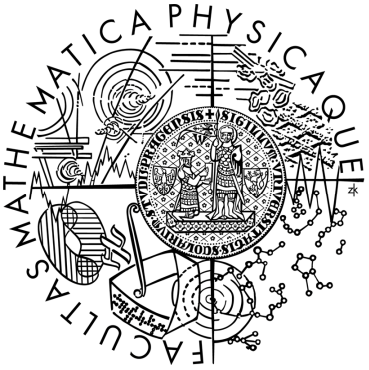
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Vratislav Dřínek

Automatické úpravy kontextu v textových polích

Katedra aplikované lingvistiky

Vedoucí bakalářské práce: RNDr. Ondřej Bojar

Studijní program: Informatika

Studijní obor: Programování

Praha 2016

Rád bych poděkoval vedoucímu bakalářské práce RNDr. Ondřejovi Bojarovi Ph.D. za cenné rady, vstřícnost a trpělivost při psaní této práce.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.
Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne …… Podpis autora

Název práce: Automatické úpravy kontextu v textových polích

Autor: Vratislav Dřínek

Ústav: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: RNDr. Ondřej Bojar Ph.D., Ústav formální a aplikované Lingvistiky

Abstrakt: Tématem této bakalářské práce je pomocník při úpravě textu. Program se snaží předvídat uživatelovy záměry a usnadnit mu jejich dokončení. Úloha, kterou se zabývá tato práce je zcela nová a funkcionalita není nejen v běžných textových editorech k dispozici. Téma práce bylo inspirováno uživatelským rozhraním programovacího prostředí Visual Studio, které občas samo navrhuje programový kód, který programátor s nejvyšší pravděpodobností hodlá napsat. Pro svou práci pomocník používá morfologickou analýzu vět pomocí již existujícího nástroje Morphodita.

Klíčová slova: větný rozbor, český slovník, Morphodita tagger, autocomplete

Title: Automatic Modifications of Context in Text Fields

Author: Vratislav Dřínek

Department: Institute of formal and applied Linguistics

Supervisor: RNDr. Ondřej Bojar Ph.D., Institute of formal and applied Linguistics

Abstract: The topic of this bachelor thesis is text editor assistant. The program tries to anticipate user's intentions and propose their quick accomplishion. The aim of this thesis solves is completely new and its function is not available even in advanced text editors. The topic is inspired by the user's interface of the programming enviroment Visual Studio, which sometimes proposes the program code the user is probably going to write. The assistant uses morphological analyses provided by Morphodita to analyse sentences morpologically.
Keywords: sentence analysis, Czech dictionary, Morphodita tagger, autocomplete

Obsah práce

1. Analýza Úlohy

Práce se zabývá možnostmi, jak uživateli usnadnit upravování již napsaného textu v textovém editoru. Tento koncept chování je inspirován programovacím prostředím Visual Studio, které nabízí uživateli návrhy kódu, který pravděpodobně hodlá napsat, jen s tím rozdílem, že v našem případě se jedná o běžný text, nikoli o zdrojový kód s formální gramatikou.

Při úpravách textu se často stává, že uživatel potřebuje přeformulovat nějakou větu nebo v nějaké větě vyměnit slovo. Přitom se zpravidla nemění jen jedno slovo, ale v důsledku této úpravy musí být změněno i několik slov okolo, přičemž v jazycích s bohatým tvaroslovím, jako je čeština, je taková nutnost ještě pravděpodobnější. Všeobecně se na úpravu věty lze dívat tak, že máme nějakou původní a cílovou větu (viz níže příklad v rámečku). Jak je vidět, existuje více způsobů, jak se „dopracovat“ od jedné verze ke druhé. Zde je uveden jen jeden příklad, ale 3 elementární úpravy slov mohly být udělány celkem šesti možnostmi (3!). V jistém ohledu lze říci, že existují jakési mezifáze mezi původní a přeformulovanou větou. Cílem editoru je právě snažit se dedukovat cílovou větu, pokud už má k dispozici její mezifázi a dokončení naznačené úpravy nabídnout uživateli s cílem usnadnit mu práci.

Na okně sedí malá a bílá kočka.
Na okně sedí malá a bílá pejsek.
Na okně sedí malá a bílý pejsek
Na okně sedí malý a bílý pejsek.

Editor průběžně sleduje práci uživatele s textem, a pokud rozpozná nějaký z předem definovaných typů úprav (tzv. úpravy s podporou), nabídne její dokončení.

Při řešení úlohy dojde v prvé fázi k základní klasifikaci úpravy, tedy o určení, o který druh úpravy s podporou se jedná, případně o určení, že úprava podporu nemá. Nejlépe je to vidět na schématu metod. (přílohový obrázek na konci práce)

Uvádíme příklady úprav, jejich analýzy a řešení, v závorkách uvádíme text, který uživatel právě připsal nebo změnil.

„(na) bílá pláň“ -> „na bílou pláň“; „na bílé pláni“

Při vepsání předložky se počítá, že jde jen o částečnou změnu, ve které hodlá uživatel dále pokračovat, a počítá se s tím, že nabídnutá věta stále není cílová, ale jen pokročilejší mezifáze. Jedná se tedy jen o menší pomoc uživateli ve psaní.

„Nemám rád (vytírat –> vytírání) podlahu.“ -> „Nemám rád vytírání podlahy.“
Zde se uživateli zjevně nelíbilo sloveso „vytírat“, proto ho přepsal na podstatné jméno. Pokud program tuto situaci rozpozná, nabídne změnu čtvrtého pádu původního předmětu slovesa na druhý pád.

„Na okně sedí malá a bílá (kočka –> pejsek).“ –> „Na okně sedí **malý a bílý** pejsek.“
„Maminka připravuje bramborový (salát –> kaši).“ –> „Maminka připravuje bramborovou kaši.“
„Hodný tatínek natírá dřevěný (plot -> ohradu).“ –> „Hodný tatínek natírá dřevěnou ohradu.“
Zde se předpokládá, že uživatel (s ohledem na jednoduchost a témata vět se může jednat například o tvůrce slabikáře) nejprve upraví podstatné jméno. „Mezifází“ je tedy například výraz „Na okně sedí malá a bílá **pejsek**.“. Takto větu samozřejmě uživatel nechat nechtěl, předpokládá se, že chce změnit i tvary přídavné jmenné skupiny před dotyčným podstatným jménem a editor v tomto případě navrhne uživateli změnu tvarů přídavných jmen. Tento typ úpravy je prováděn na základě implicitního slovníku v zakomponované Morphoditě. Počítá se s tím, že jmenná skupina může obsahovat interpunkci (,) a spojky (a, i, …).

„Hodný tatínek (natírá -> natírán) dřevěný plot.“ -> „Dřevěný plot je natírán hodným tatínkem.“; „Hodný tatínek je natírán dřevěným plotem.“
V tomto případě jde o změnu aktiva na pasivum. Pomocník reaguje pouze tehdy, pokud uživatel načne změnu přepsáním slovesa (aktivum <–> pasivum). Zde se předpokládá, že je pravděpodobně nutné prohodit podmět a přísudek a také změnit pád přísudku. I v méně pravděpodobném a zde sémanticky nesmyslném, ale gramaticky korektním případě, kdy uživatel chce změnit směr děje, je nutno změnit pády. V obou těchto případech je mezifází výraz „Hodný tatínek natírán dřevěný plot.“.

„(Když –> *smazáno*) prší, tak je všechno mokré.“ –> „Pršení způsobí, že je všechno mokré.“, „Pršák způsobí, že je všechno mokré.“,
Zde se jedná o ukázku pravidla navrhování jedné z možných rozsáhlých reformulací. Možností, jak větu přeformulovat je samozřejmě mnohem. Mezifází je zde věta se smazaným počátečním slovem „když“. Tento typ změny používá explicitní slovník příbuzných slov. Nutné zde je, aby sloveso za smazaným „když“ bylo bez podnětu a přísudku. Editor se podívá do slovníku příbuzných slov, a pokud najde jedno nebo několik příbuzných podstatných jmen, nabídne pro každé z nich novou frázi tvaru „X způsobí, že Y“.

„Postihla nás velká (povodeň –> *smazáno*)“ –> „Postihla nás velká záplava.“
Zde uživatel pouze smazal slovo. Předpokládá se, že ho chce nahradit jiným, proto se zde použije slovník synonym a nabídne se uživateli náhrada za slovníkové synonymum a to v nejvíce pasujícím tvaru. Mezifází je zde věta se smazaným slovem.

