. Lze definovat kritické hodnoty (minimum) pro všechny počítané statistiky. Oproti pre-evaluační filtraci lze na tomto stupni navíc i definovat, zda pro překlenutí filtru musí být dosažena všechna daná minima nebo postačí, pokud je dosaženo libovolné z nich. Pokud nejsou splněny podmínky filtru, N-gram není zařazen mezi výsledky (na výstup).

Kritické hodnoty pro všechny testové statistiky se uvádí jako kladná **reálná** čísla.

4.6 Zjišťování kontextu

Význam některých slov a slovních spojení se mění v závislosti na jejich kontextu, proto znalost kontextu, ve kterém se daný N-gram nejčastěji vyskytuje, může mít pozitivní vliv na přesnost rozhodnutí, zda je či není kolokací.

Program nabízí „zjišťování kontextu“ jako volitelnou funkci. Pokud je tato funkce aktivní, pak se během extrakce pro každý N-gram ukládá také jeho kontext,

jenž tvoří slova vyskytující se ve specifikovaném **povrchovém** okolí N-gramu[9]. Přesněji řečeno každé **slovo v kontextu** identifikuje dvojice údajů: jeho lemma a morfologická značka.

Je nutno poznamenat, že zjišťování kontextu mnohonásobně zvýšuje paměťové nároky programu! Zatímco běžná detekce vyžaduje pro každý N-gram uložení jeho N složek a jednoho čísla (četnosti výskytu), v případě zjišťování kontextu se navíc jedná o **seznam** slov spolu s četnostmi výskytů[10].

Pro každý N-gram se rozlišuji dva **typy kontextu**:

1. *Úzký kontext* — patří do něj slovo, které se v povrchovém slovosledu věty vyskytuje před prvním slovem N-gramu a slovo, které se vyskytuje za posledním slovem N-gramu.
2. *Široký kontext* — patří do něj všechny slova ve specifikovaném okolí N-gramu (to se označuje jako tzv. *kontextové okénko*).

Kontextové okénko

Kontextové okénko specifikuje okolí, ve kterém leží slova patřící do širokého kontextu N-gramu. Má svůj střed a svoji velikost (poloměr) — co je středem a co přesně určuje velikost okénka, to závisí na jeho typu:

1. *Větné kontextové okénko* má svoji velikost (poloměr) vyjádřenou v jednotkách vět a jeho středem je věta, ve které se nachází zkoumaný N-gram. Velikost 0 zde znamená, že kontextové okénko tvoří pouze tato věta, velikost 1 označuje kontextové okénko o rozsahu 3 vět („aktuální" věta a věty sousední) atd. pro další velikosti.
2. *Slovní kontextové okénko* má svoji velikost (poloměr) vyjádřenou v jednotkách slov a jeho středem je řídící slovo právě zkoumaného N-gramu. Velikost 0 proto v tomto případě evidentně postrádá smysl, velikost 1 obdobně jako výše definuje kontextové okénko, jenž tvoří řídící slovo (střed) a sousední slova (atd. pro další velikosti).

Do kontextu se nezahrnují slova, jenž zkoumaný N-gram tvoří (ovšem pokud se ve specifikovaném okolí vyskytují stejná slova, uložená jsou).

9 Tj. kontext nezávisí na tvaru závislostního stromu věty, ale jejím povrchovém slovosledu.

10 V kapitole zabývající se analýzou implementace programu zjistíme, že pro výpočet testových statistik **jednoho** N-gramu potřebujeme více údajů než jen jeho četnost výskytů. Na uvedeném srovnání se ovšem nic nezmění, protože v případě zjišťování kontextu se příslušné seznamy slov ukládají i pro tyto „údaje navíc" (jedná se o tzv. hvězdičkové N-gramy — viz kapitola 5).

Pro slova v **širokém** kontextu lze specifikovat (morfologický) *kontextový filtr*, jenž funguje naprosto stejně, jako morfologický filtr pro N-gramy. Pravidla se aplikují na jednotlivá slova, proto musí obsahovat pouze jednu část. Slova, která nevyhoví filtru, se jednoduše ignorují. Slova patřící do **úzkého** kontextu se ukládají vždy.

Formát výstupního souboru

Výsledky (kontexty) se ukládají do výstupního souboru specifikovaného parametrem programu. Záznam pro každý N-gram obsahuje 3 části: definici N-gramu, jeho úzký kontext a široký kontext.

Definice N-gramu je uvedena ve stejném formátu jako v případě souboru, který obsahuje výstup s testovými statistikami (viz podkapitola 4.4). Kontext je ovšem zjišťován i pro tzv. *hvězdičkové N-gramy*, které podrobně definujeme až v následující kapitole (zjednodušeně jsou to N-gramy, kterým „chybí“ některé složky — pak je místo dané složky na příslušném řádku uveden pouze znak hvězdičky „*“).

Druhá (resp. třetí) část obsahuje seznam slov v úzkém (resp. širokém) kontextu. Údaje pro každé slovo jsou uvedeny na samostatném řádku jako trojice: lemma, morfologická značka (ta může být prázdná) a příslušná četnost.

Hodnoty na řádku jsou odděleny tabulátorem, jednotlivé části záznamu (definice, úzký kontext a široký kontext) jsou odděleny jedním prázdným řádkem, záznamy pro jednotlivé N-gramy jsou odděleny dvěma prázdnými řádky.

4.7 Další funkce

Statistiky datových souborů

Pro každý datový soubor se počítají všechna slova, věty a extrahované N-gramy (pokud je aktivní morfologický filtr, tak se zvlášť počítají i N-gramy, které vyhověly filtru). Navíc se tyto údaje spočítají i celkově (pro všechny datové soubory celkem).

Statistiky uložených dat (objektů)

Lemmata, morfologické značky a N-gramy se do paměti ukládají unikátně, takže jejich statistiky uvádějí unikátní výskyty (na rozdíl od statistik datových souborů). U N-gramů se navíc uvádí počet unikátních N-gramů každého typu (*typ N-gramu* definujeme v následující kapitole).

4.8 Ovládání a parametry programu

Popsali jsme si vlastnosti a funkce programu, nyní je vhodné uvést, jakým způsobem se program ovládá.

Program pracuje v textovém režimu a jeho běh se dá ovlivnit pouze pomocí parametrů. Parametry se uvádějí s úvodní pomlčkou a můžou být uvedeny v libovolném pořadí.

Přehled všech parametrů spolu se stručným popisem je (také) obsahem nápovědy k programu — ta se zobrazí, pokud je program spuštěn s chybnými parametry nebo bez parametrů. Nápověda k programu navíc obsahuje informace o minimálních a maximálních povolených hodnotách některých číselných parametrů[11].

Bitová maska

Některé parametry vyžadují jako svou hodnotu bitovou masku. Znaky určující signifikantní a nesignifikantní pozice lze zjistit z nápovědy (v současné verzi programu je to *hvězdička* pro určení signifikantní a *pomlčka* pro určení nesignifikantní pozice).

4.8.1 Povinné parametry programu

-i „cesta k souboru“

Cesta k souboru se seznamem (vstupních) datových souborů. Pokud jsou jména datových souborů relativní vzhledem k jinému adresáři, než je pracovní adresář programu, lze cestu k tomuto adresáři úvest v parametru -d (viz níže).

-n „přirozené číslo“

Parametr určuje velikost extrahovaných N-gramů. Minimální povolená hodnota je 2, maximální povolenou hodnotu lze zjistit v nápovědě k programu.

-o „cesta k souboru“

Cesta k výstupnímu souboru.

11 Rozsah pro povolené hodnoty některých parametrů je definován pomocí konfiguračních konstant ve zdrojovém kódu a závisí tak na hodnotě, jakou mají tyto konstanty v době kompilace. Většina omezení je ale motivována praktickou použitelností programu.

4.8.2 Volitelné parametry programu

-c „cesta k souboru“

Cesta k souboru pro uložení výsledků „zjišťování kontextu“ (uvedení parametru zapne zjišťování kontextu). Pokud není kontextové okénko specifikováno parametrem, pak se jako výchozí použije větné kontextové okénko o poloměru 0.

-cf „cesta k souboru“

Cesta k souboru s morfologickým filtrem pro slova v kontextu.

-cs „celé číslo“

Velikost (poloměr) kontextového okénka **ve větách**. Povolené hodnoty lze zjistit v nápovědě k programu.

-cw „celé číslo“

Velikost (poloměr) kontextového okénka **ve slovech**. Povolené hodnoty lze zjistit v nápovědě k programu.

-d „cesta k adresáři“

Pomocí tohoto parametru lze určit, který adresář bude výchozí pro relativní cesty k datovým souborům (uvedené v souboru specifikovaném parametrem -i).

-f „cesta k souboru“

Cesta k souboru s morfologickým filtrem (uvedení parametru zapne filtraci extrahovaných N-gramů — viz příslušná kapitola).

-fs „cesta k souboru“

Cesta k souboru pro uložení statistik morfologického filtru.

-m „bitová maska“

Maska určující signifikantní a nesignifikantní pozice morfologické značky slov v korpusu. Pokud není uvedena, morfologická značka se pro všechny slova ignoruje!

Příklad: Pokud bude zadána maska „*--**“, program bude pracovat pouze s první, čtvrtou a pátou pozicí morfologické značky.

-p „přirozené číslo“

Přesnost, se kterou se vypíší výsledky testových statistik.

-s „cesta k souboru“

Cesta k souboru pro uložení statistik datových souborů a celkových statistik.

-ta

Uvedení parametru způsobí, že v rámci filtrace výsledků bude nutné dosažení všech uvedených minim (viz kapitola o filtraci výsledků).

-tc „kladné reálné číslo“

Minimum pro hodnotu χ^2 testu (uvedení parametru zapne filtraci výsledků — viz příslušná kapitola).

-te „kladné reálné číslo“

Minimální hodnota pro očekávanou četnost (uvedení parametru zapne filtraci výsledků — viz příslušná kapitola).

-tf „přirozené číslo“

Minimální hodnota pro skutečnou četnost (uvedení parametru zapne filtraci výsledků — viz příslušná kapitola).

-tl „kladné reálné číslo“

Minimum pro hodnotu poměru pravděpodobnosti (uvedení parametru zapne filtraci výsledků — viz příslušná kapitola).

-tm „kladné reálné číslo“

Minimum pro hodnotu vzájemné informace (uvedení parametru zapne filtraci výsledků — viz příslušná kapitola).

-to

Uvedení parametru způsobí, že v rámci filtrace výsledků bude postačovat dosažení jednoho z uvedených minim (viz kapitola o filtraci výsledků).

-tp „kladné reálné číslo“

Minimum pro hodnotu Pearsonova koeficientu (uvedení parametru zapne filtraci výsledků — viz příslušná kapitola).

-tt „kladné reálné číslo“

Minimum pro hodnotu t-testu (uvedení parametru zapne filtraci výsledků — viz příslušná kapitola).

-tz „kladné reálné číslo“

Minimum pro hodnotu Z score (uvedení parametru zapne filtraci výsledků — viz příslušná kapitola).

4.9 Informační a chybová hlášení

Program pro svou práci nevyžaduje interaktivitu s uživatelem, proto nemá žádné grafické rozhraní. Typický běh programu však trvá mnohem déle než jen pár sekund, proto program průběžně vypisuje informace o dosavadním průběhu a v případě složitějších operací jejich průběh znázorňuje pomocí jednoduchého ukazatele. Všechny informace (resp. chybové zprávy) jsou vypisovány na standardní (resp. chybový) výstup.

Bezchybný běh programu závisí na korektním vstupu (včetně parametrů) a dostatku volné operační paměti. Na případné chybové stavy program reaguje jedním ze dvou způsobů (trochu po vzoru kompilačních nástrojů): varováním nebo chybou.

4.9.1 Varování

Varování jsou reakcí programu na menší chyby, které nezpůsobí selhání (pád) programu, nicméně ale můžou ovlivnit výstup programu. Pokud se vyskytnou, je vypsáno (na standardní chybový výstup) hlášení o chybě a o způsobu, jakým ji program ošetřil a běh programu dále pokračuje.

Varování způsobí mj. následující situace:

1. Kombinace vstupních parametrů je logicky chybná — např. je definována cesta k souboru s morfologickým filtrem (zapnutá filtrace), ale není uvedena maska pro morfologickou značku (morfologické informace se neextrahují a filtrace je tedy neúčinná).
2. Špatné očíslování slov nebo špatný formát řádku ve vstupním souboru (příslušná věta je ignorována).

3. Délka věty překročila maximální povolenou hodnotu (příslušná věta je ignorována).
4. Nelze otevřít datový vstupní soubor během zpracování (příslušný soubor je ignorován).

4.9.2 Chyby

Některé chybové situace jsou natolik závažné, že neumožňují další běh programu nebo by jejich případné ignorování velmi výrazně změnilo výstup programu. Pokud se takový stav vyskytne, je vypsáno patřičné hlášení (na standardní chybový výstup) a běh programu je **okamžitě ukončen**.

Chybu způsobí mj. následující situace:

1. Chybějící povinný parametr programu nebo neplatná hodnota parametru programu.
2. Prázdný seznam datových souborů.
3. Nelze otevřít datový vstupní soubor při startu programu.
4. Nelze otevřít soubor s morfologickým filtrem nebo kontextovým filtrem.
5. Některý ze specifikovaných výstupních souborů (pro výstup programu, pro kontext, pro statistiky) nelze otevřít pro zápis.
6. Došlo k vyčerpání dostupné operační paměti.
7. Došlo k přetečení indexů použitých pro uložení řetězců (podrobněji viz kapitola 6).

Všechny chyby až na poslední dvě jsou z uživatelského hlediska „neškodné“, protože se můžou vyskytnout jen ve fázi inicializace. Poslední dvě chyby se ovšem můžou vyskytnout v průběhu extrakce N-gramů a okamžité ukončení programu tak způsobí ztrátu už spočtených výsledků extrakce. Jednoduchá analýza možného řešení je jedním z témat 7 kapitoly.