Algoritmickým jádrem úlohy je metoda, která jako vstup dostane jako jeden parametr českou větu (předpokládá se, že samu o sobě správnou) a jako druhý parametr větu s načatou, ale nedokončenou úpravou. Tato metoda je implementována v podprojektu EditorAssistantCore, což je back-endová část programu.
Než je zavolána hlavní metoda, editor nejprve identifikuje změnu, zjistí řádek (odstavec), ve kterém byla provedena změna, řádek poté tokenizuje a zjistí větu, ve které byla provedena změna. Následně zavolá hlavní metodu a z výsledků (věty) poté opět sestaví řádky a věty dá do seznamu návrhů. Způsob zapojení do editoru je poměrně jednoduchý, ale dá se předpokládat, že výhledově by mohl být srovnatelně kvalitní, jako v již zmíněném Visual Studiu návrhy textu kódu. Front-end programu je již navržen tak, aby navrhování mohlo fungovat na tzv. silném a slabém doporučení. Tedy tak, že v případě silného doporučení se navrhovaná úprava přímo vepíše do textového pole a pokud s ní uživatel nesouhlasí, musí se návrhu aktivně bránit nebo si aktivně vybrat jiný alternativní. Pokud je doporučení slabé, jeden nebo více návrhů se pouze vypíše do výběru mimo textové pole a do textového pole se vepíše až po aktivním uživatelově souhlase. Je-li v nabídce více návrhů a uživatel žádný neoznačí, bere se automaticky ten první z nabídky. Nicméně v současné verzi

se používá pouze koncept slabého navrhování, ačkoliv je implementovaná i podpora silného navrhování.

Dále, až je zavolána hlavní metoda, tak ve větách, které dostane, lokalizuje rozsah, ve kterém se obě věty liší (mají prakticky vždy nějaký společný začátek nebo konec). Na nezměněnou verzi věty zavolá větný tagger Morphodity, na změněnou verzi tagger nevolá, protože se předpokládá, že věta je mezifází změny, tedy z hlediska pravopisu chybná. Podle tagů následně určí, o jaký druh úpravy jde.

Pokud je vepsána předložka, zjistí se s jakými pády se dotyčná předložka může pojit (pomocí výčtu z Morphodity). Následně se zjistí rozsah následující jmenné skupiny a poté se nabídnou návrhy změn tvarů jmen. Pokud za předložkou nenásleduje jmenná skupina, nic se nedělá.

Editor je schopen poznat změnu slovesného aktiva na pasivum a obráceně, poznat dokáže i některé případy zvratných pasiv. Pokud je změněno aktivum na pasivum, předpokládá se, že v původní větě je konatel jako podmět v prvním pádě a předmět ve čtvrtém (jmenné skupiny). Pokud se mění pasivum na aktivum, předpokládá se, že konatel je v sedmém pádě. Pokud se tyto jmenné skupiny najdou, při podání návrhů následuje změna pádů a i prohození podmětu s přísudkem. Počítá se s tím, že pisatel spíše chtěl zachovat význam, ale možná chtěl změnit směr děje (viz uživatelská dokumentace).

Editor umí poznat také změnu podstatného jména na sloveso nebo obráceně. Zde se předpokládá, že za slovesem je jmenná skupina ve čtvrtém pádě a za (dějovým) podstatným jménem v pádě druhém. Při změně se tedy předpokládá, že slovo se stále vztahuje k té samé jmenné skupině a úkolem pomocníka je upravit pád jmenné skupiny.

Pokud je změněn tvar (nebo celé podstatné jméno – tedy potenciálně rod) za přídavnou jmennou skupinou, metoda určí rozsah jmenné skupiny a počítá s tím, že může obsahovat i spojky (a, i, ani, nebo) a čárky (interpunkce) a poté navrhne změnu tvarů těchto slov. Návrhy nemusí být vždy jednoznačné, pokud nová forma podstatného jména může mít potenciálně více pádů.

Pokud jde o smazání části textu, zabýváme se jenom případem, kdy je smazáno jen jedno slovo. Zvlášť se řeší specifická situace, kdy je smazané slovo „když“ na začátku věty a v tomto případě se navrhne změna slovesa na (příbuzné) podstatné jméno podle slovníku příbuzných slov. (Příklad: Když prší, tak je všechno mokré -> Pršení/Pršák způsobí, že je všechno mokré.). V ostatních případech smazání jednoho

slova se navrhne jeho případné synonymum ze slovníku příbuzných slov a to v nejpodobnějším tvaru.

2. Interpretace chování uživatele

Důležitou částí úlohy je také vyhodnocení chování uživatele. Uživatel může například smazat slovo a následně ho nahradit jiným, může při psaní slova na chvíli „usnout“ a po nějaké době dopsat jeho zbytek, a další. Základní signály o chování uživatele v editoru vznikají při pohybu kurzoru v textovém poli a změnách textu v textovém poli. Pomocí programátorských nástrojů lze docílit získání signálů s požadovaným zpožděním a také signálů sdělující například: Teď došlo k nečinnosti uživatele po dobu X.

V tomto editoru je interpretace nastavena tak, že dokud uživatel provádí změny bez prodlev delších než stanovený časový limit, předpokládáme, že úpravy jsou součástí jedné změny. Výchozí časový interval je 1500 ms. Taktéž má uživatel možnost zvolit, zda pohyb kurzoru má být vnímán jako činnost, od které se znovu odpočítává čas. Pokud tedy uživatel smaže slovo a po delší době než zvolený časový interval napíše jiné, je chování vyhodnoceno jako dvě změny a pro každou z nich se editor pokusí vypočítat návrhy úprav dokončení úprav. Pokud ale nové slovo napíše hned po smazání původního, je jeho činnost vyhodnocena jako jedna změna a nové návrhy jsou počítány na základě původní věty lišící se v celém změněném slově.

Problém taktéž může nastat, pokud uživatel přeruší psaní slova v průběhu jeho psaní a začne znovu psát až po uplynutí kritického časového intervalu. Editor se v takovém případě pokusí spočítat návrhy na základě nové věty obsahující na nějakém místě chybně půlku slova, které nejspíše ve slovníku nebude pokryto a tento výpočet skončí pochopitelně s prázdným výsledkem. Po dopsání slova se naopak jako výchozí věta bude brát věta s půlkou slova, které bude taggerem bráno jako neznámý slovní druh a výpočet též skončí s prázdným výsledkem. Nicméně takovéto případy by neměly nastat, pokud si uživatel zvolí dostatečně dlouhý časový interval a umí dostatečně rychle psát. Tento koncept není ideální, ale vzhledem k tomu, že program je pouze experimentálním dílem, tato implementace stačí.

3. Použité nástroje

Tato kapitola popisuje nástroje, které jsou použity jako součást úlohy. Popsány jsou zde nástroje přímo související s Morphoditou, tak i generické nástroje řešení všeobecně.

3.1 Morphodita

Morphodita je slovník a také nástroj, který provádí základní tvaroslovný rozbor. Jako slovník obsahuje slovní lemmata a ke každému lemmatu všechny jeho tvary. Slovní lemma je název přiřazený danému slovu a jeho tvarům. Vždy jde o základní tvar slova, který je někdy spojený se zpřesňujícími značkami.

Při tvaroslovném rozboru k větě ke každému tokenu Morphodita přiřadí lemma a morfologickou značku. Tento rozbor není vždy přesný a je založen na statistice – Morphodita dostala jakási vzorová data (velkou množinu ručně značkovaných českých vět), ze kterých získala pravidla typu „za slovesem ve tvaru X je s pravděpodobností Y jméno ve tvaru Z“.

Když Morphodita taguje větu, bere ji jako posloupnost forem. Každá forma má nějakou množinu možných významů – odpovídá různým tvarům různých slov (lemmat). Při tagování tedy musí pro každou formu vybrat právě jeden její význam a to dělá na základě statistiky vytvořené z trénovacích dat. Vezměme si například větu:

„Tupá pila špatně řeže.“

Její tagging je:

Tupá|tupý|AAFS1----1A----

pila|pila|NNFS1-----A----

špatně|špatně_^(*1ý)|Dg-------1A----

řeže|řezat|VB-S---3P-AA---

.|.|Z:----------

Jaké jsou ale významy každé z jednotlivých forem?

Tupá: (může to být i příjmení)

Tupá|Tupá_;S_^(*1ý)|Tupá|NNFS1-----A----

Tupá|Tupá_;S_^(*1ý)|Tupá|NNFS5-----A----

Tupá|tupat_:T_,l_^(ťupkat;_tupě_hledět)|tupat|VB-S---3P-AA---

Tupá|tupý|tupý|AAFS1----1A----

Tupá|tupý|tupý|AAFS5----1A----
Tupá|tupý|tupý|AANP1----1A----
Tupá|tupý|tupý|AANP4----1A----
Tupá|tupý|tupý|AANP5----1A----
pila:
pila|pít|pít|VpQW---XR-AA---
pila|pila|pila|NNFS1-----A----
špatně:
špatně|špatně_^(*1ý)|špatně|Dg-------1A----
řeže:
řeže|řezat|řezat|VB-S---3P-AA---
řeže|řež|řež|NNFP1-----A----
řeže|řež|řež|NNFP4-----A----
řeže|řež|řež|NNFP5-----A----
řeže|řež|řež|NNFS2-----A----

Pro ilustraci, že správný tagging vět nemusí být vždy jednoznačný se podívejme na umělou a víceznačnou frázi „Ženu holí stroj.“. Tagger dal nakonec přednost významu „Žena je holena strojem.“:
Ženu|žena|NNFS4-----A----
holí|holit_:T|VB-S---3P-AA---
stroj|stroj|NNIS1-----A----
.|.|Z:----------
Věta připouští ale celkem 5 významů:

(1) Proháním stroj horskou loukou.
(2) Proháním stroj pomocí klacku.
(3) Žena je holena strojem.
(4) Oblékej ženu, která vlastní klacky.
(5) Oblékej ženu pomocí klacku.

Výčet významů pro každou z forem je:
(V závorkách jsou označeny všechny možnosti gramaticky správných výběrů tagů a lemmat, toto jsem přidal ručně a dodatečně.)
ženu:
ženu|hnát-1_^(tvar_slovesa;_[utíkat])|hnát|VB-S---1P-AA--- **(1) (2)**

ženu|žena|žena|NNFS4-----A---- **(3) (4) (5)**

holí:

holí|hole-2_,t_^(horská_louka)|hole|NNFP2-----A----

holí|hole-2_,t_^(horská_louka)|hole|NNFS7-----A---- **(1)**

holí|holit_:T|holit|VB-P---3P-AA---

holí|holit_:T|holit|VB-S---3P-AA--- **(3)**

holí|holý|holý|AAMP1----1A----

holí|holý|holý|AAMP5----1A----

holí|hůl_^(klacek)|hůl|NNFP2-----A---- **(4)**

holí|hůl_^(klacek)|hůl|NNFS7-----A---- **(2) (5)**

stroj:

stroj|stroj|stroj|NNIS1-----A---- **(3)**

stroj|stroj|stroj|NNIS4-----A---- **(1) (2)**

stroj|strojit_:T|strojit|Vi-S---2--A---- **(4) (5)**

stroj|strojit_:T|strojit|Vi-S---3--A---4

Z hlediska fonetického by se dal uvažovat i význam „Ženu holý stroj“, který je foneticky shodný:

ženu|hnát-1_^(tvar_slovesa;_[utíkat])|hnát|VB-S---1P-AA---

holý|holý|holý|AAMS4----1A----

stroj|stroj|stroj|NNIS4-----A----

.|.|Z:---------- 15-1

3.2 Merge-Sorter

Jedná se o standardní řadící algoritmus používaný při načítání dat během spouštění. Používám ho kvůli tomu, že běžný knihovní „quick“ sort jsem shledal neoptimální. Důvody blíže vysvětluji v kapitole „Všeobecně použitelné komponenty“.

3.3. ReadOnly-Dictionaries

ReadOnly dictionaries jsou opět součást mé vlastní knihovny. Tyto nástroje slouží pro rychlé vyhledávání podle klíče v neměnných datech, která jsou připravena jen jednou při spouštění.

3.4. ConcurrentCanonicMerger

ConcurrentCanonicMerger<T> je jednoduchý generický nástroj pro úsporu paměti naprogramovaný především pro tuto aplikaci. Slouží k maximalizaci sdílení referencí na různé instance objektů stejné hodnoty. Zde je používán především pro stringy, jedná se fakticky o množinu stringů, a je používán pouze pro data, o kterých se předem ví, že budou v aplikaci po celou dobu jejího běhu (slovník, atp.) a instance této pomocné třídy je po načtení a zpracování slovníkových dat smazána (implicitně „garbage-collectorem“). Nástroj dostane jako vstup objekt (typu string). Pokud už uložená množina obsahuje string se stejnou hodnotou, vrátí referenci na tuto instanci již existujícího stringu, která má stejnou hodnotu. Toto při optimálním zakomponování umožní „Garbage Collection“ (a tedy úsporu paměti) vstupního stringu s již existující hodnotou. Pokud ji v sobě nenajde, přidá do sebe vstupní instanci stringu a též tu samou instanci vrátí jako výstup.

4. Použitá data, komponenty a jejich příprava

Tato kapitola se zabývá daty, která byla použita pro zakomponování do programu a která program potřebuje pro své fungování.

4.1 Morphodita

Nejdůležitějším souborem je slovník pro knihovnu morphodita_csharp.dll. Formát těchto dat je určen touto použitou knihovnou.

Morphodita je nejprve konvertována do jazyka C# automaticky generovaným kódem. Vzhledem k tomu, že rozhraní tohoto syrového modulu jsem shledal jako nepraktické, zabalil jsem ho do praktičtějšího samotného podprojektu RawWrappedMorphodita. Více je toto popsáno v kapitole 5.1.

4.2 Slovník synonym

Dalším prvkem, který jsem už musel připravovat, je slovník (graf) synonym. Pomocník ho používá při smazání slova uživatelem, aby mohl uživateli nabídnout nové slovo jako náhradu za již smazané slovo, jelikož se předpokládá, že uživatel chtěl smazané slovo nahradit jiným.

Po zpracování je graf synonym v takovém formátu, že na každém řádku je samostatně vypsána jedna dvojice lemmat oddělených znakem | (ASCII 124). Tato lemmata jsou připravena tak, aby odpovídala Morphoditě.

Slovník synonym byl vytvořen na základě zdroje parafrází **http://www.cis.upenn.edu/~ccb/ppdb.** Řádky tohoto zdroje mají následující tvar (zalomení a odsazení doplněno pro lepší přehlednost):

[DT] ||| regionální ||| regionálním |||
Abstract=0 Adjacent=0 Alignment=0-0 CharCountDiff=1
CharLogCR=0.09531 ContainsX=0 GlueRule=0 Identity=0
Lex(e|f)=0.46942 Lex(f|e)=3.34958 Lexical=1 LogCount=0.69315
Monotonic=1 PhrasePenalty=1 RarityPenalty=0
SourceTerminalsButNoTarget=0 SourceWords=1
TargetTerminalsButNoSource=0 TargetWords=1 UnalignedSource=0
UnalignedTarget=0 WordCountDiff=0 WordLenDiff=1.00000
WordLogCR=0 p(LHS|e)=2.37079 p(LHS|f)=2.18097 p(e|LHS)=12.00943
p(e|f)=3.44200 p(e|f,LHS)=3.52517 p(f|LHS)=8.88681 p(f|e)=0.50920
p(f|e,LHS)=0.40254

Seznam parafrází obsahoval velké množství pro mě nepodstatných a nesrozumitelných dat. Na jednom řádku bylo kromě dvojic frází asi 20 čísel udávající „kvalitu“ fráze z různých hledisek. Tyto údaje jsem při zpracování vynechal.

Prohledávat celý slovník ručně by bylo nepřiměřeně drahé, proto po prohlédnutí dostatečně velkého statistického vzorku jsem zjistil, že většina dat nedává pro zdejší účel použití smysl. Pro extrahování potřebných dat jsem jako vyhovující postup shledal stanovení predikátu, který považuji za rozumný dle své intuice a poté projít dostatečný náhodný statistický vzorek jeho aplikace na původní data. Pokud výsledek bude mít relativně málo výsledků „false positive“–neužitečná data ve výsledku i „false negative“ – ztráta užitečných dat, budu predikát brát jako správný. Pokud budu mít hodně „false positive“, ale málo „false negative“, bude predikát vhodný pro konjunkci s dalším predikátem pro další zúžení výsledné množiny. Příkladem tohoto predikátu je neobsahování stejného lemmatu v množině svých významů. Toto například porušují dvojice frází níže. Lze ho intuitivně nazvat jako „neideální vytrhávač plevu“. Pokud naopak budu mít málo „false positive“, ale hodně „false negative“, bude predikát užitečný pro disjunkci s jiným predikátem,

tedy s rozšířením výsledné množiny o další užitečná data. Intuitivně lze nazvat jako „neideální sklízeč zrna“. Pokud neshledám splněnou ani jednu z předchozích možností, predikát prozatím nepoužiju, protože se zatím ukázal jako zcela neužitečný. Nicméně i tak se může stát užitečný pro složení s jiným predikátem. Může se stát například, že v konjunkci s „neideálním vytrhávačem plevu“ se stane „neideálním sklízečem zrna“.

Neužitečné dvojice syrových frází jsou například:

nebude ||| nebudou

nebudou ||| nebude

polovina ||| polovině

polovině ||| polovina

polovina ||| polovině

polovině ||| polovina

Tato částečně zpracovaná data jednak obsahují hodně duplicit zřejmě kvůli tomu, že jsem ignoroval většinu údajů a také většina dvojic jsou pouze různé tvary téhož lemmatu – nepodstatné pro slovník synonym. Tuto mezifázi jsem nikam neukládal, ale hned dále zpracovával.

Přípravu slovníku jsem provedl ve zvláštním programu, kde jsem nejprve filtroval dvojice frází (již zmíněno), následně deduplikoval. Poté jsem každou frázi nechal projít přes Morphoditu (stejnou, která je použita v pomocníku). Pro každý slovní druh významu (v případně nejednoznačnosti) jsem vytvořil separátní dvojici množin lemmat (vysvětleno na příkladu níže). Dále pokud jsem shledal, že obě množiny z dvojice mají jediné a společné lemma, nebo že průnik těchto množin lemmat v případě nejednoznačnosti je neprázdný, dvojici jsem ignoroval. Dále jsem potřeboval, aby obě množiny měly jednoznačné lemma. Pokud neměly, neuměl jsem algoritmicky rozhodnout, které je správné, a raději jsem dvojici též nepoužil a vsadil jsem na to, že mi to nezpůsobí mnoho ztrát užitečných dat. Výsledkem jsou tedy takové dvojice, které mají každá právě jedno lemma a každá má toto lemma různé.