5 Analýza implementace programu

Metody automatické detekce kolokací vrácejí přesnější výsledky v případě obsáhlejších dat. Optimalizace **paměťové náročnosti** programu proto bude hlavním cílem. I přesto je důležité přihlížet také k časové složitosti. Program, který využívá minimum paměti, ale jehož výsledků se nedočkáme, je v praxi nepoužitelný.

5.1 Volba jazyka a platformy

Volba jazyka a operační platformy byla součástí doplňujících podmínek k vypracování práce, jenž zněly takto:

1. Program musí být implementován v jazyce C nebo C++.
2. Program musí fungovat na 64-bitové platformě.
3. Program musí fungovat na operačním systému Linux.

První podmínka je pro tento typ programu v podstatě axiomatická. Z hlediska tvorby programu s důrazem na paměťovou efektivitu je „jazyk" C/C++ zcela jistě nejlepší volbou. Na autorovi tak zůstala pouze volba, zda zvolit jazyk C nebo C++. Vybrán byl nakonec jazyk C++, protože povaha zadání vyžaduje použití různých datových kontejnerů a jen ztěží lze v tomto směru vymyslet něco lepšího, než je „Standard Template Library" [9]. Program přesto používá některé standardní knihovny jazyka C.

Druhá podmínka je také velmi přirozená — 32-bitové procesory dokáží adresovat pouze 4 GB paměti, což je v případě objemných vstupních dat příliš málo.

Volba operačního systému pak byla motivována pozdějším nasazením aplikace (právě v prostředí OS Linux).

5.2 Efektivní řešení paměťové složitosti

Efektivního využití paměti lze obecně dosáhnout pomocí kombinace dvou faktorů: dobře provedené **analýzy problému** (včetně volby vhodného programovacího jazyka) a efektivním použitím všech programovacích **technik**, jenž zvolený jazyk dovoluje.

5.2.1 Analýza paměťové složitosti extrakce N-gramů

V průběhu práce programu projdou vstupní data zpracováním, jehož výsledkem je výpočet výstupních statistik pro nalezené N-gramy (popř. další data jako např. kontext N-gramů). Během tohoto zpracování program pracuje s velkým množstvím těchto objektů: *řetězců*, *slov* a *N-gramů*. Optimalizace jejich reprezentace v paměti programu tak přináší zdaleka nejvýraznější úsporu celkově požadované paměti.

Soustředíme se proto právě na optimalizaci paměťové reprezentace *řetězců*, *slov* a *N-gramů*. V této kapitole provedeme analýzu jejich vlastností, v kapitole 6 se budeme zabývat detaily implementace.

Vlastnosti, které mají vliv na optimalizaci paměťové složitosti uvedených objektů (v rámci naší úlohy), jsou *unifikovatelnost*, *persistence* a *kontext*, přičemž platí, že:

- Řetězce a morfologické značky jsou unifikovatelné, persistentní, nemají kontext.
- Slova jsou unifikovatelné (ale jsou značně variabilní), nejsou persistentní, mají kontext.
- N-gramy jsou unifikovatelné, persistentní, mají kontext (dán kontextem slov, jenž N-gram tvoří).

Unifikovatelnost

Pro každé dva objekty stejného typu lze definovat, zda jsou si identické a dva identické výskyty stejného objektu nejsou rozlišitelné. V případě unifikovatelných objektů tak postačí mít v paměti jednu kopii daného unikátního objektu a v místech použití v programu se na ní odkazovat.

Persistence

Informace o výskytu objektu daného typu je nutná až do konce běhu programu (vypsání výsledků). Persistentní objekty nevyžadují samostatnou dealokaci — díky tomu pro jejich uložení lze snadno navrhnout efektivní *memory pool* (viz následující podkapitola).

Kontext

Nepostačí znát informaci o výskytech jednotlivých objektů, ale je nutné znát také jejich okolí (sousední objekty). Objekty mající kontext vyžadují buďto „kontextové“

uložení v paměti nebo datové položky navíc (s informací o kontextu). V programu jsou na různých místech použity oba přístupy.

5.2.2 Techniky pro úsporu paměti (i času)

Techniky pro úsporu paměti by bylo možné rozdělit dále do dvou kategorií: techniky použitelné v jazyce C++ a obecně použitelné (programátorské) techniky. Je ovšem někdy dost obtížné vytyčit hranici mezi oběma kategoriemi a zde to ani není nutné.

Předvýpočty (inicializace statických položek)

Parametr N má velký vliv na chování programu, potřebuje jej znát téměř každý modul programu a závisí na něm funkčnost mnoha tříd. Jeho hodnota samozřejmě není známá při kompilaci programu, ale je známá ihned po jeho spuštění a následně se již nemění.

Po startu programu lze proto provést předvýpočty návratových hodnot metod některých objektů, jenž závisí právě na parametru N. Tyto předvýpočty se většinou ukládají do statických proměnných příslušných tříd.

Bufferování

Program bufferuje načítaná vstupní data tak, aby bylo možné extrahování N-gramů (pro extrakci je nutná vždy celá věta) a také případné zjišťování kontextu. Zjišťování kontextu vyžaduje přístup ke slovům v kontextu o rozsahu, jenž může přesahovat aktuální větu (viz příslušná kapitola).

Díky bufferování tak není třeba načítat do paměti celý vstupní soubor najednou, postačí maximálně několik vět (velikost povoleného rozsahu je omezená konstantou v programu).

Zarovnávání („data alignment“) a sbalení („packing“)

Nyní lehce předběhneme a zkusíme navrhnout datovou strukturu reprezentující jednu složku N-gramu. Mohla by vypadat nějak takto:

```
struct NGramMember {
      unsigned char parent; // sizeof(unsigned char) = 1
      unsigned int lemma; // sizeof(unsigned int) = 4
      unsigned short tag; // sizeof(unsigned short) = 2
      unsigned char dependency;
}
```

Když se pokusíme zjistit, jaká bude výsledná velikost takto definované struktury, budeme překvapeni. Sečtením velikosti všech položek získame výsledek 8 bajtů, ale pro většinu (32-bitových) překladačů bude volání operátoru *sizeof(NGramMember)* ve skompilovaném programu vracet hodnotu 12!

Je tomu tak proto, že většina překladačů tzv. „zarovnává" položky struktur na určité adresy (hranice). Např. *int* je typicky zarovnán na adresu, jenž je násobkem šířky slova[12] u daného počítače [10, kap. 5.7]. Proto pokud chceme navrhovat struktury s minimálními paměťovými nároky, musíme jejich položky deklarovat v sestupném pořadí podle jejich velikosti — tímto efekty případného zarovnávání minimalizujeme.

Nezbavíme se tímto ale případného zarovnání struktury na jejím konci. Pokud je paměť důležitým zdrojem, jak je tomu v našem případě, pak se vyplatí vždy navrhovat struktury tak, aby i jejich celková velikost „zarovnávala". Alternativním řešením je technika nazývaná *pragma packing* — použití speciálního *pragma* příkazu[13], kterým můžeme ovlivnit hranici, na kterou je prováděno zarovnání, na libovolnou dvojkovou mocninu.

Skládání

Skládání s výhodou použijeme v situaci, kdy máme společně ukládat dva různě velké číselné údaje a datový typ pro větší z nich má kapacitu dostatečně velkou i pro uložení menšího údaje.

Příklad: Chceme uložit dvě nezáporná čísla, první s maximální hodnotou 2^8-1, druhé s maximální hodnotou 2^{24}-1. Pro první postačí 8-bitový číselný typ, pro druhé ovšem musíme použít 32-bitový číselný typ — tj. celkem 40 bitů, ačkoli uložení obou čísel nikdy nebude vyžadovat více než 32 bitů.

C++ nabízí speciální typ struktury — *union* — strukturu, kde má každý prvek stejnou adresu [10, kap. 10.4]. Použití unie umožňuje složit oba číselne údaje tak, že v paměti zabírají stejné místo — tj. pouze místo vyhrazené pro uložení většího údaje. S menším údajem ale dokážeme manipulovat tak, že zabírá nejvýznamnější bity společné datové položky, které nikdy při práci s větším údajem nevyužijeme.

12 Pojem slovo je zde použit ve smyslu nejmenší paměťové jednotky, se kterou počítač pracuje.

13 *Pragma* příkaz je speciální příkaz pro překladač. Jeho význam závisí na implementaci, proto už z principu se jedná o nepřenositelnou techniku!

Memory pooling

Při dynamické alokaci (pomocí *operátoru new*) kromě paměti pro objekt je navíc vyhrazen i malý kus paměti pro uložení velikosti alokované paměti, tak aby *operátor delete* věděl, kolik paměti má uvolnit. Tato informace se běžně ukládá před paměť vyhrazenou pro objekt [11, kap. 10]. Pokud je objekt malý, pak celkové množství paměti nutné pro jeho dynamickou alokaci je **výrazně** větší než paměť, kterou objekt přímo vyžaduje.

Proto programy v C/C++ jenž dynamicky inicializují velké množství malých objektů často používají speciální techniku pro ošetření těchto „ztrát" paměti, tzv. *memory pooling*. Paměť určená pro uložení malých objektů je alokována ve větších blocích, jenž se poté postupně zaplňují tak, jak jsou objekty inicializovány. Když dojde k vyčerpání bloku, alokuje se další. Pouhé přiřazování volné paměti v rámci bloku zpravidla není obtížné, dealokace jednotlivých objektů už je složitější a různé implementace ji řeší různými způsoby.

Placement new

Tuto techniku samostatně nelze označit jako techniku pro úsporu paměti, ale použití *operátoru placement new* je často jediným způsobem, jak používat memory pools pro objekty[14].

Ukládání unikátních tvarů

Pracujeme-li s objekty, jenž lze unifikovat (viz předchozí podkapitola), můžeme tuto jejich vlastnost použít pro velmi výrazné snížení použité paměti.

Efektivita tohoto řešení je však závislá na několika faktorech:

1. Velikosti domény všech možných hodnot objektů daného typu — tu lze většinou s různou přesností odhadnout. Např. odhad počtu lemmat lze (velmi zhruba) provést na základě odhadů pro počty slov v češtině[15].
2. Redundanci objektů — čím větší redundence, tím větší úspora paměti. Redundance většinou závisí pouze na konkrétních vstupních datech.

14 Operátor *placement new* umožňuje určit místo v paměti, ve kterém bude uložena instance nově vytvářeného objektu (zde je pojmem *objekt* míněn datový typ s konstruktorem a vytvoření nové instance je důsledkem volání konstruktoru).

15 Opravdu by se jednalo o velmi hrubý odhad. I když se při zpracování omezíme jen na české texty, tak i ony obsahují spoustu cizích slov.

3. Podílu velikosti objektů a velikosti odkazu na ně — nemá smysl se snažit unifikovat objekty, jejichž reprezentace v paměti zabere méně místa než reprezentace odkazu na ně.

V rámci implementace programu můžeme ovlivnit pouze poslední faktor, tj. efektivitu odkazování. V C++ lze řešit různými způsoby, např.:

1. Používáním ukazatele na unikátní kopii objektu. Zde je nutné mít na paměti, že velikost ukazatele není obecně vždy stejná (závisí na velikosti adres, které používá daný stroj).
2. Číselným identifikátorem, jenž danou kopii identifikuje v rámci nějakého indexovaného úložiště (kontejneru). Identifikátor může být každému objektu přiřazen na základě jeho pozice v úložišti nebo může být spočítán na základě dat objektu — pak se jedná o obdobu *hašování*.

Při ukládání unikátních tvarů šetříme paměť na úkor času. Větší spotřebu strojového času způsobuje především zjišťování, zdá kopie nově načteného objektu již existuje. Časově náročnější je však také přístup k datům.

5.3 Algoritmus extrakce N-gramů

Princip algoritmu pro extrakci N-gramů ze závislostního stromu věty jsme již stručně zmínili v kapitole 3. Idea algoritmu je velmi jednoduchá. V závislostním stromě věty vyhledáme všechny souvislé podstromy velikosti N, protože ty odpovídají hledaným kolokujícím N-gramům. Jedná se tedy o úlohu **hledání všech souvislých podstromů velikosti N**.

Jednoduchým řešením takové úlohy je rekurzivní algoritmus, ale ve své neoptimalizované podobě má exponenciální časovou složitost. Pokud by si rekurzivní algoritmus pamatoval již spočtené výsledky, dosáhneme zcela jistě zlepšení, ale algoritmus se stane zbytečně složitým. Elegantním řešením problému je algoritmus založený na principech dynamického programování, tj. postupné skládání dílčích výsledků.

Algoritmus hledání souvislých podstromů velikosti N

Vstupem algoritmu je parametr N a *indexovaný strom T*. Indexovaný strom je takový strom, jehož uzly jsou očíslovány přirozenými čísly od 1 do |T| (kde |T| je počet uzlů stromu) a každý index se vyskytuje právě jednou, takže každý uzel

stromu má unikátní index. Indexovaný strom používáme proto, že jej lze zcela přirozeně sestavit na základě údajů ze závislostního stromu věty. Podotkněme ještě, že strom **nemusí** být souvislý.

Výstupem algoritmu je množina souvislých podstromů stromu T velikosti N. Ta může být i prázdná, pokud T neobsahuje souvislý podstrom o alespoň N uzlech.

Každý uzel stromu obsahuje informaci o rodiči a svých dětech. Navíc pro každý uzel udržujeme seznam již nalezených souvislých podstromů, pro které je daný uzel kořenem (ovšem má smysl si pamatovat pouze podstromy velikosti rovné nebo menší než N). Na začátku algoritmu jsou všechny seznamy všech uzlů prázdné.

Algoritmus dále pracuje s pomocnou frontou, do které jsou postupně zařazovány uzly, jenž mají projít zpracováním.

Na začátku algoritmu je pomocná fronta inicializována **listy** stromu.