Vzorkem syrových užitečných dvojic je:

hodnocení ||| posouzení

komunikaci ||| sdělení

drahé ||| nákladné

Tyto dvojice byly převedeny na lemmata Morphodity, už finální formát:

hodnocený_^(*4tit)|posouzený_^(*4dit)

hodnocení_^(*4tit)|posouzení_^(*4dit)

komunikace|sdělení_^(*3it)

drahý|nákladný

Jeden řádek by v principu mohl obsahovat i více lemmat oddělených | a byl by hodnocen jako grafová klika. Stejný formát dat má i slovník příbuzných slov popsaný v kapitole 4.3.

Formát XML jsem shledal jako neoptimální pro zdejší použití, neboť režijní data by zabírala minimálně jednou tolik místa, což je při větším množství dat neefektivní a ztěžuje to editaci v textovém editoru s omezenou pamětí.
Jak je vidět, dvojice „hodnocení-posouzení“ vytvořila 2 dvojice, protože podle morphodity jednoznačně odpovídaly jak jednoznačné dvojici přídavných jmen, tak jednoznačné dvojici podstatných jmen. Dvojice sdělení a komunikace byla jednoznačná, nákladný a drahý také.

4.3 Slovník příbuzných slov

Dalším použitým grafem je slovník příbuzných slov. Jako zdroj jsem použil Derinet **http://ufal.mff.cuni.cz/derinet**. Jelikož po prohlédnutí statistického vzorku jsem zjistil, že všechna data jsou smysluplná, nijak jsem je nefiltroval. Tedy filtroval jsem je jen tím, že jsem vzal ty řádky, které měly lemma (pokud ne lemma, alespoň kanonický tvar) shodující se s Morphoditou. Těch bylo naštěstí více než 99%. Každopádně jsem se přesto rozhodl ho převést do formátu, který se bude lépe načítat při spuštění editoru a se kterým editor bude snadněji pracovat. Z hlediska uživatele je důležitá doba čekání při spuštění editoru, nikoliv doba předpřipravení dat mimo editor.
Derinet má konvenci, že na řádku má klíč (jednoznačné ID), slovo v kanonickém tvaru (k tvar), jeho lemma v drtivé většině případů odpovídající Morphoditě a ID slova, ze kterého je slovo odvozeno. Celý slovník je orientovaným grafem, kde směr hrany je pro účely dalšího zpracování brán z odkazovaného slova na odkazující se slovo (například „muž“ –> „mužík“).

Každé slovo tak může mít více následníků, ale jen jediného předchůdce.
Na příkladě ilustrujeme formát řádků:

ID	k. tvar	lemma	sl. d.	ID zdroje	naše pozn.
168199	muž	muž	N		**(1*)**
168204	mužik	mužik	N	168199	**(1) (2*)**
168205	mužikův	mužikův_^(*2)	A	168204	**(2)**
168206	mužnost	mužnost_^(*3ý)	N	168207	**(3)**
168207	mužný	mužný	A	168199	**(1) (3*)**
168208	mužně	mužně_^(*1ý)	D	168207	**(3)**
168210	mužskost	mužskost_^(*3ý)	N	168214	**(4)**
168213	mužsky	mužsky_^(*1ý)	D	168214	**(4)**
168214	mužský	mužský	A	168199	**(1) (4*)**
168215	mužstvo	mužstvo	N	168199	**(1)**
168216	mužství	mužství	N	168214	**(4)**
168219	mužík	mužík	N	168199	**(1)**
168223	mužův	mužův_^(*2)	A	168199	**(1)**

Zkratky v názvech sloupců znamenají: kanonický tvar, slovní druh, naše poznámka (vysvětleno níže)
ID zdroje se odkazuje na případného rodiče ve valenčním grafu (stromě).
Dále byl graf převeden na neorientovaný a to tak, že každý vrchol spolu se všemi svými následníky vytvořil jednu grafovou kliku. V příkladě výše jsou 4 kliky a každá z nich je označena značkou ve sloupci naše poznámka. Je-li u čísla navíc znak „*“, je to vrchol, který tvoří dotyčnou kliku spolu se svými následníky.
Logickou formulí lze konverzi hran vyjádřit:

$$[v1, v2] \in E2 \Leftrightarrow ([v1, v2] \in E1) \vee (\exists v0: (([v0, v1] \in E1) \wedge ([v0, v2] \in E1)))$$

Kde G1: (V, E1) je původní orientovaný graf a G2) : (V, E2) je výsledný neorientovaný graf
Neformálně řečeno:
Dvojicemi ve výsledném grafu se stanou dvojice (rodič–potomek v původním grafu), nebo „sourozenci“ v původním grafu.
Výsledek případu výše:
muž|mužik|mužný|mužský|mužstvo|mužík|mužův_^(*2) **(1)**
mužik|mužikův_^(*2) **(2)**

mužnost_^(*3ý)|mužný|mužně_^(*1ý) **(3)**

mužskost_^(*3ý)|mužsky_^(*1ý)|mužský|mužství **(4)**

Každá klika je tedy vyjádřena na jednom řádku a oddělena znakem | (ASCII 124)

5. Programátorská dokumentace

Tato kapitola se zabývá způsobem implementace a procedurálním popisem programu. Názvy metod a proměnných jsou v programu pojmenované anglicky, jen komentáře a dokumentace jsou občas české pro lepší srozumitelnost. V příloze na konci práce je znázornění hierarchie metod programu.

5.1 Důležitý pomocný prvek - Morphodita

S Morphoditou jsme se seznámili v použitých nástrojích, teď si povíme o tom, jak je zapojena. Morphodita je původně implementovaná v jazyce C++, proto jsem ji musel obalit pro jazyk C#, jak již bylo zmíněno. Je zabalena v prvé řadě do strojem generovaného kódu, se kterým jsem raději jinak nemanipuloval a následně ho zabalil do třídy *WrappedMorphoditaCSharp* a do třídy *WrappedTaggerCSharp*, která je potomkem předchozí třídy. *WrappedTaggerCSharp* je tagger, který taguje větu morfologickými značkami a jeho předek *WrappedMorphoditaCSharp* je slovník, který mi vypíše všechny trojice forma|lemma|tag podle zadané formy nebo lemmatu. K této komponentě přistupuji přímo jen v omezeném regionu (neboť přístup k datům a metodám v komponentách Morphodity je časově náročný) a ve zbytku programu pouze přes metody, které mi data převedou do mnou definovaného (a pro mě kontrolovatelného) formátu. Jelikož jsem shledal, že Morphodita má konvenci, že výstup vrací ve sdílené části paměti a při příštím dotazu někdy na stejné místo vloží nový výsledek, výstupní data okamžitě kopíruji do vlastního datového typu a to hned bezprostředně při jejich přebírání. Toto je praktické také kvůli konvenci výstupu dat. Původní datové typy ihned po použití uklízím pomocí metody Dispose(), kterou volám implicitně přes klíčové slovo „using“. Totéž je praktické také proto, že výstupní data mohu převést do praktického pole.

Třída *WrappedTaggerCSharp* má konstruktor s argumentem jména souboru, ze kterého se načtou data Morphodity. Hlavní instancí Morphodity je tagger, ze kterého se následně získá Morpho pomocí metody *getMorpho*. Instanci Tagger uloží do privátní proměnné této třídy a instanci Morpho do privátní proměnné třídy předka přes neveřejný konstruktor. Předek má taktéž konstruktor, podle kterého se umí načíst ze souboru, ale ten není v této aplikaci dále použit. Pokud se načtení nepovede (je získána null reference), je vyhozena výjimka *FileLoadException*.

Veřejnými metodami této třídy jsou:

public TokenWithRange[][] ParseLineToSentenceTokens (string line)

- Rozdělí celý řádek na slova, respektive tokeny (kromě slov je samostatný token i interpunkce) po větách. Vnější pole obsahuje jednotlivé věty, jeho vnitřní pole jsou věty a obsahují tokeny se svými rozsahy (co položka vnějšího pole to věta, co položka vnitřího pole to token věty s rozsahem. Metoda získá proměnnou tokenizer pomocí metody GetTokenizer z Taggeru uvnitř a do tohoto tokenizeru vloží celý řádek (string). Rozsahem se rozumí začátek a konec ve vstupním stringu. Vytvoří si dočasné datové typy - List<TokenWithRange[]> result – sem postupně přidává hotové věty a List<TokenWithRange> subResult – sem přidává slova jednotlivých vět a po dokončení věty převede na pole, vloží do „result“ a smaže (Clear). Toto provádí ve while cyklu.

V původním rozhraní je nepraktické především to, že Tokeny a jejich rozsahy jsou v oddělených datových typech (Forms a TokenRanges). Proto je pomocí metody Zip (LINQ extenze) slévám do datového typu *TokenWithRange* (výše).

public TaggedLemmaForm[] TagSentence (TokenWithRange[] sentence)

- Přidá k (nejspíše správné české) větě lemmata a tagy - rozsahy z původního „TokenWithRange“ typů ve výstupu nebudou obsaženy. Jedná se fakticky o zprostředkování metody tag původního Taggeru. Do datového typu Form dá původní větu, poté výsledek v datovém typu Tagged Lemmas postupně enumeruje a i s původní větou převádí na datový typ TaggedLemmaForm.

Veřejnými zděděnými metodami této třídy jsou:

public TaggedLemmaForm[] AnalyseForm(string form, int guesser)

public TaggedLemmaForm[] AnalyseForm(string form)

- Výstupem je výčet všech tvarů dané formy - jedná se o zprostředkování metody analyze Morphodity. Druhá metoda jen provolá první s guesser = 0 (jiný není v aplikaci použit).

public TaggedLemmaForm[] FormsOfLemma(string lemma, string tag_wildcard, int guesser)

public TaggedLemmaForm[] FormsOfLemma(string lemma)

- Výstupem je výčet všech tvarů daného lemmatu - jedná se o zprostředkování metody generate Morphodity. Druhá metoda jen provolá první s guesser = 0 a wild_card = null.

public string LemmaId(string lemma)

- Zprostředkování metody lemmaId Morphodity

public string RawForm(string form)

- Zprostředkování metody rawForm Morphodity

public string RawLemma(string lemma)

- Zprostředkování metody rawLemma Morphodity

(poslední 3 metody nejsou v této práci dále užity)

RawMorphoditaWrapper jinak nemá žádné privátní metody, pouze dvě proměnné pro Morphoditu a Tagger

5.2 Třída EditingAssistant (obsahuje tématické jádro programu)

Třída EditingAssistant má konstruktor: EditorAssistant(string morphoFileName, IEnumerable<string> valenceGraphFileNames, IEnumerable<string> synonyms-FileNames, Action<string> logWarning).

Uvnitř si vytvoří instanci RawMorphoditaWrapper, kterou nutně potřebuje pro své fungování. Také načítá soubory ve valenceGraphFileNames a synonymGraph-FileNames Zde je využívána komponenta ConcurrentCanonicMerger (již zmíněno), dále jen optimalizér.

1) Načítání grafu příbuzných slov i synonym probíhá v cyklu, kde se načte řádek, který je rozdělen na slova (lemmata) znakem | (124 ASCII). Slova se rozdělí, jsou puštěna skrze optimalizér a každá tato množina slov je brána jako grafová klika, která je přidána do grafu příbuzných slov. To znamená, že takto vytvořené kliky grafu se mohou vzájemně protínat. Během načítání je graf dočasně reprezentován datovým typem Dictionary<string, SortedSet<string>>, který je po načtení převeden na datový typ ReadOnlySortedDictionary<string, ReadOnlyStringList> - viz **všeobecně užitečné komponenty (kapitola 6)**. Pokud je pro graf synonym a příbuzných slov stanoveno více souborů, dojde k prostému sjednocení dat, jako kdyby všechny soubory byly spojeny.

EditingAssistant má veřejnou vlastnost WrappedTaggerCSharp InnerMorphodita, která je jen pro čtení.

EditingAssistant má dvě soukromé proměnné jen pro čtení typu ReadOnlyHashedDictionary<string, ReadOnlyStringList>: _valenceGraph a _synonymGraph (viz. výše – načítání grafu)

Třída má také hlavní veřejnou metodu, která je již zmíněným tématickým jádrem celého programu:

public ReadOnlyStringList[] DraftsAfterModification (TokenWithRange[] originalSentence, TokenWithRange[] elaboratedSentence)

- Tato metoda dostane jako vstup 2 parametry původní větu a její modifikaci s naznačenou změnou. Výstupem jsou navrhované věty možných celých změn.

Metoda má 7 přímých soukromých podřízených metod.

Metoda nejdříve najde rozsahy, které říkají, které části vět se liší („while“ cykly).

Když zjistí, jaký je změněný úsek, předurčí, o který druh změny se jedná. Zde jde o rozdělení úlohy do několika základních skupin. Může jít jen o pouhé vymazání slova, změnu tvaru slova, úplnou změnu slova, nebo vepsání slova.

Pokud jde o pouhé smazání slova (části textu), volá se privátní metoda:

private ReadOnlyStringList[] PartTextDelete (TaggedLemmaForm[] TaggedOriginalSentence, int indexOfDeleting, int deletedWordCount)

Pokud jde o smazání více, než jednoho slova, metoda vrátí prázdný výsledek (pole délky 0), protože taková úprava není podporovaná, nicméně implementaci je teoreticky možné změnit. Pokud je smazáno právě jedno slovo, provolá se metoda:

private ReadOnlyStringList[] DeletingOfOneWord (TaggedLemmaForm[] TaggedOriginal-Sentence, int indexOfDeleting)

Která dále počítá výsledek. V této metodě se vytvoří lokální proměnná List<ReadOnlyStringList> partialDraftsCumulativeResult, do které jsou průběžně ukládány výsledky, které jsou nakonec převedeny na pole. Je zde ošetřen speciální případ, kdy je smazáno slovo „když“ na začátku věty. V tomto případě se volá privátní metoda

private void DeletingWord_Kdyz_inBeginOfSentence(TaggedLemmaForm[] tagged-OriginalSentence, List<ReadOnlyStringList> partialDraftsOutput).

- Tato metoda změní případný tvar: Když sloveso (předmět ve 4.pádě), (tak)... na (podstatné příbuzné jméno) způsobí, že... Příkladem je věta: Když prší, tak je všechno mokré. -> Pršení/Pršák způsobí, že je všechno mokré. K tomuto je používán graf příbuzných slov. V ostatních případech, pokud nejde o slovo „když“, je řešen pouze případ, kdy je smazáno jen jedno slovo. Program se podívá do grafu synonym a pokud najde synonym(um,a), nabídne náhradu tohoto slova jeho

nalezeným(i) synonym(em,y), samozřejmě v co nejpodobnějším tvaru. Tato metoda používá metodu **ModifyNounOrAdjectiveFormsInRange**.

Pokud je vepsána předložka, volá se metoda:

private void InsertionOrModificationPreposition (TaggedLemmaForm[] prepositionFormsOfModifiedWord, TaggedLemmaForm[] originalTagged-Sentence, int indexOfInsertionOrModification, List<ReadOnlyStringList> newPartialDraftsOutput)

- Tato metoda najde rozsah následující jmenné skupiny (pomocí posouvání indexů ve while-cykle), projde všechny předložkové tvary nového slova, pomocí metody na změnu tvarů v rozsahu provede úpravu v následující jmenné skupině a jako výsledek vrátí úpravy podle tvarů nového slova.

void OneVerbModificationOrAnyWordToVerbModification (TaggedLemmaForm[] verbMeaningsOfModifiedWord, TaggedLemmaForm[] taggedOriginalSentence, int indexOfModifiedVerb, List<ReadOnlyStringList> partialDraftsOutput)

Tato metoda nejdříve určí slovní druh původního slova. Pokud jde o podstatné jméno, zavolá metodu **NounToVerbModification**. Pokud jde o sloveso, zjistí, zda je to aktivum nebo pasivum, a pokud jiný slovní druh, bez dalšího končí, protože se jedná o neznámý případ. Pokud jde o slovesný tvar pasivum, zohlední tvar pomocného slovesa „být“ před tímto původním slovesem, pokud jde o aktivum, zohlední, jestli za ním není zvratné zájmeno „se“, což je nakonec zvratné pasivum. Poté na každý ze slovesných tvarů nového zavolá metodu **OneVerbModification,** jako první parametr mu předá aktuální slovesný tvar, poté tagovanou původní větu, tak jak ji dostala, dále index změněného slovesa, seznam do kterého vkládá výsledky, začátek a konec úseku, kterého se změna týká a informaci, zda jde o pasivum, aktivum nebo zvratné pasivum. Začátek a konec úseku je určen jmennými skupinami před a za slovesem.

void VerbToNounModification(string nounFormOfModifiedWord, TaggedLemmaForm[] taggedOriginalSentence, int indexOfModifiedWord, List<ReadOnlyStringList> partialDraftsOutput)

Metoda dostane jako vstupní data formu nového podstatného jména, původní tagovanou větu a index, kde došlo ke změně slova na toto podstatné jméno. Dále se

pokusí najít jmennou skupinu těsně za změněným slovem. Pokud ji najde, nabídne větu, kde je tato jmenná skupina změněna na druhý pád, jinak nenabídne nic (úprava zůstane bez podpory).

void OneVerbModification (TaggedLemmaForm verbMeaningOfModifiedWord, TaggedLemmaForm[] taggedOriginalSentence, int indexOfModifiedVerb, List<ReadOnlyStringList> partialDraftsOutput, int beginInclusive, int endInclusive, VerbVoice originalVerbVoice)

Metoda už ví podle vstupního parametru, jakého druhu bylo původní sloveso (opisné pasivum, zvratné pasivum, čisté aktivum). Ví také rozsah jmenných skupin, které má měnit podle vstupních parametrů *beginInclusive* a *endInclusive*. Dále zjistí podle tagu, zda nové sloveso je aktivum nebo pasivum. Pokud jsou oboje pasiva nebo oboje aktiva, metoda končí a žádný výsledek nedává, protože se jedná o případ mimo její kompetence (nejde o typ úpravy aktivum <–> pasivum). Totéž udělá i v případě, že nové sloveso patří jinému lemmatu, nežli původní nebo je sloveso zvratným pasivem (které má tag aktiva).