Algoritmus pak pokračuje tak, že postupně zpracovává frontu až do jejího vyčerpání. Fáze zpracování má dvě části:

1. Nejdříve z fronty vyjmeme jeden uzel (U) a postupně provedeme všechny kombinace všech podstromů jeho poduzlů takové, že spojením vznikne nový (souvislý) podstrom — ten uložíme do seznamu podstromů U. Do seznamu také přiřadíme jeden podstrom velikosti 1, jenž tvoří právě uzel U.
2. V druhé fázi zjistíme, zda všichni sourozenci U již byli zpracováni (pro tento účel si udržujeme speciální seznam čítačů). Pokud ano, zařadíme rodiče U do fronty. Pokud U rodiče nemá (je kořenem T), neděláme nic.

Z popisu algoritmu vidíme, že každý uzel projde frontou a bude zpracován právě jednou. V případě naivního rekurzivního algoritmu by uzly stromu byly zpracovány mnohokrát (v závislosti na tvaru stromu a hloubce uzlu).

Pro účely extrakce podstromů velikosti N algoritmus upravíme tak, že během 1. fáze zpracování kontrolujeme velikost nově vzniklého podstromu a podstromy velikosti N zařazujeme na výstup algoritmu.

5.4 Algoritmus výpočtu polí kontigenční tabulky

O tom, co jsou to kontigenční tabulky a jaký mají význam pro detekci kolokací jsme se zmínili v jedné z předchozích kapitol. Připomeňme, že kompletní kontigenční tabulka pro N-gram $w_1 \ldots w_n$ obsahuje kromě frekvence výskytu

N-gramu i frekvence výskytu „N-gramů“ $\neg w_1 \ldots \neg w_n$, $w_1 \ldots \neg w_n$ atd. Postup, jak tyto frekvence vypočítat, je obsahem této kapitoly.

5.4.1 Rozšíření pojmu N-gram

Prozatím jsme pojem „N-gram“ používali pro označení jednoduché posloupnosti slov. Nyní tuto definici rozšíříme: **N-gram bude n-ticí symbolů, jenž odpovídají určitému jevu.** Symboly můžou signalizovat výskyty nějakých slov a pak se jedná o N-gram ve smyslu původní definice, ale zavedeme i další symboly, jenž zahrnují i jevy jako „*výskyt jiného slova než*“ a „*výskyt jakéhokoli slova*“.

Význam symbolů je tedy následující (v závorkách je uvedeno označení, jenž je později pro usnadnění používáno):

- w^i — na i-té pozici N-gramu je slovo w (tzv. *běžný člen*)
- $\neg w^i$ — na i-té pozici N-gramu je jakékoli jiné slovo než w (tzv. *člen s negací*)
- $*^i$ — na i-té pozici N-gramu je jakékoli slovo (tzv. *hvězdička*)

Jelikož u skupiny symbolů je pozice každého symbolu v rámci N-gramu jednoznačně určená, budeme (horní) index pozice vynechávat (naopak budeme používat spodní index pro znázornění, že se nemusí jednat o výskyty stejných slov).

Definice 5.1 (Úplný N-gram)

> N-gram, který obsahuje pouze běžné členy (tj. N-gram ve smyslu původní definice) budeme označovat jako *úplný N-gram*.

Úplné N-gramy nás zajímají nejvíce, protože představují kandidáty na kolokace a pro ně program počítá výsledné statistiky. K zjištění těchto statistik ale potřebujeme více údajů než jen četnost jednotlivých úplných N-gramů. Definujeme si proto tři různé skupiny N-gramů.

Definice 5.2 (Skupiny a stupně N-gramů)

> N-gramy, které obsahují pouze běžné členy a hvězdičky (tj. w_1*, $*w_2$, w_1*w_3 apod.), budeme označovat jako *hvězdičkové N-gramy*. Počet běžných členů, které hvězdičkový N-gram obsahuje, budeme nazývat jeho *stupněm*.
>
> N-gramy, které obsahují pouze běžné členy a členy s negací (tj. $w_1\neg w_2$, $\neg w_1 w_2$, $w_1\neg w_2 w_3$ apod.), budeme označovat jako *smíšené N-gramy*. Obdobně jako výše

definujeme *stupeň smíšeného N-gramu* jako počet běžných členů, které N-gram obsahuje.

N-gramy, které obsahují pouze hvězdičky a členy s negací (tj. $*\neg w_2$, $\neg w_1*$, $*\neg w_2*$ apod.), budeme označovat jako *negované hvězdičkové N-gramy*. *Stupněm* negovaného hvězdičkového N-gramu je počet **členů s negací**, které obsahuje.

Pozn.: N-gramy s kombinacemi všech tří symbolů pro nás nebudou zajímavé.

Všimněme si, že frekvence výskytu **smíšených** N-gramů jsou hodnotami polí kontigenční tabulky! Můžeme provést také několik dalších pozorování, které jsou obsahem tvrzení 5.1.

Tvrzení 5.1

1.) Pro daný úplný N-gram $w_1 \ldots w_n$ existuje $\mathbf{2^N}$ různých N-gramů v každé skupině.

2.) Jevy, jenž odpovídají **smíšeným** N-gramům, jsou vzájemně **nezávislé** a navíc pokrývají celý pravděpodobnostní prostor (výskyt každého úplného N-gramu N_1 odpovídá vždy jednomu jevu ze skupiny smíšených N-gramů pro libovolný úplný N-gram N_2).

3.) Každé dvě skupiny mají právě jeden společný N-gram[16].

5.4.2 Vztahy mezi četnostmi různých skupin N-gramů

V této kapitole budeme implicitně předpokládat, že mluvíme-li členech některé skupiny N-gramů, máme na mysli jejich výskyt vzhledem **k danému úplnému N-gramu** (tj. nehovoříme o hvězdičkových N-gramech všech extrahovaných N-gramů).

Pro výpočet testových statistik potřebujeme znát četnosti smíšených N-gramů, protože, jak jsme si již uvedli, jejich hodnoty korespondují s poli kontigenční tabulky. Počítání těchto četností během fáze extrakce je bohužel podstatně náročnější než v případě hvězdičkových N-gramů. Každý detekovaný (úplný) N-gram zvětší čítač výskytů u 2^N *hvězdičkových N-gramů* (a ty už zahrnují i onen úplný N-gram). Oproti tomu *smíšených N-gramů*, které na každý takový N-gram „sedí“, je přesně tolik, kolik je všech unikátních úplných N-gramů (to vyplývá z 2. bodu tvrzení 5.1).

16 Např. pro trigramy jsou to: $w_1w_2w_3$ (pro 1. a 2. skupinu), $\neg w_1\neg w_2\neg w_3$ (pro 2. a 3. skupinu) a *** (pro 1. a 3. skupinu).

Proto nám nezbývá nic jiného, než četnosti smíšených N-gramů vypočítat na základě četností hvězdičkových N-gramů. Není to ovšem nikterak obtížné.

Víme, že součet polí kontigenční tabulky a taktéž i součet četností všech smíšených N-gramů (**libovolného** úplného N-gramu) je roven počtu všech (extrahovaných) N-gramů nebo také četnosti **hvězdičkového** N-gramu stupně 0 (jde o jiné vyjádření četnosti téhož jevu).

Zamysleme se teď nad obdobným vyjádřením četnosti libovolného **hvězdičkového** N-gramu **stupně 1** (kde w je na i-té pozici). Zřejmě na jeho četnost nemají vliv právě jevy, které odpovídají nějakému smíšenému N-gramu s negací w na i-té pozici. A naopak všechny jevy, které odpovídají nějakému smíšenému N-gramu, jenž **obsahuje** w na i-té pozici, odpovídají také uvažovanému hvězdičkovému N-gramu. Takže jeho četnost je součtem četnosti všech smíšených N-gramů, jenž obsahují w na i-té pozici, vyjádřeno vzorcem pak:

$$C(*...w_i...*) = C(w_1...w_i...w_n) + ... + C(\neg w_1...w_i...\neg w_n)$$

Snadno nahlédneme, že obdobná situace platí pro libovolný stupeň.

Tvrzení 5.2

> Mějme hvězdičkový N-gram S stupně d, který má na pozicích $i_1,\ldots,i_d$ **běžné členy** a který odpovídá nějakému úplnému N-gramu N. Pak jeho četnost $C(S)$ je rovna součtu četnosti **všech smíšených N-gramů** daného N-gramu N, které mají na pozicích $i_1,\ldots,i_d$ také **běžné členy**.[17]

Uvedené tvrzení nám ovšem ještě nic neříká o tom, jak vypočítat četnosti všech smíšených N-gramů. Nyní se k tomu dostaneme.

Četnost libovolného hvězdičkového N-gramu stupně N-1 je dle tvrzení 5.2:

$$C(w_1...*_i...w_n) = C(w_1...w_i...w_n) + C(w_1...\neg w_i...w_n)$$

Vzorec pro četnost **smíšeného N-gramu** stupně N-1 tedy dostaneme snadným upravením výše uvedené rovnice. Navíc četnost zbývajících dvou N-gramů v rovnici známe, úplný N-gram $w_1\ldots w_i\ldots w_n$ je totiž zároveň i hvězdičkovým i smíšeným N-gramem.

17 Protože se jedná o hvězdičkové a smíšené N-gramy daného (stejného) úplného N-gramu, není nutná podmínka, že běžné členy N-gramu S a smíšených N-gramů na pozicích $w_{i1},..,w_{id}$ se musí shodovat — platí to automaticky.

Zkusme nyní obdobným postupem vyjádřit četnost hvězdičkového N-gramu stupně N-2. Obdržíme:

$$C(w_1 \ldots *_i \ldots *_j \ldots w_n) = C(w_1 \ldots w_i \ldots w_j \ldots w_n) + C(w_1 \ldots w_i \ldots \neg w_j \ldots w_n) + C(w_1 \ldots \neg w_i \ldots w_j \ldots w_n) + C(w_1 \ldots \neg w_i \ldots \neg w_j \ldots w_n)$$

Vidíme, že na pravé straně se (opět) vyskytuje **pouze jeden** smíšený N-gram $(w_1 \ldots \neg w_i \ldots \neg w_j \ldots w_n)$ stejného stupně (N-2). Zbytek jsou smíšené N-gramy vyššího stupně, které ale již **dokážeme** spočítat.

Pro lepší názornost uvedeme upravený vzorec – četnost smíšeného N-gramu stupně N-2 lze vyjádřit jako:

$$C(w_1 \ldots \neg w_i \ldots \neg w_j \ldots w_n) = C(w_1 \ldots *_i \ldots *_j \ldots w_n) - C(w_1 \ldots w_i \ldots w_j \ldots w_n) - C(w_1 \ldots w_i \ldots \neg w_j \ldots w_n) - C(w_1 \ldots \neg w_i \ldots w_j \ldots w_n)$$

Pokud bychom odvodili vzorec pro četnost hvězdičkového N-gramu stupně N-3, situace by byla **obdobná**. Opět by na pravé straně byl pouze jeden smíšený N-gram stupně N-3 a stupně všech dalších smíšených N-gramů ve vzorci by byly vyšší (tj. rekurzivně spočítatelné). Indukcí lze ukázat, že uvedená úvaha je platná pro libovolné N.

Uvedený postup zároveň naznačuje způsob výpočtu – postupně počítáme četnosti smíšených N-gramů od stupně N směrem ke stupni 0 a během výpočtu na „nižším stupni" používáme již spočtené hodnoty z „vyšších stupňů". Pro detailnější popis algoritmu si ovšem musíme zavést ještě jednu definici, tzv. *typ N-gramu*.

Definice 5.3 (Typ N-gramu)

Typ N-gramu, podobně jako stupeň, je číselnou hodnotou. Definujeme jej pro N-gramy všech skupin a to následovně:

Typ hvězdičkového N-gramu je hodnotou čísla v dvojkové soustavě, které vznikne přepisem složek daného N-gramu dle pravidel: $w \rightarrow 1$ a $* \rightarrow 0$ (běžné členy nahradíme jedničkou a hvězdičky nulou).

Typ smíšeného N-gramu je hodnotou čísla v dvojkové soustavě, které vznikne přepisem složek daného N-gramu dle pravidel: $w \rightarrow 1$ a $\neg w \rightarrow 0$ (běžné členy nahradíme jedničkou a členy s negací nulou).

Typ negovaného hvězdičkového N-gramu je hodnotou čísla v dvojkové soustavě, které vznikne přepisem složek daného N-gramu dle pravidel: $\neg w \rightarrow 1$ a $* \rightarrow 0$ (členy s negací nahradíme jedničkou a hvězdičky nulou).

Hodnoty typů pro trigramy všech skupin uvádí tabulka 5.

typ	stupeň	hvězdičkový N-gram	smíšený N-gram	neg. hvězd. N-gram
7 (111)	3	$w_1\ w_2\ w_3$	$w_1\ w_2\ w_3$	$\neg w_1\ \neg w_2\ \neg w_3$
6 (110)	2	$w_1\ w_2\ *$	$w_1\ w_2\ \neg w_3$	$\neg w_1\ \neg w_2\ *$
5 (101)	2	$w_1\ *\ w_3$	$w_1\ \neg w_2\ w_3$	$\neg w_1\ *\ \neg w_3$
4 (100)	1	$w_1\ *\ *$	$w_1\ \neg w_2\ \neg w_3$	$\neg w_1\ *\ *$
3 (011)	2	$*\ w_2\ w_3$	$\neg w_1\ w_2\ w_3$	$*\ \neg w_2\ \neg w_3$
2 (010)	1	$*\ w_2\ *$	$\neg w_1\ w_2\ \neg w_3$	$*\ \neg w_2\ *$
1 (001)	1	$*\ *\ w_3$	$\neg w_1\ \neg w_2\ w_3$	$*\ *\ \neg w_3$
0 (000)	0	$*\ *\ *$	$\neg w_1\ \neg w_2\ \neg w_3$	$*\ *\ *$

Tabulka 5: Typy a stupně trigramů různých skupin.

Snadno vypozorujeme, že stupeň N-gramu je roven počtu jedniček v bitovém zápisu typu (u všech skupin bez rozdílu).

Informace o typu nám velmi usnadní výpočet četnosti smíšeného N-gramu. Bitový zápis typu nám totiž poskytuje přesnou informaci o tvaru N-gramu. Hezky se to dá zobrazit, pokud si poslední vzorec (četnost smíšeného N-gramu stupně N-2) vyjádříme „bitově“ (*M* jsou smíšené, *S* je hvězdičkový N-gram)

$$C(M_{1\ldots0\ldots0\ldots1}) = C(S_{1\ldots0\ldots0\ldots1}) - C(M_{1\ldots1\ldots1\ldots1}) - C(M_{1\ldots1\ldots0\ldots1}) - C(M_{1\ldots0\ldots1\ldots1})$$

Pokud se soustředíme pouze na členy, jenž se různí (tj. i-tý a j-tý člen ve smyslu původní rovnice), vypozorujeme, že bitové zápisy typů smíšených N-gramů na pravé straně rovnice musí mít na některé z těchto pozic **jedničku**, kdežto vyjádřovaný smíšený N-gram tam musí mít pouze **nuly**. Je to důsledkem dvou již uvedených faktů:

1. Na pravé straně jsou pouze smíšené N-gramy vyšších stupňů než je vyjadřovaný N-gram.
2. Stupeň N-gramu je dán počtem jedniček v bitovém zápisu jeho typu.

Tvrzení 5.3

> Mějme libovolný smíšený N-gram M stupně d a typu t, hvězdičkový N-gram S stejného typu t a skupinu N-gramů G, kterou tvoří všechny smíšené N-gramy vyššího stupně než d takové, že mají v bitovém zápisu typu jedničky na **všech** pozicích, na kterých je má i bitový zápis typu t. Pak četnost M je rovná rozdílu četnosti S a součtu četností N-gramů ze skupiny G.

Jedná se de facto o vyjádření tvrzení 5.2 z pohledu bitových zápisů. Shodnost typů pro M a S zaručuje, že všechny svoje běžné členy mají na stejných pozicích (tj. mají tvar $w_1 \ldots \neg w_i \ldots \neg w_j \ldots w_n$ a $w_1 \ldots *_i \ldots *_j \ldots w_n$ pro libovolné, různé indexy i a j). Podmínka pro N-gramy ze skupiny G zajišťuje, že také obsahují všechny běžné členy N-gramů M i S (a dále viz tvrzení 5.2).

Dosadíme-li M, G a S do výše uvedených vzorců, pak obdržíme:

$$C(S) = C(M) + \sum_{g \in G} C(g)$$

Tvrzení 5.3 je pak pouhým upravením tohoto vzorce:

$$C(M) = C(S) - \sum_{g \in G} C(g)$$

Shrnutí

Implementace algoritmu pro výpočet polí kontigenční tabulky využívá všechny výše uvedené poznatky. Hodnoty polí jsou vypočteny na základě četností smíšených N-gramů. Tyto četnosti se počítají „od shora“, tj. nejdříve se „spočte“ četnost smíšeného N-gramu stupně N a dále se počítají četností N-gramů nižších stupňů s využitím už spočtených četností N-gramů vyšších stupňů. Pro snadnou manipulaci s N-gramy se používají bitové zápisy jejich typů.

5.5 Algoritmus výpočtu očekávané četnosti

Jako poslední v této kapitole si uveďme algoritmus pro výpočet očekávané četnosti. Využijeme k tomu poznatky z předchozí podkapitoly.

Hodnotu očekávané četnosti potřebujeme znát pro výpočet teměř všech statistik. Pro některé statistiky ovšem potřebujeme znát celou (kontigenční) tabulku očekávaných četností.

Očekávaná četnost (*E*) odpovídá hypotéze, že výskyty jednotlivých slov v textu jsou nezávislé a tedy výskyt daného N-gramu je součinem N nezávislých jevů. Očekávaná četnost smíšeného N-gramu je tedy součinem pravděpodobností výskytů jeho jednotlivých členů vynásobených počtem všech extrahovaných N-gramů.

Pro trigram $w_1 \neg w_2 w_3$ je to např.:

$$E(w_1 \neg w_2 w_3) = P(w_1^1) * P(\neg w_2^2) * P(w_3^3) * C(***)$$

S použitím označení definovaného v předchozí podkapitole, pak:

$$E(w_1 \neg w_2 w_3) = \frac{C(w_1 **)}{C(***)} * \frac{C(* \neg w_2 *)}{C(***)} * \frac{C(** w_3)}{C(***)} * C(***)$$

Četnosti hvězdičkových N-gramů počítáme přímo během extrakce, ale (zatím) neznáme způsob, jakým vypočítat četnosti **negovaných** hvězdičkových N-gramů.

Je to ovšem velmi snadné. Stačí si uvědomit, že hvězdičkové N-gramy se od svých negovaných protějšků liší pouze onou negací. Stejně jako v případě hvězdičkového N-gramu stupně 1 platí:

$$C(*... w_i ...*) = C(w_1 ... w_i ... w_n) + \ ... \ + C(\neg w_1 ... w_i ... \neg w_n)$$

Tak obdobně pro negovaný hvězdičkový N-gram stupně 1 platí:

$$C(*... \neg w_i ...*) = C(w_1 ... \neg w_i ... w_n) + \ ... \ + C(\neg w_1 ... \neg w_i ... \neg w_n)$$

Příslušné tvrzení je zbytečné uvádět, protože by bylo pouze přeformulováním tvrzení 5.2 (v textu tvrzení by formulaci „běžné členy“ pouze nahradila formulace „členy s negací“).

Shrnutí

Pro výpočet četností **negovaných** hvězdičkových N-gramů využijeme již vypočtené četnosti smíšených N-gramů na základě reformulace tvrzení 5.2 pro členy s negací. Implementace programu pak opět pro usnadnění manipulace s N-gramy různých typů používá jejich bitové zápisy.

6 Popis implementace programu

V následujícím textu jsou probírány detaily implementace programu, proto se od čtenáře předpokládá znalost terminologie jazyka C++ a také základní znalost Standard Template Library (STL).

Cílovou platformou pro nasazení programu byla (je) 64-bitová verze operačního systému Linux. Program byl ale vyvíjen se snahou o snadnou přenositelnost. Tabulka 6 uvádí vztahy mezi základními datovými typy a jejich velikostmi pro různé překladače, kterými byl program v rámci vývoje skompilován. Tyto údaje jsou použity jako základ odhadů velikosti různých struktur, které se v následujícím textu vyskytují.

datový typ	**Visual Studio 2005 (32-bit)**	**GCC 4.1.2 (32-bit)**	**GCC 4.1.2 (64-bit)**
unsigned char	1 bajt	1 bajt	1 bajt
unsigned short	2 bajty	2 bajty	2 bajty
unsigned int	4 bajty	4 bajty	4 bajty
size_t	4 bajty	4 bajty	8 bajtů

Tabulka 6: Velikosti základních bezznamenkových (unsigned) typů vrácené operátorem sizeof() v různých verzích překladačů, kterými byl program skompilován.

Jak je z tabulky 6 vidět, velikosti uvedených typů se mezi oběma uvedenými 32-bitovými verzemi překladačů neliší a 64-bitová verze p5ekladače GCC se odlišuje pouze ve velikosti typu *size_t*. Standard jazyka C++ ovšem velikosti číselných typů definuje dost svobodně [10, kap. 4.6]. Při kompilaci programu v jiném překladači nebo na jiné platformě je proto nutné ověřit, zda jsou velikosti těchto typů stejné. Jinak může mít program větší paměťové nároky (v tom lepším případě) nebo nebude schopný zpracovat obsáhlejší textové korpusy (v tom horším případě).

6.1 Uložení řetězců

V PDT formátu existují tři položky typu řetězec, ale *povrchový výskyt slova* není pro extrakci N-gramů důležitý, proto při zpracování dat rozlišujeme pouze dva typy řetězců: *lemmata* a *morfologické značky*.

S oběma typy se v programu pracuje stejně a to na již popsaném principu „ukládání unikátních tvarů“. Ačkoli princip sám je jednoduchý, neexistuje jeden správný způsob implementace — zejména v případě řetězců volba konkrétního řešení závisí na několika různých faktorech.

6.1.1 Zdůvodnění zvolené implementace

C-string vs. std::string

Pro řetězce existuje v STL speciální třída string. Různé její implementace jsou ovšem vždy paměťově náročnější než použití „obyčejného“ céčkovského řetězce[18] [9, kap. 15] a jelikož nám na paměti záleží více než na jednoduché práci s řetězci, použijeme právě céčkovské řetězce.

Ukazatel vs. číselný index

Protože jsme se rozhodli pro použití céčkovských řetězců, „nabízí se“ nám přímočará implementace, jenž využije ukazatele na tyto řetězce jako odkazy na jejich unikátní instance. Věc ovšem není zdaleka tak jednoduchá. Jedním z předpokladů programu bylo to, že bude používán na 64-bitových strojích, tj. k adresaci paměti jsou použity 64-bitové adresy. Oproti tomu jen polovina (32) bitů postačí pro reprezentaci cca 4,3 mld čísel. Očekávaný počet *lemmat* přitom jen ztěží dosáhne této hodnoty — různé odhady stanoví počet slov v českém jazyce v řádu statísíců [12], takže 4,3 miliardy indexů představují dostatečnou kapacitu. Různých morfologických značek přitom bude ještě mnohonásobně méně.

V našem případě jsou tedy pro implementaci odkazů na unikátní instance řetězců výhodnější číselné indexy. Pro přístup k řetězcům sice budeme navíc potřebovat převodní tabulku: *číselný index* → *ukazatel*, ale v tabulce budou záznamy pouze pro unikátní řetězce. Oproti tomu na každém odkazu na unikátní instanci řetězce kdekoli **v programu** ušetříme několik bajtů paměti[19], přičemž takových odkazů může být v případě každé instance velmi mnoho — zejména v případě zjišťování kontextu.

Použití číselných indexů má ještě jednu výhodu: indexy pro lemma a morfologickou značku se dají „skládat“[20]. V praxi je ve výchozí verzi programu pro index lemma vyhrazeno **24 bitů** (tj. maximálně cca 16,7 mln různých hodnot) a pro

18 Tj. řetězec je posloupnost znaků ukončena znakem NULL (nulovým bajtem).
19 Přesná hodnota závisí na rozdílu velikosti ukazatele a velikosti použitého číselného indexu.
20 Technika „skládání“ byla popsána v 5. kapitole.

index morfologické značky **8 bitů** (tj. maximálně 256 různých hodnot). Oba údaje společně proto vyžadují pouze **4 bajty** paměti.

Lineární vs. logaritmický čas vkládání

Při načtení nového řetězce potřebujeme zjistit, zda již existuje jeho unikátní verze v paměti programu nebo jej teprve musíme uložit (pak je doopravdy novým řetězcem). Toto zjištění lze realizovat různými způsoby, přičemž nejjednodušší nevyžaduje žádná data navíc — stačí projít převodní tabulku a porovnat všechny doposud uložené řetězce. Načtení nového řetězce pak bude mít lineární časovou složitost.

Ačkoli je hlavním cílem efektivní použití paměti, toto je situace, kdy se vyplatí obětovat paměť na úkor času a definovat databázový index nad převodní tabulkou, který umožní vyhledání korespondujícího řetězce v rychlejším čase (přesněji logaritmickém čase — viz níže).

6.1.2 Praktické provedení zvolené implementace

Pro uložení posloupností znaků, jenž tvoří jednotlivé řetězce, použijeme znakové *memory pools* (pro typ *char*). Protože jsou řetězce persistentní[21], není nutné aby použitý memory pool zvládal jejich samostatnou dealokaci, takže kromě samotného řetězce a ukazatele na něj není nutné uložení žádných dalších informací.

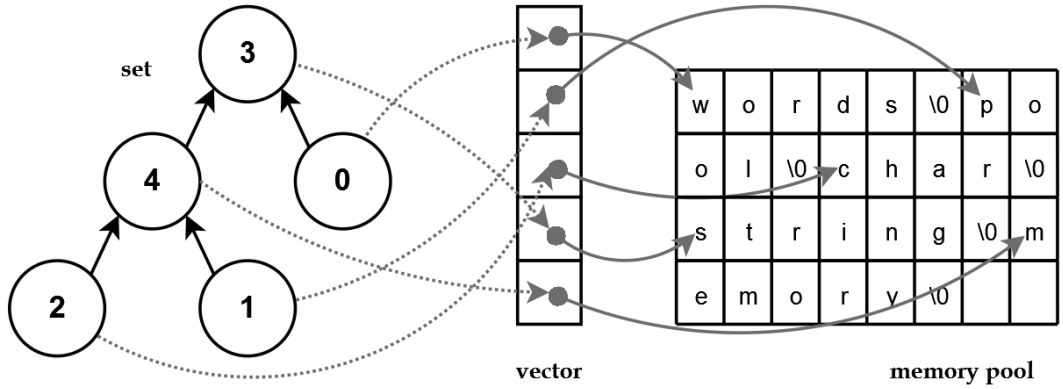
Ukazatele na jednotlivé řetězce uložíme do kontejneru typu *vector*, který tak zároveň poslouží jako převodní tabulka: *číselný index* → *řetězec*. Pro databázový index nad touto tabulkou použijeme kontejner typu *set*. Jeho prvky budou číselné indexy lexikograficky uspořádané podle hodnot řetězců, na které odkazují. Díky tomu dokážeme v logaritmickém čase zjistit, zda je daný řetězec v úložišti [13, kap. 6.5].

Schéma uložení řetězců prezentuje obrázek 2.

Aby bylo možné použít set takovým způsobem, jak jsme si popsali výše, musíme pro něj definovat speciální *komparátor*[22]. Chceme totiž v setu udržovat indexy do převodní tabulky, ale setříděné podle hodnot řetězců, ke kterým se vztahují. Proto komparátor musíme inicializovat až za běhu a předat mu referenci na konkrétní převodní tabulku (úložiště), ke kterému se dané indexy (řetězce) vztahují.