Metoda předpokládá, že uživatel editoru chtěl s nejvyšší pravděpodobností zachovat význam věty, ale možná chtěl změnit směr děje. Pokud chtěl změnit směr děje, stačí pouze změnit pád předmětu, pokud ale chtěl zachovat význam, je vhodné kromě změn pádů také prohodit pořadí podnětných a předmětných jmenných skupin. Počítá se i s tím, že podnět nebo předmět mohou někdy i chybět. Vzhledem k tomu, že metoda hledí na rod a číslo nového slovesa, proto v případě, že podnět a předmět jsou různých rodů nebo čísel, pozná, která ze dvou výše popsaných možností nastává.

Změna Aktiva na pasivum

V případě, že původní sloveso je aktivum (tedy nové je pasivum), jde o případ změny aktiva na pasivum. Nejdříve si metoda zjistí tvar slovesa „být“, který nejvíce sedí na tvar původního slovesa. Sloveso „být“ je jak známo pomocným slovesem v pasivech jako předcházející sloveso před samotným slovesem tvaru pasiva. Dále, pokud se tvar nového slovesa shoduje v rodě a čísle předmětu původní věty, editor přidá do seznamu výsledných nabídek hotovou větu se stejným významem a prohozenými podmětnými a předmětnými jmennými skupinami. Platí v obou uvedených příkladech. Dále zjistí také, jestli se nový tvar slovesa shoduje v rodě a čísle podmětu původní věty. Pokud ano, přidá do seznamu výsledných nabídek

hotovou větu s otočeným významem a pouze změněným tvarem předmětné jmenné skupiny z 4. pádu na 7. pád. Vidět je to na sémanticky nesmyslném příkladě. V příkladu slovesa „žráno“ nebyla splněna podmínka shody rodu a čísla s podmětem.

Pes (žere –> žráno) maso. –> Maso je žráno psem

Hodný tatínek (natírá –> natírán) plot. –> Plot je natírán hodným tatínkem. -> Hodný tatínek je natírán plotem.

Změna Pasiva na Aktivum

Při opačné změně, tedy když původní sloveso je aktivum, postupujeme analogicky. Jen s tím, že musíme naopak smazat pomocný tvar slovesa být a přísudkovou jmennou skupinu změnit na 4. pád místo 7. Taktéž se počítá s případy, že pisatel nejspíše chtěl zachovat význam, ale možná chtěl změnit směr děje, a hledí se na možné nové tvary nového tvaru slovesa.

Pes (žráno -> žere) maso. -> Pes žere maso.;Maso žere psa

Plot je (natírán -> natírá) hodným tatínkem. -> Hodný tatínek natírá plot.; Plot natírá hodného tatínka.

void NounToVerbModification (TaggedLemmaForm[]
verbMeaningsOfModifiedWord, TaggedLemmaForm[]
taggedOriginalSentence, int indexOfModifiedVerb,
List<ReadOnlyStringList> partialDraftsOutput)

tato metoda jen postupně ve for cyklu prochází parametr pole **verbMeaningsOfModifiedWord** pro každou položku předpokládá, že za pravděpodobně gerundiovým podstatným jménem se pravděpodobně vyskytuje jmenná skupina ve druhém pádě. Proto editor navrhne změnu této jmenné skupiny na čtvrtý pád. Tento případ je potenciálně brán jako stále neúplný, kde se možná nenabízí konečná věta po úpravě ale jen pokročilejší mezifáze úpravy. Jde o podobný koncept jako u vepsání nebo změny předložky.

Dalším případem, který se empiricky ukázal jako nejužitečnější, je změna (tvaru) podstatného jména za přídavnou jmennou skupinou. Tento případ je ošetřen v metodě

void NounChangeAfterAdjectiveGroup (TaggedLemmaForm[]
NounFormsOfNewWord, TaggedLemmaForm[]

taggedOriginalSentence, int indexOfModifiedNoun,
List<ReadOnlyStringList> partialDraftsOutput)

Metoda nejdříve najde rozsah přídavné jmenné skupiny (která může obsahovat „a“, „ani“, „i“, „nebo“, čárku) a poté ho pomocí metody **ModifyNounOrAdjectiveFormsInRange** upraví na tvary odpovídající novému tvaru podstatného jména. Příklad: Na okně sedí malá a bílá (kočka -> pejsek). -> Na okně sedí malý a bílý pejsek.

string[] ModifyNounOrAdjectiveFormsInRange (TaggedLemmaForm[]
taggedOriginalSentence, int rangeBeginInclusive, int
rangeEndExclusive, char? newGenus, char? newNumber, char?
newCase)

Metoda slouží více nadřazeným metodám. Jako vstup dostane větu s určeným rozsahem a nové tvary určené vstupními proměnnými typu „nullable char“, na které se má rozsah změnit. Pokud je jako parametr zadána hodnota „null“, znamená to, že tento atribut se měnit nemá. Rozumí se tím, že v určeném rozsahu vybere jména a změní je na nový určený tvar. Dále poté vytvoří pole typu **string[]** o délce změněného rozsahu a zkopírují do něj holé tvary takto vzniklých nových slov (políčko **form** datového typu **TaggedLemmaForm**)

6. Všeobecně užitečné komponenty

V tého kapitole popisuji své vlastní pomocné knihovny.

6.1 ReadOnly-Dictionaries

V knihovně mám implementaci dvou slovníků pouze pro čtení - mají předem pevně daná data a slouží pouze pro rychlá vyhledávání. Slovník je seřazen buď podle hash-kódů (IEqualityComparer<TKey>), nebo podle IComparer<TKey>. Principem je obyčejné binární vyhledávání v seřazeném poli.

6.2 Merge-Sorter

Součástí mé knihovny je také stabilní merge-sorter. Tzv. „quick“ sort jsem shledal jako neefektivní a to ještě nehledě na jeho neoptimální implementaci (vysvětleno níže), kterou mají knihovny C#, protože v případě některých (i když relativně vzácných) permutací (vzhledem k seřazené poslopnosti) dat může složitost degradovat a i v průměrném případě je co do počtu porovnání dražší než merge-sort. Moje implementace alokuje pomocný prostor o velikosti pouze poloviny velikosti seřazovaných dat (s druhou polovinou lze efektivně pracovat bez kopírování mimo seřazované pole). Princip je jednoduchý - pomocné pole se alokuje jen 1x ve veřejné metodě a poté se referenčně a rekurzivně předává parametrem v rámci rekurzivní privátní metody (která je obecnější - seřazuje pole jen v určeném rozsahu). Jednoprvkový rozsah je stabilně seřazen triviálně. Jinak se rozsah rozdělí na 2 poloviční velikosti, rekurzivně seřadí, první rozsah se okopíruje do pomocného pole a poté se slévá - čte se současně z pomocného pole a z druhé části seřazovaného pole. Zdrojový kód je přiložený na CD a ilustrační obrázek je vložen v příloze na konci práce.

A proč mi vadí „quick“-sort v knihovně jazyka C#? Jednak v průměrném případě je merge-sort coby do počtu porovnání rychlejší – tento rozdíl je patrný zejména na datech, kde je jedno porovnání drahé. Napsal jsem pokusný program, kde používám Comparer, který mi může vypisovat, kdy na co je volán a také počítá, kolikrát je volán. Nejdříve testuje „quick“ sort metody OrderBy rozhraní IEnumerable, poté metody Sort třídy Array a poté můj Merge-sort. Seřazoval jsem (pseudo)náhodnou permutaci jen 16 čísel od 0 do 15 (žádné číslo se tedy ve vstupu NEOPAKOVALO), přesto byly zjištěny tyto výsledky:

V následujících podrobných výpisech je vidět, že některé dotazy na vztah byly kladeny zbytečně:

Výpis programu:

15-6 15-14 15-11 15-3 15-8 15-12 15-0 15-15 15-9 15-1 15-4 15-5 15-7 15-13 15-10 15-2 15-15 15-2 9-6 9-14 9-2 9-11 9-10 9-13 9-7 9-3 9-8 9-12 9-5 9-0 9-9 9-4 9-1 9-9 9-1 11-9 11-12 11-14 11-10 11-11 11-13 11-11 9-9 9-10 9-9 12-13 12-14 12-12 13-13 13-14 13-13 8-6 8-2 8-7 8-3 8-8 8-1 8-5 8-0 8-4 8-8 8-4 3-6 3-4 3-0 3-2 3-7 3-5 3-1 3-3 3-3 2-0 2-2 2-1 0-0 0-1 0-0 5-7 5-4 5-5 5-6 5-5 6-6 6-7 6-6

Celkem 85 porovnani u IEnumerable<T>.OrderBy (query quick sort)