21 Viz podkapitola 5.2.1.

22 Funkční objekt, jenž obsahuje funkci pro porovnání dvou prvků setu.



Obrázek 2: Schéma uložení řetězců. Plné čáry představují vztah mezi řetězci a ukazateli na ně, tečkované čáry představují vztah mezi indexy v setu a příslušnými pozicemi v rámci vectoru. Indexy v setu jsou setříděné podle hodnot příslušných řetězců.

Všechny 3 části (tj. memory pool, vector a set) jsou zapouzdřeny vně společné šablonové třídy. Šablona třídy má jediný parametr a to **datový typ** pro číselné indexy. Může jím být libovolný beznamenkový typ. Měl by však být schopen pojmout indexy všech unikátních řetězců, které chceme ukládat do daného úložiště[23]. Třída kontroluje, zda vložení nového řetězce nezpůsobí přetečení indexu. Na případné přetečení reaguje vyhozením speciálním vyjímky.

Současná implementace programu tuto vyjímku vyhodnocuje jako chybu a reaguje okamžitým ukončením programu[24]!

6.2 Slova

Slovo je objekt reprezentující jeden řádek vstupního souboru PDT. Slovo „vzniká“ při načtení daného řádku vstupního souboru, je uloženo do čtecího bufferu a poté **v rámci celé věty** použito při extrakci N-gramů a případném zjišťování kontextu. Pro další práci s N-gramy není nutné znát celá slova, pouze některé jejich položky — ty jsou během extrakce skopírovány, proto ve chvíli, kdy dané slovo (resp. věta, do které patří) přestane být pro extrakci a zjišťování kontextu relevantní, je možné jej zahodit.

Slova nemůžeme načítat jednotlivě, ale díky použití bufferování je počet slov v paměti v každém okamžiku práce programu velmi malý (ve srovnání s počtem

23 V programu lze typy použité pro indexy řetězců definovat v konfiguračním souboru.

24 Zcela jistě by lepším řešením bylo uložení mezivýsledků do speciálního souboru — vstupní analýza tohoto rozšíření je jedním z témat kapitoly 7.

řetězců a N-gramů). Jeho přesná hodnota závisí na délce a počtu vět, jenž jsou v dané chvíli v čtecím bufferu. Délku vět přitom můžeme předpokládat za přirozeně omezenou a jejich počet je omezen velikostí kontextového okénka.

6.3 Uložení N-gramů

Efektivní reprezentace N-gramů v paměti programu představuje určitě nejsložitější problém celé úlohy. N-gramy mají podstatně větší variabilitu tvarů než řetězce a ta navíc roste spolu s parametrem N. Doslova každý bajt, který v případě N-gramů ušetříme, se znatelně projeví na celkové paměťové náročnosti programu.

6.3.1 Požadavky

Unifikaci N-gramů musíme provádět i bez důrazu na paměťovou efektivitu, protože chceme počítat frekvence jednotlivých unikátních výskytů.

V předchozí kapitole jsme navíc zjistili, že znalost pouhé frekvence výskytu úplných N-gramů nám nepostačí, ale potřebujeme si pamatovat frekvence výskytu všech hvězdičkových N-gramů.

Navíc v případě aktivní funkce zjišťování kontextu musíme pro N-gramy nějak ukládat i jejich kontext (kontext chceme ukládat pro každý hvězdičkový N-gram s vyjímkou stupně nula[25]).

6.3.2 Základní návrh

Pro každý N-gram si musíme v paměti udržovat informaci o jeho četnosti, složkách a případně i jeho stupni (resp. typu). Pro uložení četnosti použijeme typ *unsigned int*, každá složka N-gramu pak musí obsahovat:

- lemma — reprezentováno indexem příslušného řetězce (velikost 3 bajty)
- morfologickou značku — reprezentována indexem příslušného řetězce (velikost 1 bajt)
- index typu závislosti — reprezentován indexem (velikost 1 bajt)
- index řídícího slova — hodnota menší než N (postačí 1 bajt)

25 Hvězdičkový N-gram stupně 0 „odpovídá“ všem úplným N-gramům, takže jeho kontext obsahuje veškerá extrahovaná slova.

Abychom omezili efekt zarovnávání, použijeme techniku *pragma packingu*[26].

Deklarace příslušné struktury ve zdrojovém kódu pak vypadá přibližně takto:

```
class NGram {
      class Member {
            // TaggedLemma holds lemma and tag indices altogether.
            TaggedLemma tagged_lemma;
            ngram_size_t parent;
            dependency_index_t dependency;
      };
      // NGram only data is his frequency:
      freq_counter_t frequency;
};
```

Uložení stupně (typu)

Třída NGram zatím obsahuje pouze jednu datovou položku pro uložení frekvence. K ní bychom ovšem ještě rádi přidali další položku — pro uložení typu N-gramu[27].

Pro uložení typu N-gramu nepotřebujeme mnoho místa (1 bajt postačí), ale jak jsme si uvedli — každý bajt navíc se projeví. Proto pro uložení typu použijeme již popsanou techniku „skládání" a typ uložíme na místo nejvýznamnějších bitů čítače výskytů. Omezíme tak maximální použitelnou hodnotu čítače a zvětšíme tím riziko jeho přetečení, ale i tento problém (později) vyřešíme.

6.3.3 Komplikace

Definice struktur pro uložení N-gramu již máme, nyní nám zbývá vyřešit odkazování na jednotlivé dynamicky alokované instance třídy NGram a na (k ní) příslušné instance třídy NGram::Member.

Hlavní komplikací je v tomto případě variabilita počtu složek N-gramu. Máme-li hvězdičkový N-gram stupně S potřebujeme si pamatovat právě S složek (u složek nahrazených hvězdičkou je důležitá pouze jejich pozice, samy o sobě žádné další informace nenesou). Navíc ani N není z pohledu programu konstantní hodnota a nemůže být např. použita jako velikost pole[28].

26 Ve zdrojovém kódu ovšem příslušné pragma příkazy „zabalíme" do podmíněných příkazů preprocesoru, tak abychom je zobrazovali pouze „vyzkoušeným" překladačům.

27 Ukládat budeme typ N-gramu, protože stupeň na základě typu zjistíme snadno a typ navíc určuje pozici běžných členů a hvězdiček.

28 Konstatní je pouze maximální dovolená hodnota parametru N, ta se pro definici velikosti některých polí používá. Její použití zde by ovšem bylo ohromným plýtváním paměti.

Knihovna STL nabízí dynamické kontejnery a pro náš případ by plně postačil ten nejjednodušší z nich — *vector*. Různé implementace typu *vector* mají různé velikosti[29]: např. 12 bytů v případě knihovny STL, jenž je součástí překladače GCC (verze 4.1.2) nebo 16 bytů v případě knihovny STL, kterou obsahuje Microsoft Visual Studio 2005. Pro uložení dat N-gramu nám přitom postačí 4+N*6 bajtů, takže použití vectoru by zvětšilo objem použité paměti až o desítky procent. Tento nárůst by se v podobném měřítku promítnul i do celkového obsazení paměti.

6.3.4 Uložení N-gramů (bez kontextu)

Připomeňme si dva fakty, který jsme již zmínili:

1. Pro každý hvězdičkový N-gram známe počet jeho složek.
2. Počet složek každého hvězdičkového N-gramu je neměnný.

Uvedené fakty nám napovídají, že hromadná alokace složek by mohla být dobrým řešením. Pro uložení složek N-gramu můžeme používat jeden vector (resp. memory pool) a na první složku N-gramu se můžeme odkazovat číselným indexem (resp. ukazatelem). Jelikož známe počet složek pro každý N-gram, snadno zajistíme, aby různé N-gramy omylem „nesdílely“ svoje složky.

Výše uvedené řešení vyžaduje k datům N-gramu uložení „pouze“ jednoho indexu (resp. ukazatele) navíc, což je podstatně menší objem dat než v případě vectoru. Slovo **pouze** je však uvedeno v uvozovkách, protože i tento objem dat o několik procent zvětší velikost paměti nutné pro uložení N-gramu.

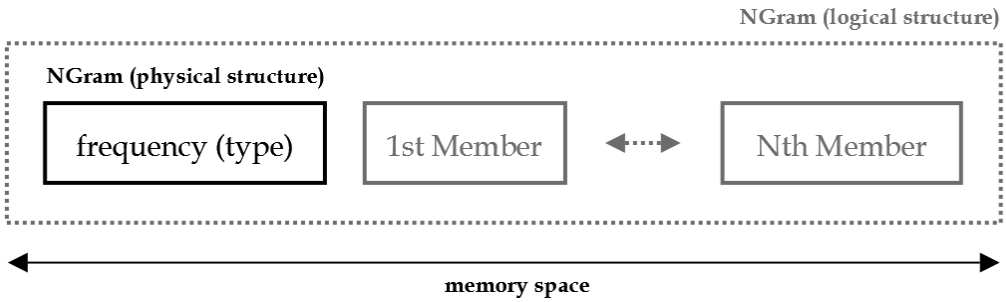
Dynamické alokaci se ovšem nevyhneme a nějaky ukazatel na dynamicky alokovaná data potřebujeme. Nejúspornější řešení by tak muselo místo dvou ukazatelů — jednoho na data N-gramu a jednoho na jeho složky — používat ukazatel jenom jeden. Na první pohled se to možná zdá jako nesmysl, ale v jazyce C++ to není problém.

C++ umožňuje opravdu neomezený přístup k paměti, kterou si **dynamicky naalokujeme**. Můžeme na námi zvolené adrese *X* inicializovat (uložit) objekt *A* a ihned za objektem *A* (tj. na adrese *X*+sizeof(*A*)) uložit objekt *B*, pokud nám na tom záleží. Nebo celou skupinu objektů *B* — a to je přesně to, co potřebujeme! Inicializovat N-gram (objekt A) na nějaké adrese v paměti, kterou máme k dispozici

29 Všechny následující velikosti platí v případě 32-bitových verzí knihovny STL a programu.

a ihned za ním inicializovat jeho složky (skupinu objektů B). Obrázek 3 znázorňuje celou situaci graficky.

K implementaci využijeme techniky probrané v minulé kapitole, tj. *memory pool* a operátor *placement new*. Memory pool naalokujeme zvlášť pro každý stupeň N-gramu (každý stupeň má různý počet složek a různí se proto i velikostí obsazené paměti), operátor placement new budeme používat pro inicializaci objektů pro reprezentaci složek na přesně daných adresách.



Obrázek 3: Logická a fyzická reprezentace N-gramu v paměti. Z fyzického pohledu objekt třídy NGram obsahuje pouze jedinou datovou položku (pro frekvenci — ta v sobě ovšem zahrnuje i typ N-gramu a příznak přetečení čítače), logicky k němu patří i jeho složky (objekty třídy Member), které jsou v paměti uloženy bezprostředně za ním. Z typu N-gramu lze zjistit, kolik členů má daná instance třídy NGram.

Pro danou instanci N-gramu pak zjistíme adresu jeho první složky tak, že k ukazateli *this*[30] přičteme velikost instance — dostaneme tak adresu paměti bezprostředně za instancí, tj. přesně adresu první složky. Adresy dalších složek pak snadno získáme pomocí *ukazatelové aritmetiky*.

6.3.5 Uložení kontextu

Slovo v kontextu definuje jeho lemma a morfologická značka a také je nutné ukládat jeho frekvenci (v kontextu daného N-gramu). Pro definici struktury pro slovo v kontextu použijeme opět techniku *pragma packingu*, aby případná změna datových typů použitých pro uložení indexů lemmat a morfologických značek nezpůsobila nepřiměřený nárůst její velikosti.

30 Uvnitř každé nestatické metody v C++ máme k dispozici ukazatel *this*, který ukazuje na aktuální instanci (instanci objektu, na kterém byla metoda zavolána) [10, kap. 10.2.7].

Kontext každého N-gramu může obsahovat různý počet slov. Ten lze jen ztěží odhadnout a navíc se může výrazně lišit pro různé N-gramy. To je pro implementaci velmi nepříjemné zjištění a znamená, že se v tomto případě použití kontejneru typu *vector* nevyhneme.

Slova uložená v kontejneru budeme udržovat sestupně setříděná podle jejich (průběžné) četnosti výskytu. Slova, která se v kontextu daného N-gramu vyskytují častěji, tak v rámci kontejneru nalezneme rychleji[31].

Kontejner pro kontext daného N-gramu uložíme do paměti hned za jeho poslední složku — využijeme tedy úplně stejného principu, jaký jsme si demonstrovali při popisu uložení složek N-gramu. V rámci definice třídy NGram, pak stačí vhodně upravit konstruktory (tak aby kontrolovaly, zda je zjišťování kontextu zapnuto), přidat metodu pro vložení nového slova do kontextu a metodu pro vrácení ukazatele na kontejner se slovy v kontextu.

Naprosto cíleně se v tomto případě vyhybáme použití dědičnosti a virtuálních funkcí. Ačkoli by to z hlediska objektově-orientovaného návrhu bylo určitě čistější řešení, stálo by nás další paměť navíc. Instance tříd s virtuálními funkcemi totiž vyžadují jednu (skrytou) datovou položku — ukazatel na příslušnou tabulku virtuálních funkcí [14, kap. 24].

6.3.6 Přetečení čítačů

Datové typy určené pro čítače frekvencí mají samozřejmě svoje maximum a situace, kdy dojde k překročení tohoto maxima a následnému *přetečení* čítače je velmi nepříjemná.