6-15 6-9 15-9 6-9 14-9 9-15 9-2 11-9 9-10 9-13 9-7 3-9 8-9 12-9 9-5 0-9 9-9 9-4 1-9 9-9 9-1 9-13 9-15 13-15 9-13 12-13 11-13 13-13 13-15 13-14 13-10 13-14 13-15 14-15 13-14 14-14 14-15 14-14 9-12 9-10 12-10 9-10 10-10 10-12 10-11 10-10 11-12 11-12 11-11 11-12 11-11 6-8 6-1 8-6 1-6 2-6 7-6 6-8 6-4 3-6 6-6 6-0 5-6 6-6 6-5 6-7 6-8 7-8 6-7 7-7 7-8 7-7 1-4 1-5 4-5 1-4 2-4 4-4 4-5 4-0 3-4 4-4 4-3 4-5 4-5 4-4 4-5 4-4 1-2 1-3 2-3 1-2 2-2 2-3 2-0 1-0 0-1 0-0 0-1 0-0 2-3 2-3 2-2 2-3 2-2

Celkem 105 porovnani u Array.Sort (array quick sort)

6-14 11-3 6-3 6-11 14-11 8-12 0-15 8-0 8-15 12-15 3-0 3-8 6-8 11-8 11-12 14-12 14-15 1-4 5-7 1-5 4-5 13-10 2-9 10-2 10-9 1-2 4-2 4-9 5-9 7-9 0-1 3-1 3-2 3-4 6-4 6-5 6-7 8-7 8-9 11-9 11-10 11-13 12-13 14-13

Celkem 44 porovnani u Merge sort

Poté jsem 1024x opakoval pokus s náhodnými permutacemi délky 65536 a výsledek pokusu byl:

Průměrný počet porovnání query quick sort: 1425120, z toho jasně redundantních 158763

Průměrný počet porovnání array quick sort: 1357976, z toho jasně redundantních 256920

Průměrný počet porovnání merge sort: 965705, z toho jasně redundantních 0

Maximální počet porovnání merge sort: 966134

Maximální teoretický počet porovnání merge sort: 983041

Směrodatná odchylka počtu porovnání query quick sort je 17645
Směrodatná odchylka počtu porovnání array quick sort je 17524
Směrodatná odchylka počtu porovnání merge sort je 141

Jak je vidět, metody OrderBy u rozhraní IEnumerable a Sort u třídy Array používají „quick“-sort - výpis porovnání tomu jasně nasvědčuje. U malého počtu prvků méněkrát porovnává „quick“-sort u OrderBy, u většího je tomu naopak. Nicméně, jak je vidět, merge-sort vždy vede.

Kromě toho implementace „quick“-sort knihovny C# dělá porovnání prvků, jehož výsledek už mám možnost znát z porovnání prováděných dříve, nebo dokonce klade opakovaně stejné dotazy. Dále budeme opakované dotazy i v obráceném pořadí nebo porovnání prvku se sebou samým nazývat „jasně redundantní porovnání“, které pokusný program počítá.

Jak je vidět opakovaně se zcela nesmyslně porovnávají dvojice stejných prvků (ačkoli žádné dvě vstupní hodnoty stejné nebyly) a u případu Array.Sort se dokonce opakovaně kladou stejné dotazy, konkrétně například 2-3 se porovnává 5x. Proto zde neshledávám jako nevyhovující samotný „quick“ sort, ale jeho způsob implementace je hodně neoptimální. Přitom i „quick“ sort lze naimplementovat tak, aby nedělal redundantní dotazy - pivota dám vždy stranou, nikdy ho neporovnám sám se sebou a nepošlu ho ani do rekurze. Pokud pivota vybírám třeba jako medián 3 náhodných elementů, mohu ihned vědět, ve kterých částech budou zbylé 2 prvky a už je s pivotem porovnávat nemusím. Ještě v lepším případě z rekurze vynechám i prvky rovné pivotu (udělám třeba zvláštní seznam s indexy těchto prvků) a před zavoláním rekurze pivota (a jemu rovné) zařadím již na správná místa.

Proto jsem se rozhodl použít vlastní merge-sort. Testoval jsem ho spolu se zmíněnými implementacemi „quick“ sortu. A jak je vidět, i nejhorší teoretický případ (tedy spolehlivý horní odhad na počet porovnání) je pořád lepší, než průměrný případ quick-sortu coby do počtu porovnání.

Nejhorší případ je jednoduše vyčíslen rekurentním vztahem (značme ho P pro n celé nezáporné číslo):

$P(0) = 0;$

$P(1) = 0;$

$P(n) = (n - 1) + P((n - 1) / 2) + P((n + 1) / 2); n > 1; n$ liché;

$P(n) = (n - 1) + 2 * P(n / 2); n >= 2; n$ sudé;

n - 1 je režie v nejhorším případě - nastává, pokud na konci slévání zůstane 1 neprázdná posloupnost s 1 elementem. V mnohých případech to ale je trochu lepší, pokud slévání skončí s neprázdnou posloupností s více než 1 elementem.
Z výsledků je také zřejmé, že merge-sort má mnohem menší rozptyl počtu porovnání, než zmíněné „quick“ sorty.

7. Uživatelská dokumentace editoru

7.1 Možnosti editoru

Jedná se o experimentální editor, ve kterém není kladen důraz na běžné vlastnosti jako možnosti ukládání, volba písma, … Jedná se o jednoduchý obyčejný textový editor, pracuje pouze s plain-textem. Má však rozšířenou funkci, že při úpravách textu někdy umí nabídnout dokončení úpravy.

7.2 Druhy úprav, při kterých editor umí nabídnout pomoc:

Editor má funkci, že při úpravě textu umí v některých případech uživateli navrhnout dokončení uživatelem již započaté úpravy. Rozlišuje více druhů možných úprav.

7.2.1 Vepsání nebo změna předložky

Při změně nebo vepsání předložky program nabídne změnu pádu případné následující jmenné skupiny. Toto je založeno na tom, že každá předložka se vždy pojí jen s určitými pády.
Příkladem je fráze ((na) bílá pláň) -> (na bílou pláň/na bílé pláni)

7.2.2 Změna slovesného aktiva na pasivum a obráceně (rozliší i zvratná pasiva)

Editor pozná změnu tvaru aktivum <-> pasivum a nabídne změnu tvarů slov a také prohození podmětu a přísudku. Počítá se s případy, že autor chce i nechce změnit směr děje.
Hodný tatínek (natírá -> natírán) plot. -> Plot je natírán hodným tatínkem.; Hodný tatínek je natírán plotem.

Zde je první nabídka nesmyslná, ale celkově může být užitečné počítat s tím, že upravující chtěl změnit význam.

7.2.3 Změna (gerundiového) podstatného jména na sloveso a obráceně

Změní pád následující jmenné skupiny - při změně na sloveso změní druhý pád na čtvrtý, při změně slovesa na podstatné jméno čtvrtý pád na druhý (dělat koho/co <-> dělání koho/čeho)

7.2.4 Změna (nejen tvaru) podstatného jména za přídavnou jmennou skupinou

Pokud se změní podstatné jméno, nebo jen jeho tvar, editor zjistí, zda se před ním nenachází přídavná jmenná skupina (zohledněny jsou i čárky a spojka „a“). Tato přídavná jména poté přetvaruje na rod, číslo a pád podstatného jména. Příkladem je věta „Na okně sedí malá a bílá (kočka -> pejsek).“ -> „Na okně sedí **malý a bílý** pejsek.“, nebo „Objednal jsem si od vás moc velkou (zásilku -> balík).“-> „Objednal jsem si od vás moc **velký** balík.“

7.2.5 Smazání části textu

Zde je ošetřen speciální případ smazání slova „Když“ na začátku věty. Zde se použije slovník příbuzných slov a pokud za slovem když je sloveso, najde se k němu příbuzné podstatné jméno. Nejnázornější je ukázka na příkladu: „(Když -> 0) prší, (tak) je všechno mokré.“ dá tyto 2 návrhy: (Pršení/Pršák) způsobí, že je všechno mokré.

Druhým a jednodušším případem je smazání jednoho slova. Pokud jsou ve slovníku nalezeno synonymní lemma, editor automaticky navrhne napsání tohoto synonyma místo původního výrazu v nejvhodnějším morfologickém tvaru nebo několik možných tvarů.

7.3 Popis množiny nutných souborů

Prvním povinným souborem, který se musí být umístěn v místě spuštění aplikace je morphodita_csharp.dll a editor_configuration.xml, v němž jsou jména všech ostatních souborů volitelná. Soubor editor_configuration.xml má následující formát:

Kořen se jmenuje „Configuration“, má povinný atribut „morphoditaFileName“ s názvem souboru morphodity.

Soubory grafu synonym jsou v něm uvedeny jako elementy s názvy: „SynonymGraph“ a každý s atributem „fileName“.

Soubory grafu příbuzných slov jsou v něm uvedeny jako elementy s názvy: „ValenceGraph“ a každý s atributem „fileName“.

Pokud bude XML soubor obsahovat další nadbytečné údaje, budou ignorovány, pokud nebude mít správný název kořene, nebo neobsahovat název souboru pro morphoditu, program skončí s chybovou hláškou. Pokud XML soubor nebude mít platný formát (například neuzavřený element), program též může skončit s chybovou hláškou. Pozor, toto není zaručeno, může se stát, že i z neplatného XML souboru se podaří načíst potřebné údaje.

Relační a synonymní graf jsou textové soubory s formátem již popsaným v kapitole 4. (Není nutno porozumět - je samozřejmou součástí)

8. Testování aplikace

Tato kapitola se zabývá samotným testováním aplikace testovacími daty.