Přetečení čítače N-gramu

Pro uložení frekvence N-gramu jsme definovali datovou položku typu *unsigned int*, která dokáže pojmout číslo o velikosti cca 4,3 mld. U popisu implementace struktury pro uložení N-gramu jsme si ovšem uvedli, že v rámci nejvyšších bitů čítače držíme data pro identifikaci typu N-gramu. Pro čítač nám tak zbývá „pouze" 24 bitů, což je postačující paměť pro uložení čísla o velikosti cca 16,7 mln. Uvážíme-li, že **celková** velikost vstupních dat má být v řádu miliard **slov**, můžeme předpokládat, že pro většinu hvězdičkových N-gramů s takovou velikostí čítače

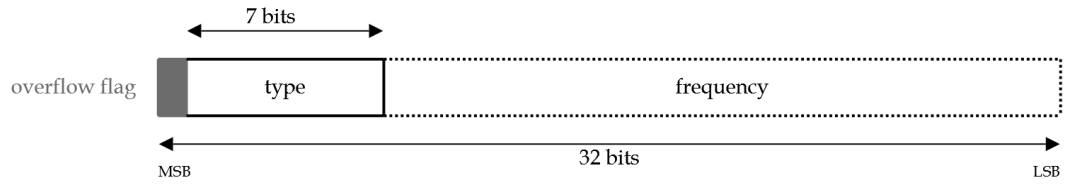
31 To ovšem platí jen za předpokladu, že pravděpodobnost výskytu slova na určitém místě v textu je pro všechny slova stejná.

vystačíme, ale nemůžeme vyloučit, že při zpracování narazíme i N-gramy s větší četností (typicky jim bude hvězdičkový N-gram stupně 0).

Proto třída NGram obsahuje navíc jednu speciální statickou položku — kontejner (typu *map*) pro „přetečené“ četnosti. Pokud při inkrementaci čítače zjistíme, že provedením inkrementace bychom už zasáhli do bitů pro identifikaci typu, uložíme (přičteme) hodnotu čítače do zmíněného kontejneru a čítač nastavíme na hodnotu 1. Kontejner pro uložení hodnot používá typ *size_t*, jehož velikost je na 64-bitové platformě typicky 8 bajtů — maximální hodnota takového čísla je prakticky nedosažitelná.

Pro zjištění frekvence daného N-gramu musíme sečíst aktuální hodnotu čítače s případnou hodnotou uloženou v kontejneru. Abychom nemuseli zbytečně prohledávat kontejner i v případech N-gramů, jejichž čítač nepřetekl, „ukrojíme“ nejvýznamnější bit z bitů vyhrazených pro typ N-gramu a použijeme jej pro uložení příznaku přetečení[32] — viz obrázek 4.

Poznamenejme ještě, že pro mapování instancí N-gramů na hodnoty jejich přetečených čítačů je použit ukazatel na instanci (díky použití *memory poolu* pro uložení instancí je jejich pozice v paměti neměnná).



Obrázek 4: Schéma uložení typu a příznaku přetečení vně paměti pro čítač N-gramu. Velikost paměti dostupné pro uložení frekvence záleží na velikosti typu size_t. Při jejím překročení, je nastaven příznak overflow_flag na jedničku a použit speciální kontejner.

Přetečení čítače u slova v kontextu

V případě slov v kontextu je situace komplikovanější — na konkrétní instanci se nelze odkazovat ukazatelem[33], ani není jednoznačně identifikovatelná pouze vlastními daty (vždy navíc patří nějakému N-gramu). Každá položka struktury pro mapování instancí a jejich přetečených čítačů by proto musela obsahovat mnohem

32 Maximální povolená hodnota parametru N je proto 7 (tolik zbývá bitů pro uložení typu).

33 Při vložení nového prvku do kontejneru typu vector může dojít k realokaci vnitřní paměti a v důsledku toho zneplatnění všech vnějších ukazatelů na uložené instance.

více dat než v případě N-gramů. Při zjišťování kontextu se tak „spokojíme“ s maximální hodnotou čítače o velikosti 2^{32}-1.

Pokud přesto dojde k jejímu přetečení, hodnota čítače zůstane zafixovaná na svém maximu (obdobně v případě N-gramu).

6.3.7 Přístup k N-gramům

Pro rychlý přístup k uloženým instancím N-gramů je použit mechanismus hašování. Implementace programu obsahuje dvě různé verze hašovacích tabulek (resp. hašovacích kontejnerů). Jedna verze je rychlejší, ale paměťově náročnější, druhá je naopak znatelně pomalejší, ale spotřebovává méně paměti. Program je vždy skompilován tak, že používá jen jednu verzi. Pokud je program skompilován s definovaným makrem *_USE_HASHSET* je použitá první (rychlejší) verze, v opačném případě je použitá verze druhá (úspornější).

Verze „hash_set“

Rychlejší a paměťově náročnější verze úložiště N-gramů používá „téměř standardní“ knihovnu *hash_set* [9, kap. 25]. Ačkoli se nejedná o součást STL, má tato knihovna de facto standardní rozhraní — existují pouze určité rozdíly v inicializaci (parametrech šablony). Co se samozřejmě liší znatelněji, jsou konkrétní implementace knihovny.

Pro nás je zejména důležitý způsob, jakým jsou ukládána hašovaná data. Dvě nejpoužívanější verze knihovny[34] ukládají hašovaná data do *spojového seznamu* a to buďto jednosměrného nebo obousměrného (typicky se pro něj používá standardní kontejner typu *list*). Jako hašovací tabulka je použit kontejner typu *vector*, který obsahuje iterátory do spojového seznamu s daty. Iterátory tak rozdělují celý spojový seznam na intervaly, jenž odpovídají jednotlivým řádkům hašovací tabulky (jedná se o tzv. „*kapsy*“ neboli *buckets*).

Každý uzel jednosměrného (resp. obousměrného) spojového seznamu obsahuje kromě samotných dat také jeden (resp. dva) ukazatele na následující uzel (resp. sousední uzly). Ukazatel obsahuje také každý iterátor uložený v hašovací tabulce. Uložená data jsou oproti tomu relativně malá (jedná se o ukazatel na dynamicky

34 Jsou to implementace od SGI (je součástí překladače GCC) a od Dinkumware (je součástí překladače MS Visual Studio).

alokovanou instanci), proto použití kontejneru *hash_set* vyžaduje cca **2-3 krát** více „obslužné“ paměti než je zapotřebí pro samotná data.

Verze „hash_vector“

Paměťově úspornější verze hašovacího kontejneru nepoužívá spojový seznam pro uložení prvků. Všechny hašované prvky, které patří do stejné kapsy, jsou společně uloženy ve vlastním *vectoru*. Složitost operace odstranění prvku z dané kapsy vzroste (je nutné sesunout všechny následující prvky), ale dochází k ní jenom občas (při přehašování). Naopak eliminace spojového seznamu prvků a s ním spojených ukazatelů na sousední prvky se promítne do celkové úspory paměti.

Další výhodou je možnost parametricky ovlivnit chování kontejneru. K tomu slouží následující parametry:

- **-xb** — parametr udává maximální podíl počtu všech uložených prvků a velikosti hašovací tabulky. Zadává se jako mocnina dvojky, tj. parametr 3 znamená, že uvedený podíl bude maximálně 8. Při jeho překročení dojde k zvětšení hašovací tabulky na dvojnásobek a k přehašování prvků. Výchozí hodnotou je 4.
- **-xc** — parametr stanoví maximální podíl **fyzické** velikosti kapsy a počtu prvků v ní uložených. Jeho hodnota se ovšem ověřuje jenom po přehašování prvků. Je-li tento podíl 1, každá kapsa (tj. příslušný kontejner typu *vector*) bude po přehašování oříznuta na minimální nutnou velikost. Hodnota 0 určuje, že se velikosti kapes nemají vůbec kontrolovat. Výchozí hodnotou je 2.

Hašovací funkce

Obě verze hašovacích kontejnerů používají speciální hašovací funkci, která hašuje N-gramy na základě dat jejich složek. Hašovací funkce byla implementována především s důrazem na rychlost.

6.4 Struktura a fáze běhu programu

V programu můžeme rozlišovat dva základní typy entit:

1. **Moduly** — provádí nějakou činnost a jsou většinou spojeny s určitou fází programu. Jejich implementace může být různá — extrakční modul je např. tvořen sadou funkcí, čtecí buffer je implementován jako třída. Blíže se jednotlivým modulům budeme věnovat v další podkapitole.

2. **Objekty** — reprezentují nějaké objekty „reálného světa“ (např. N-gramy) nebo slouží jako jejich úložiště (datový kontejner). Jedinými operacemi, které provádějí, je manipulace s vlastními daty.

Existují ovšem i objekty, které poskytují určitou funkčnost (stejně jako moduly). Dobrým příkladem takového objektu je morfologický filtr — obsahuje sice vlastní data (morfologická pravidla), jeho hlavní úlohou ovšem není uložení těchto dat, ale filtrace N-gramů.

Morfologický filtr ovšem budeme považovat za objekt. Není nijak složité si představit situaci, kdybychom potřebovali těchto filtrů několik[35]. Naopak existence více instancí stejného modulu ve většině případů nemá smysl (pokud nedojde k **výrazné** změně zadání). Tuto vlastnost můžeme považovat za další rozlišující kritérium.

Fáze běhu programu

Běh programu lze rozdělit do tří samostatných fází:

1. Inicializace
2. Zpracování vstupu
3. Zápis výstupu

Každou fázi si znázorníme pomocí diagramu, jenž představuje tok dat a zpráv mezi jednotlivými moduly a objekty. Šipky představují operace: čtení (R), zápis (W) a posílání zpráv (C). Tečkované šipky jsou použity v případě, že daná operace **nemanipuluje** přímo ze zpracovávanými daty, naopak „plné“ šipky představují operaci na těchto datech.

6.4.1 Inicializace

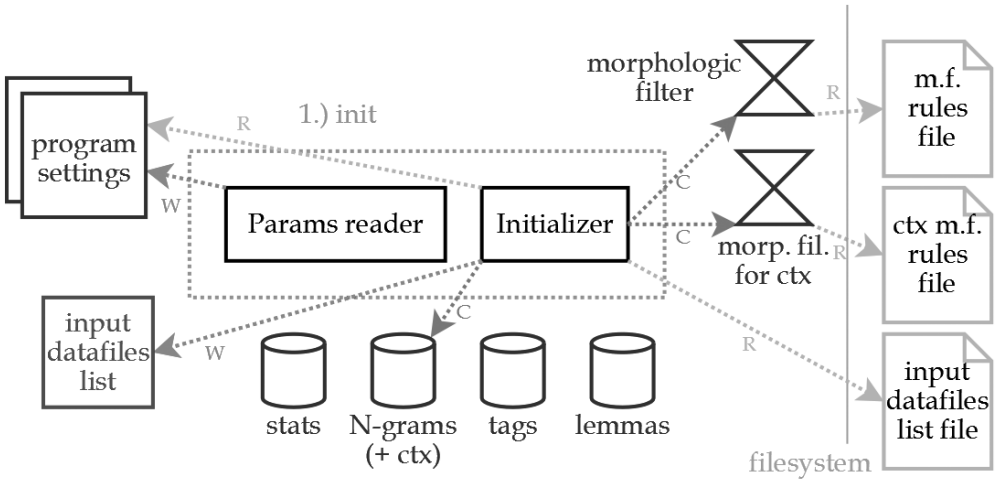
Fáze inicializace zahrnuje start programu, když dochází k:

- Inicializaci globálních proměnných — datových úložišť, filtrů apod.
- Načtení vstupních parametrů, inicializaci nastavení programu a ověření typové a logické správnosti uvedených parametrů.
- Inicializaci statických položek tříd na základě uvedených parametrů (především hodnoty parametru N).

35 Už i v rámci této verze programu jsou morfologické filtry dva: pro N-gramy a pro slova v kontextu.

- Otevření všech specifikovaných výstupních souborů — pro data, kontext a statistiky.
- Načtení seznamu datových souborů ke zpracování a zjištění jejich celkové velikosti.

Inicializační fáze je znázorněná na obrázku č. 5.



Obrázek 5: Diagram představuje fázi inicializace. Parametry programu definují nastavení, kterým se pak při práci řídí inicializační modul. Hlavním úkolem inicializačního modulu je načíst seznam vstupních souborů. Seznam je lokální proměnnou funkce main(), avšak další objekty, které jsou v této fázi inicializovány (tj. filtry a úložiště) jsou globální. Jediným úložištěm, jenž není inicializováno zcela automaticky, je úložiště pro N-gramy.

6.4.2 Zpracování vstupu

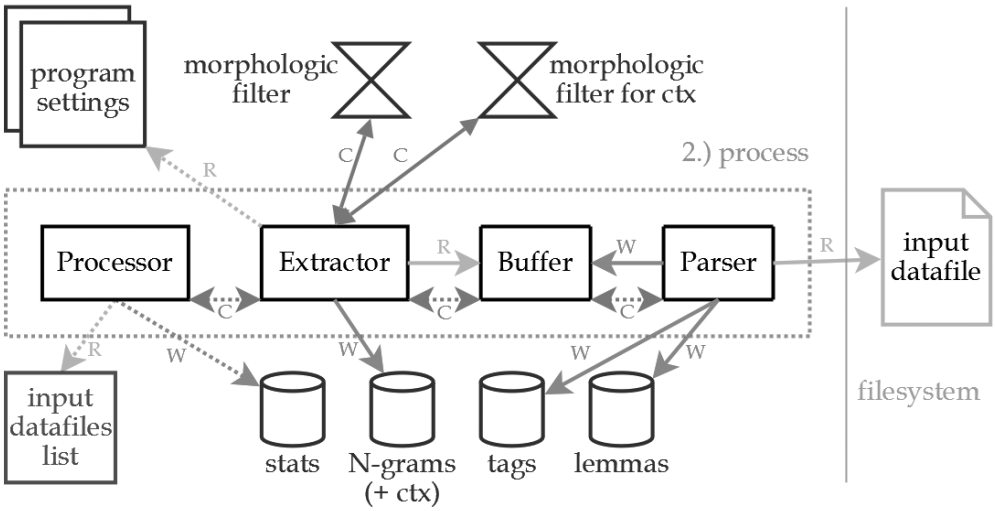
Ve fázi zpracování vstupu jsou postupně zpracovány zadané datové soubory. Pro každý soubor se provedou následující činnosti:

- Extrakce N-gramů (pokud je zapnut morfologický filtr, jsou N-gramy filtrovány).
- Výpočet statistik morfologického filtru (pokud je aktivní).
- Výpočet statistik souborů, tj. počtu slov, vět a N-gramů (pokud je morfologický filtr aktivní, pak se rozlišují N-gramy, jenž (ne)vyhověly filtru).

Kromě toho se volitelně provádí zjištění kontextu (a případná filtrace).

Tato fáze je časově nejnáročnější, proto jsou průběžně vypisovány informace o stávajícím postupu zpracování (a to jak celkovém, tak právě zpracovávaného souboru).

Fáze zpracování vstupu je znázorněná na obrázku č. 6.



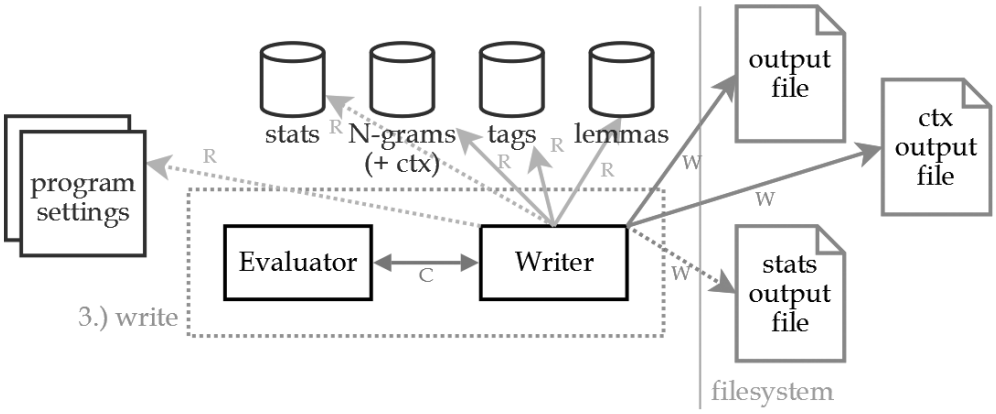
Obrázek 6: Diagram představuje fázi zpracování vstupních dat. Procesní modul načítá jména souborů ze seznamu, otevírá je a postupně předává ke zpracování extrakčnímu modulu. Ten pro každý soubor inicializuje čtecí buffer a parsovací modul. Buffer se řídí požadovanou velikostí kontextového okénka a té přizpůsobuje rychlost načítání dat. Extrakční modul přistupuje k datům v bufferu a z nich extrahuje N-gramy (provádí přitom případnou filtraci). Statistiky slov a vět počítá buffer, extrakční modul k nim přidává statistiky N-gramů a o uložení souhrnných statistik (pro každý soubor) se stará procesní modul.

6.4.3 Generování výstupu

Během poslední fáze programu dochází ke generování výstupu, tj. především se počítají testové statistiky pro všechny (úplné) N-gramy. **Volitelně** se také provádí následující činnosti (pokud jsou specifikovány příslušné parametry):

- Filtrace N-gramů na základě jejich četnosti a testových statistik.
- Vypsání kontextů všech hvězdičkových N-gramů stupně většího než nula.
- Vypsání statistik datových souborů a unikátních úložišť.
- Vypsání statistik morfologického filtru.

Fáze generování výstupu je znázorněná na obrázku č. 7.



Obrázek 7: Diagram představuje fázi generování výstupu. Výstupní modul načítá data ze všech úložišť a pro každý N-gram provádí vyhodnocení testových statistik (pomocí výpočetního modulu). Provádí také případnou filtraci výsledků. Generovaná data pak zapisuje do zadaných výstupních souborů.

6.5 Moduly

Většina modulů již byla stručně zmíněna v předchozí podkapitole. Zde uvedeme velmi zjednodušený popis jejich rozhraní, tj. u metod a funkcí bude uvedeno pouze jméno a k čemu slouží (bez parametrů a návratových typů). Přesné hlavičky lze snadno dohledat v programátorské dokumentaci, jenž je součástí CD přiloženého k práci.

6.5.1 Konfigurační modul

Konfiguraci programu určují globální proměnné a konstanty definované v konfiguračním modulu. Všechny patří do jednoho ze dvou jmenných prostorů:

1. *constants* — konstanty nutné pro běh programu (jsou neměnné).
2. *settings* — proměnné, které určují nastavení programu. Jejich hodnoty závisí na konkrétních parametrech programu. Součástí jmenného prostoru jsou i funkce:
   - *ready()* — ověří, zda jsou uvedené všechny povinné parametry.
   - *correct()* — ověří, zda mají všechny uvedené parametry povolené hodnoty.

6.5.2 Inicializační modul

Všechny úkony, které mají být provedeny ihned po startu programu, jsou součástí inicializačního modulu. Inicializační modul obsahuje dvě funkce:

- *read_and_check_input_datafiles()* — načte seznam vstupních souborů a ověří, že je lze otevřít pro zápis. Navíc spočítá celkovou velikost těchto souborů.
- *startup_init()* — inicializuje všechny statické položky tříd, provádí logickou kontrolu vstupních parametrů (některé parametry mají smysl pouze, pokud jsou definovány jiné) a také volá funkce pro inicializaci filtrů, pokud je nějaká filtrace vyžadována.

6.5.3 Notifikační modul

Notifikační modul není nezbytný pro extrakci N-gramů, proto ani nebyl znázorněn na diagramech v předchozí podkapitole. Úkolem notifikačního modulu je poskytování rozhraní pro komunikaci směrem k uživateli všem modulům, které to vyžadují. Notifikační modul definuje 2 globální objekty, které slouží jako komunikační kanály (oba jsou součástí jmenného prostoru *notifier*):

- *error* — objekt je svázán ze standardním chybovým výstupem (*stderr*) a slouží pro vypisování varování (metoda *warning()*) a chybových hlášení (metoda *fatal_error()*).
- *progress* — objekt je svázán ze standardním výstupem (*stdout*) a slouží pro vypisování informací o průběhu práce programu (metoda *info()*) a času, který uběhl od začátku programu (metoda *elapsed()*).

6.5.4 Parsovací modul

Parsovací modul načítá data ze zadaného vstupního souboru a ukládá je jako slova do čtecího bufferu, avšak vždy po celých větách. Kromě toho také ukládá lemmata a morfologické značky do příslušných úložišť.

Parsovací modul sám zajišťuje logickou kontrolu načtených dat (tj. kontroluje pořadí slov ve větách, délku vět apod.). Případné chybné věty ignoruje a informuje o nich uživatele prostřednictvím várování (zprávy jdou skrz objekt *notifier::error*).

Modul je implementován jako třída. Při inicializaci musí konstruktor jako parametr dostat referenci na otevřený vstupní proud (*ifstream*), jenž reprezentuje

příslušný vstupní datový soubor. Pro každý datový soubor je proto inicializována nová instance objektu.

Práci modulu řídí čtecí buffer, ačkoli modul zapisuje data do čtecího bufferu, ne naopak! Rozhraní je velice jednoduché, tvoří jej 3 metody:

- *eof()* — pro kontrolu, zda byl dosažen konec souboru.
- *parsed()* — vrací informaci o množství dat načtených posledním voláním metody *next()*.
- *next()* — pokusí se načíst požadované množství dat do čtecího bufferu, signalizuje, zda se povedlo načíst alespoň nějaká data (přesné množství lze zjistit pomocí metody parsed).

6.5.5 Čtecí buffer

Čtecí buffer poskytuje jednoduché rozhraní pro (nadřazený) extrakční modul. Po své inicializaci sám zajišťuje, aby obsahoval dostatečné množství dat jak pro extrakci N-gramů, tak pro zjišťování kontextu. Extrakční modul proto parsovací modul pouze inicializuje, jeho práci pak už řídí čtecí buffer.

Slova v čtecím bufferu jsou uložena v kontejneru typu *deque* (jenž je definován jako typ *words_store_t*). Extrakční modul přistupuje přímo k datům v tomto kontejneru (prostřednictvím iterátorů), aby nedocházelo k jejich zbytečnému kopírování.

Pokud jsou k dispozici nějaká data (soubor ještě nebyl dočten do konce), je v každé chvíli jedna věta v čtecím bufferu považována za aktuální.

Rozhraní čtecího bufferu tvoří především metody:

- *next()* — přejdi na následující větu (označ ji jako aktuální).
- *current()* — vrať interval slov jenž identifikuje aktuální větu.

Dále buffer poskytuje dvě různé verze metody *context_range()* — ty slouží pro vrácení intervalu slov ve větném a slovním kontextu.

Interval slov je vždy reprezentován párem iterátorů do kontejneru obsahujícího slova (tj. *words_store_t::const_iterator*) a to iterátorem na první slovo patřící do intervalu a iterátorem na první slovo, které již do intervalu nepatří. Tyto

iterátory jsou platné pouze do doby dalšího volání metody *next()*[36] (což ovšem plně postačí).

6.5.6 Extrakční modul

Extrakční modul je srdcem celého programu, extrahuje N-gramy z načtených vět a ukládá jejich výskyty do speciálního úložiště. Navíc, pokud je to požadováno, ukládá také větný kontext (o identifikaci intervalu slov, jenž patří do kontextového okénka se ovšem stará čtecí buffer).

Rozhraní extrakčního modulu je velice jednoduché — tvoří jej funkce *extract()*, která jako parametr očekává referenci na vstupní datový proud (*ifstream*) a referenci na čítač statistik pro daný soubor. Interně je modul implementován prostřednictvím několika lokálních funkcí a využívá také služeb modulu pro extrakci souvislých podstromů velikosti N, jenž implementuje algoritmus popsaný v 5. kapitole.

6.5.7 Procesní modul

Procesní modul je vstupní branou do druhé fáze programu. Jeho rozhraní je podobně jednoduché jak v případě extrakčního modulu, také jej tvoří pouze jediná funkce: *process()*. Procesní modul postupně prochází zadaným seznamem vstupních souborů, každý soubor otevře a předá ke zpracování extrakčnímu modulu. Průběžně také vypisuje informaci o tom, kolik procent vstupu již bylo zpracováno a zajišťuje uložení četnostních statistik každého souboru a statistik morfologického filtru (pokud je filtr aktivní).

6.5.8 Persistentní objekty

Program při své práci používá několik objektů, ke kterým přistupuje většina modulů a které jsou vyžadovány od začátku programu do jeho konce. Tyto objekty jsou proto definovány jako globální a pro lepší čitelnost programu jsou všechny součástí jednoho „modulu“ (ačkoli zde se nejedná o modul ve smyslu naši definice).

Persistentní objekty jsou zapouzdřeny ve jmenném prostoru *persistent*. Patří k nim:

- úložiště pro lemmata a úložiště pro morfologické značky

36 Volání metody next() může skončit voláním metod push_back() a pop_front() na kontejneru pro slova a to v případě kontejneru typu deque zneplatní veškeré iterátory [13, kap. 6.3].

- úložiště pro N-gramy
- morfologický filtr a (morfologický) filtr pro slova v kontextu
- čítač statistik datových souborů a čítač statistik morfologického filtru

Některé z uvedených objektů nejsou vždy vyžadovány (např. filtr pro slova v kontextu), ale i přesto jsou definovány globálně — paměť nutná pro jejich existenci je zanedbatelná ve srovnání s celkovými nároky programu.

6.5.9 Výpočetní modul

Úkolem výpočetního modulu je vypočítat hodnoty polí kontigenční tabulky na základě spočtených frekvencí hvězdičkových N-gramů a to jak hodnoty zjištěné, tak i očekáváné. Algoritmus výpočtu již byl popsán dříve — v podkapitolách 5.4 a 5.5.

Výpočetní modul také definuje sadu funkcí pro výpočet všech testových statistik. Ty jsou implementovány na základě vzorců uvedených v kapitole 3.

6.5.10 Výstupní modul

Výstupní modul má především za úkol vypsat výsledky extrakce N-gramů do zadaných výstupních souborů. Výpočet testových statistik provádí výpočetní modul, úkolem výstupního modulu je tak (navíc) pouze provedení případné filtrace vypisovaných výsledků.

Výstupní modul také, pokud je to požadováno, vypisuje kontexty N-gramů a všechny dostupné statistiky.

Výstupní modul je tvořen sadou funkcí:

- *write()* — vypisuje výsledky extrakce N-gramů
- *write_contexts()* — vypisuje kontexty N-gramů
- *write_stats()* — vypisuje statistiky datových souborů
- *write_morphologic_filter_stats()* — vypisuje statistiky morfologického filtru

6.6 Externí knihovny

Kromě už zmíněné knihovny STL a „téměř standardní“ hašovací knihovny, jenž jsou stabilními součástmi použitých překladačů, program využívá také knihovnu

Boost. Knihovna Boost je volně dostupnou sadou různých knihoven, které implementují užitečné funkce, jenž v knihovně STL (zatím) chybí [15].

Program využívá knihovnu *boost::progress* pro zobrazení času, jenž uplynul od spuštění programu a pro vykreslování stavového řádku informujicího o postupu složitějších operací. Nejedná se o zásadní funkce programu, proto lze program skompilovat do funkční podoby i v případě, že knihovna Boost není k dispozici.

7 Možná rozšíření

Tato kapitola obsahuje přehled možností rozšíření a zdokonalení funkcí programu. Některé z těchto možností se objevily už během psaní programu, ale nebyly tak důležité, aby se staly jeho součástí, další jej napadly až během testování, kdy už na zásadní změny nezbýval čas.

7.1 Grafické rozhraní

Grafické rozhraní by mohlo především usnadnit zadávání parametrů, ale případně by mohlo poskytovat i důkladnější informace o průběhu extrakce (průběžné statistiky, stav použité paměti apod.)

Implementace grafického rozhraní by ovšem výrazně skomplikovala snadnou přenositelnost programu.

7.2 Obecnější vstupní formát

Program je implementován na míru formátu dat specifikovaném v 4 kapitole. Modularizace programu dovoluje snadné přizpůsobení programu pro jiný vstupní formát za předpokladu, že daný formát obsahuje „extrahované" položky slova: lemma, morfologickou značku, index řídícího slova a typ závislosti. V takovém případě postačí vhodně upravit parsovací modul, zásah do jiných modulů není třeba.