8.1. Sběr testovacích dat

Pro ověření funkčnosti a zároveň pro zjištění praktické užitečnosti programu byla připravena testovací data a současně také testovací program, který pro každý vzorek vyhodnotí kvalitu podpory dané úpravy.

Soubor s testovacími daty má formát XML, který má kořen s názvem *Samples*. V kořeni jsou dále elementy s názvem *Category*, který má atribut *description* (popis, nemá další vliv), a elementy s názvem *sample*, který už má jen atributy *original* (původní znění věty), *elaborated* (věta s naznačenou úpravou) a *modified* (dokončená věta). Tento soubor je čten XML (de)serializérem do praktických odpovídajících datových struktur.

Testovací data byla získána na základě čtení textů, převážně šlo o pohádky ze stránek www.abatar.cz, nebo v menšině případů o běžné noviny, či zprávy. Když jsem u nějaké věty intuitivně shledal, že má možnou reformulaci, která by byla pro testování tohoto editoru zajímavá, zkopíroval jsem ji do testovacích dat i včetně její reformulované verze a případně i její mezifázi úpravy. Ve vzorníku jsou však též věty, které nemají stanovenou mezifázi úpravy, protože se zpravidla jednalo jen o

změnu pořadí slov a nebylo jasné, jak úpravu započít. Nicméně tyto věty nejsou dále testovány a slouží jen jako ukázkové příklady dalších druhů úprav, které nejsou zde v této práci řešeny, ale je teoreticky možné se jimi někdy zabývat. Testovací data jsou ručně předtříděna do 4 základních kategorií (viz. tabulka). Tyto kategorie však nemají na testování vliv a slouží pouze pro přehlednost.

Základní kategorie	Počet vzorků	Z toho testovatelných
Změna přídavně jmenné skupiny při změně podstatného jména	134	134
Náhrada slova jeho synonymem nebo slovem s posunutým významem	16	16
Úprava rodu/čísla přísudku při změně podnětu (i v kombinaci se změnou jmenné skupiny)	22	22
Jiné	61	34
Celkem	233	206

V praxi se ukázalo, že nejčastějším požadovaným typem pomoci je změna přídavné jmenné skupiny při změně podstatného jména, a to ať už jde jen o změnu rodu (celého slova) nebo jen o změnu čísla, či pádu. Druhým nejčastějším typem požadované pomoci je změna tvaru slovesa (přísudku) při změně podstatného jména (podmětu), nicméně tento typ úpravy bohužel není implementován a vzhledem k tomu, že jako nástroj je používán tagger založený na statistice, nikoli větný rozbor dávající jasný strom závislostí slov na sobě, by tato implementace nefungovala spolehlivě, protože přísudek může být od podmětu vzdálený i více slov a někdy může dokonce být i v další větě, například, když je zmíněn jen v první větě a v dalších větách je na něj odkazováno zájmenem. Konec konců ani jmenná skupina není vždy jednoznačně před podstatným jménem, ale vzhledem k tomu, že tomu tak je nejméně v 90 procentech případů, v algoritmu přídavnou jmennou skupinu hledám vždy před podstatným jménem. Další požadovanou pomocí je nabídka jiného slova při pouhém smazání slova, ale nicméně množina požadovaných slov je natolik široká, že tento typ pomoci je bohužel málokdy užitečný.

Nicméně i další podporované druhy úprav jsou testovány, ale spíše jen umělými daty a jejich implementace má spíše akademickou hodnotu.

8.2. Testovací program a výsledky testů

Při popisu tříd kvality rozlišujeme návrhy očekávané (vyskytují se přesně v poli odpovědí) a užitečné (viz níže). Pro každý jednotlivý výsledek testovací položky existují tyto třídy kvality podpory, tyto třídy kvality jsou zde seřazené od nejvyšší k nejnižší hodnotě: *(1) jeden a očekávaný návrh, (2) více návrhů a první z nich*

očekávaný, (3) více návrhů a jiný, než první z nich očekávaný, (4) jeden užitečný návrh, (5) více návrhů a první z nich užitečný, (6) více návrhů a jiný, než první a nikoli první z nich užitečný, (7) bez návrhů,(8) jeden nebo více návrhů, ale žádný z nich užitečný.

Pro úplnost ještě dodám, že splní-li výsledek logické podmínky jak pro více tříd kvality, bere se tak, že patří do třídy s vyšší kvalitou.

Pod pojmem užitečný, ale nikoli očekávaný návrh se rozumí, že editor navrhl pouze pokročilejší fázi úpravy, ale nikoli konečnou fázi. Tento případ nastává například pokud program správně nabídne úpravu rodu přídavné jmenné skupiny při změně podstatného jména, ale už nedokáže poznat, že se má též změnit rod přísudku.

Pro ilustraci částečné pomoci je opět výpis posloupnosti mezifází úpravy. V tomto případě však editor při převedení věty do první mezifáze nabídne pouze mezifázi pokročilejší (na 2. řádku nabídne řádek 3.), nikoli koncovou verzi (řádek 4).

1) Bydlela v malém domečku pod tou nejkrásnější kopretinou.
2) Bydlela v malém domečku pod tou nejkrásnější tulipánem.
3) Bydlela v malém domečku pod tou nejkrásnějším tulipánem.
4) Bydlela v malém domečku pod tím nejkrásnějším tulipánem.

V samostatném programu bylo automaticky zpracováno 184 platných testovacích vzorků. Platný testovací vzorek je vzorek mající původní, rozpracovanou a koncovou verzi. Některé vzorky totiž rozpracovanou verzi nemají a mají pouze ilustrační smysl (bylo již zmíněno). Program s aplikací sdílel back-endovou část programu – třídu EditorAssistant, hlavní metodu ale volal s daty získaných z XML souboru, který testovací data obsahoval. Podle toho, zda návrh obsahoval očekávaný návrh, nebo alespoň návrh očekávanému návrhu podobnější, nežli rozpracovaná verze věty, zařadil výsledek testu do příslušné kategorie. Podobnost vět byla hodnocena podle počtu shodných slov. Pokud šlo jen o užitečný návrh (tj. ne úplně shodný), do výsledku byl tento návrh vypsán.

Statistika výsledků:

Způsob podpory	počet	Podíl v %
právě jeden a očekávaný návrh	61	28,77
více návrhů a první z nich byl očekávaný	38	17,92
více návrhů a jeden a nikoli první z nich byl očekávaný	1	0,47

právě jeden užitečný návrh	30	14,15
více návrhů a první z nich byl užitečný	23	10,85
více návrhů, ale žádný z nich nebyl užitečný	8	3,77
jeden návrh, ale nebyl očekávaný	6	2,82
Nemělo žádný návrh	45	21,23
Celkem	212	100

Jak je vidět, 84 vzorků (45,65 %) tedy padlo celkem do prvních 2 nejkvalitnějších kategorií, dalších 46 mělo jako první nebo jediný užitečný návrh.

Vzhledem k tomu, že jen v jediném případě bylo více návrhů, z nichž byl očekávaný nebo užitečný jiný, než první usuzuji, že seřazení návrhů podle pravděpodobnosti očekávání je správné a v případném dalším vývoji by stálo za úvahu, zda v editoru vždy nedávat maximálně jeden návrh místo seznamu návrhů.

9. Závěr

Prvotním motivem práce bylo prozkoumání možnosti usnadnit práci uživatelům textových editorů při úpravách běžného textu. Jak již bylo zmíněno v úvodu, práce je inspirovaná programovacím prostředím Visual Studio, kde editor nabízí uživateli kusy kódu, který pravděpodobně hodlá napsat. Musím z vlastní zkušenosti přiznat, že jsem dokonce na této funkci závislý a když mi na chvíli přestane z nějakého důvodu fungovat, mám problémy vůbec pracovat, zejména musím-li znovu opisovat názvy proměnných, které ani přesně neznám.

V současné podobě představuje práce první prototyp možného řešení dané úlohy.

V back–endu by se určitě dala zlepšit například detekce úprav s podporou a slovník synonym, který by mohl být širší. Nicméně jako hlavní úspěch této práce shledávám implementování nejčastějšího typu podpory změny tvarů přídavné jmenné skupiny při změně podstatného jména, přičemž testovací data pro tento typ úpravy úspěšně prošla testem.

V souladu se zadáním byl front-end navržen a implementován zcela minimalisticky. Je však potřeba přiznat, že otázka interpretace chování uživatele by zasloužila větší pozornost. Je sice nastavitelný interval, kdy je uživatelovo psaní považováno za ukončené, ale jak již bylo zmíněno, chybí například pokus interpretovat chování přes více kontrolních bodů a snažit se zjistit, jestli některá kontrola nebude mít smysluplný výsledek. V současné implementaci se pokus získat smysluplný výsledek provádí jen jednou. Po uživateli je taktéž v jistém ohledu požadováno, aby své chování uzpůsobil pro stanovenou interpretaci.

Jinak z práce jsem si odnesl teoretické poznatky ohledně vztahu mezi českým jazykem a programováním. Dále si myslím, že by bylo pro budoucí textové editory vhodné pokusit se tuto funkčnost zakomponovat.