Pokud by cílem změn mělo být přizpůsobení programu vstupnímu formátu, jenž neobsahuje syntaktickou analýzu vět a v důsledku toho neumožňuje extrakci N-gramů ze závislostních stromů, pak by změny vyžadovala i část extrakčního modulu, jenž provádí identifikaci N-gramů a některé položky N-gramu by se staly irelevantní (typ závislosti, index řídícího slova)[37].

7.3 Lepší ošetření fatálních chyb

Výskyt chyby způsobí okamžité ukončení programu a to i v případě, že k chybě dojde až během fáze extrakce. Toto řešení je velmi jednoduché, na druhou stranu

37 Zdrojové kódy programu už jsou pro tuto variantu přizpůsobeny. Důvodem byla nutnost program otestovat na obsáhlých datech. K dispozici však byla pouze data bez syntaktické analýzy vět, proto byla funkčnost programu vhodným způsobem rozšířena (viz příloha B).

má jednu citelnou nevýhodu. Veškeré provedené mezivypočty jsou při předčasném ukončení programu nenávratně ztraceny. Proto by vhodným rozšířením byl mechanismus, který by do zadaného *„dump“ souboru* uložil všechny mezivýpočty spolu se stavem programu (nastavení parametrů, pozici v aktuálně zpracovávaném vstupním souboru apod.).

Do „dump“ souborů by musely být uložena následující data:

- řetězce a jejich číselné indexy
- hvězdičkové N-gramy (nejlépe každý typ do vlastního souboru)
- slova v kontextu pro jednotlivé hvězdičkové N-gramy (opět nejlépe každý typ do vlastního souboru)
- nastavení programu (včetně pravidel filtrů)
- aktuální stav extrakce — jméno aktualního souboru a pozice poslední zpracované věty (resp. N-gramu)

Zejména poslední položka je problematická. Program by totiž musel zrušit všechny změny v paměti ovlivněné zpracováním aktuální věty (resp. N-gramu), pokud toto zpracování nedoběhlo do konce.

Při zotavení by pak muselo dojít k reinicializaci nastavení, k znovuvytvoření indexů vně řetězcových úložišť, k rozhašování N-gramů (a případně i k uložení jejich kontextu) a k obnovení aktuálního stavu extrakce (tj. inicializaci parseru, čtecího bufferu a extrakčního modulu ve stavu, ve kterém došlo k přerušení).

7.4 Dvoufázové zjišťování kontextu

Při zjišťování kontextu se kontext ukládá i pro ty N-gramy, které se při vyhodnocení výsledků ukážou jako nezajímavé z důvodu příliš nízké frekvence výskytu. Frekvence výskytů ale samozřejmě neznáme předem, proto během fáze extrakce musíme ukládat všechny nalezené (neodfiltrované) N-gramy. Proces zjišťování kontextu ovšem můžeme provést zvlášť až v dalším průběhu programu a před jeho startem můžeme odfiltrovat N-gramy s příliš nízkou frekvencí výskytu.

8 Závěr

Cílem této práce byla efektivní implementace metod pro extrakci kolokací ze syntakticky anotovaných korpusů. Hlavním cílem byla především optimalizace využití operační paměti, přesto jsme vždy zkoumali i složitost časovou a snažili jsme se najít rozumný kompromis.

V úvodu práce jsme si uvedli vlastnosti kolokací, jenž jsou klíčové pro návrh programu a popsali implementované metody. Poté jsme podrobně popsali funkce programu a provedli jsme analýzu jeho chování z hlediska paměťové náročnosti. Díky tomu jsme zjistili, které funkce (resp. části) programu má smysl optimalizovat.

Ukázali jsme si, že možnosti jazyka C++ pro efektivní práci s operační pamětí jsou velmi bohaté. Kromě obecně známých a uznávaných programátorských technik, jako např. hašování nebo „*memory pooling*“, jsme využili i některá nestandardní řešení, které nám umožňovala specifikace zadání a to především při řešení způsobu reprezentace N-gramů v paměti.

Důležitou částí práce byl také návrh efektivního algoritmu pro extrakci souvislých podstromů velikosti N ze závislostního stromu věty.

Ačkoli byla práce zaměřená na extrakci závislostních N-gramů z Pražského závislostního korpusu, bylo naší snahou navrhnout strukturu programu tak, aby případné rozšíření jeho funkčnosti nevyžadovalo velké zásahy do zdrojového kódu. Proto jsme v práci věnovali poměrně dost místa pro popis rozdělení programu do modulů a také pro popis jejich vzájemné komunikace.

V předposlední kapitole jsme si ukázali, že existují ještě další způsoby zdokonalení programu a to jak po stránce uživatelské, tak i po stránce výkonnostní. Zejména funkce zjišťování kontextu je stále místem, které lze více optimalizovat.

Povaha chování programu je ovšem do značné míry nedeterministická a přesná spotřeba paměti a času závisí ve velké míře na konkrétních vstupních datech a na zadaných parametrech. Dalším možným (a zajímavým) rozšířením by tak mohlo být přidání samoučicího mechanismu, který by upravoval chování programu (zejména množství paměti alokované předem) na základě dosud načtených dat.

Literatura

[1] Pecina P., Holub M. (2002): *Sémanticky signifikantní kolokace*. ÚFAL/CKL Technical Report TR–2002–13

[2] Anděl J. (1998): *Statistické metody*. Matfyzpress

[3] Pecina P., Schlesinger P. (2006): *Combining Association Measures for Collocation Extraction*. Proceedings of the 21th International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING/ACL 2006)

[4] Church Kenneth W., Mercer Robert L. (1993): *Introduction to the special issue on computational linguistics using large corpora*. Computational Linguistics 20

[5] Manning C. D., Schűtze H. (1999): *Foundations of Statistical Natural Language Processing*. MIT Press

[6] Mood Alexander M., Franklin A. Graybill, Duane C. Boes (1974): *Introduction to the theory of statistics*. New York: McGraw Hill

[7] Justenson John S., Katz Slava M. (1995): *Technical terminology: some lingustic propertiesand an algorithm for identification in text*. Natural Language Engineering 1

[8] *Pražský závislostní korpus 2.0*. http://ufal.mff.cuni.cz/pdt2.0/index-cz.html

[9] Meyers S. (2001): *Effective STL*. Addison-Wesley Professional

[10] Stroustrup B. (1997): *The C++ Programming Language (3rd Edition)*. Addison Wesley Professional

[11] Meyers S. (1997): *Effective C++ (2nd Edition)*. Addison-Wesley Professional

[12] *Čeština*. http://cs.wikipedia.org/wiki/%C4%8Ce%C5%A1tina

[13] Josuttis Nicolai M. (1999): *The C++ Standard Library, A Tutorial and Reference*. Addison-Wesley Professional

[14] Meyers S. (1996): *More Effective C++*. Addison-Wesley Professional

[15] *Boost C++ Libraries*. http://www.boost.org/

[16] *Korpus SYN2005*. http://ucnk.ff.cuni.cz/syn2005.html

Přílohy

A Příklad obsahu datového souboru PDT 2.0

Pozn.: Testovací data, z nichž pochází příklad, obsahují pouze 4-znakovou morfologickou značku.

```
1     Zřejmě      zřejmě      D-1A  6     Adv
2     si    se    P---  6     Aux
3     v     v-1   R---  6     Aux
4     té    ten   PF--  5     Atr
5     organizaci  organizace  NF-A  3     Adv
6     cenili      cenit VM-A  0     Pred
7     času  čas   NI-A  6     Obj
8     pracovníků  pracovník   NM-A  7     Atr
9     více  hodně D-2A  6     Adv
10    ,     ,     Z---  11    Aux
11    než   než-2 J---  9     Aux
12    je    být   V--A  11    Adv
13    u     u-1   R---  12    Aux
14    nás   já    P---  13    Adv
15    zatím zatím D---  12    Adv
16    zvykem      zvyk  NI-A  12    Pnom
17    .     .     Z---  0     Aux
```

Závislostní strom uvedené věty je znázorněn na obrázku 1 (kapitola 3).

B Příklad pravidel morfologického filtru

V tabulce 7 jsou uvedená pravidla slovnědruhového filtru navržená pro hledání kolokací v českém jazyce [1, kap. 5.6].

A N	D A	N C
N N	D V	D D
N A	V D	D N
V N	A A	A V
N V	C N	C C
V V	N D	A D

Tabulka 7: Pravidla slovnědruhového filtru pro hledání kolokací v českém jazyce (N — podst. jméno, A — přídavné jméno, V — sloveso, C — číslovka, D — příslovce).

C Výsledky testů časové a paměťové náročnosti programu

Testy byly provedeny na výpočetním stroji s operačním systémem Linux Fedora. Stroj disponoval operační pamětí o velikosti 16 GB a dvěma procesory AMD Opteron Dual Core 2 Ghz. Testována byla 64-bitová verze programu. Paměťová a časová náročnost byla měřená pomocí utility *top*.

Jako testovací data posloužil textový korpus SYN2005, jenž je součástí Českého národního korpusu [16]. Součástí korpusu SYN2005 není syntaktická analýza vět, proto pro účely testování musela být funkčnost programu rozšířena o možnost extrakce *„povrchových N-gramů“*[38]. Tento způsob extrakce vyžaduje navíc parametr pro určení velikosti kolokačního okénka (**-w**). Přesný princip extrakce i definice kolokačního okénka lze nalézt např. v práci Pavla Peciny a Martina Holuba [1, kap. 4.4]. Nejčastěji použitá kombinace parametrů „**-n 2 -w 1**“ znamená, že program extrahoval bigramy, jenž tvořily sousední slova (jedné věty).

Statistiky korpusu SYN2005

Nejdříve si uveďme statistiky korpusu SYN2005, tak jak je spočítal program:

- 122 551 735 slovních jednotek (včetně interpunkce)
- 8 078 351 vět
- 805 111 unikátních tvarů (lemmat)

Lemmata v rámci korpusu SYN2005 mají několik rozlišujících informací navíc, jako relevantní však byla brána pouze informace o významu slova.

Parametry testů

Testy programu rozdělíme do dvou skupin. V první skupině budou testy, jenž se zaměřují na různé parametry **extrakce N-gramů**. V druhé skupině budou testy, jenž se zaměřují na různé parametry **zjišťování kontextu**.

38 Rozšíření o „povrchovou extrakci“ N-gramů bylo vyřešeno podobně jako v případě hašovacích kontejnerů, tj. program je vždy na základě (ne)existence speciálního makra skompilován jen pro jednu verzi.

U všech testů extrahována morfologická značka obsahovala pozice pro slovní druh, jmenný rod, stupeň a negaci. Pokud byl použit morfologický filtr (parametr **-f**), pak obsahoval pravidla uvedená v příloze B.

Testy extrakce N-gramů

V rámci první skupiny testů bylo provedeno také srovnání dvou verzí programu s různou implementací hašovacích tabulek pro uložení N-gramů[39]: verze používající *hash_set* (HS) a verze používající *hash_vector* (HV) s parametry -xb 4 a -xc 2.

Výsledky testů prezentuje tabulka 8. Srovnání výsledků testu 1 a testu 4 názorně dokazuje, že velikost použité paměti závisí na počtu unikátních instancí, ne na počtu extrahovaných N-gramů (svou roli ovšem také hraje velikost parametru N).

Číslo testu	**1.**	**2.**	**3.**	**4.**
Parametry extrakce	-n 2 -w 1	-n 2 -w 1 -f	-n 2 -w 3 -f	-n 3 -w 2
Spotřeba paměti (HS)	1 261 MB	964 MB	2 381 MB	7 415 MB
Spotřeba paměti (HV)	760 MB	491 MB	-	-
Spotřeba času (HS)	12,1 min	10,3 min	18,1 min	28,1 min
Spotřeba času (HV)	916,5 min	188,8 min	-	-
# extrah. N-gramů	114 473 384	114 473 384	298 220 541	106 580 860
# odfiltrovaných N-gramů	0	78 024 155	200 759 004	0
# vyhovujících N-gramů	114 473 384	36 449 229	97 461 537	106 580 860
# unikát. úplných N-gramů	21 098 032	14 088 963	43 095 915	60 003 340

Tabulka 8: Výsledky testů časové a paměťové náročnosti programu pro různé parametry extrakce N-gramů a různé verze programu z hlediska ukládání N-gramů.

Testy zjišťování kontextu

Testy zjišťování kontextu byly provedeny pouze pro „*hash_set*“ verzi programu se stejnými parametry extrakce N-gramů jako v případě testu číslo 2 z předchozí tabulky (tj. byly extrahovány pouze bigramy tvořené sousedícími slovy a byla zapnuta slovnědruhová filtrace). Při všech testech byl navíc použit filtr pro slova v širokém kontextu, takže do širokého kontextu se započítavaly pouze tzv. „*open class words*“: podst. jména, přídavná jména, slovesa a příslovce.

39 Viz podkapitola 6.3.7.

Z výsledků lze vypozorovat, že i přes použité filtry je funkce zjišťování kontextu paměťově velmi náročná. Pro větné kontextové okénko o poloměru 1 se ani nepovedlo v dostupné paměti zpracovat celý korpus.

Výsledky testů prezentuje tabulka 9.

Parametry	Zpracováno	Spotř. paměti	Spotř. času
-cs 0	100 %	12,3 GB	458 min
-cs 1	50 %	10,2 GB	345 min

Tabulka 9: Výsledky testů časové a paměťové náročnosti programu pro různé parametry zjišťování kontextu extrahovaných N-gramů.

D Obsah CD

CD přiložené k práci obsahuje:

1. Zdrojové kódy programu včetně makefile — adresář *src*
2. Skompilovaný program (verze pro Windows i Linux) — adresář *bin*
3. Programátorskou dokumentaci (v anglickém jazyce) ve formátu HTML — adresář *docs*
4. Tuto práci ve formátu PDF — adresář *thesis*
5. Testovací data (vzorek korpusu PDT 2.0) — adresář *data*