UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Anže Šuštar

Leksikalna substitucija z učenjem iz uporabnikovega odziva

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Janez Demšar
Somentorica: prof. dr. Dunja Mladenić

Ljubljana, 2011

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01731/2011
Datum: 15.03.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ANŽE ŠUŠTAR**

Naslov: **LEKSIKALNA SUBSTITUCIJA Z UČENJEM IZ UPORABNIKOVEGA ODZIVA**

LEXICAL SUBSTITUTION WITH LEARNING FROM USER RESPONSE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Z vedno večjo prisotnostjo zmogljivih računalnikov v vsakdanjem življenju so jezikovne tehnologije postale eno od najpomembnejših raziskovalnih področij tudi v umetni inteligenci. Eno od njegovih podpodročij je tudi leksikalna substitucija, ki se ukvarja z iskanjem ustreznih sopomenk za besede iz nekega besedila in njihovo smiselno zamenjavo. Pri tem je osnovni problem zaznavanje konteksta, v katerem se pojavlja beseda in, na podlagi tega, razpoznavanje pomena ter sopomenk v danem kontekstu.

V okviru diplomskega dela zasnujte in implementirajte sistem za leksikalno substitucijo, ki bo temeljil na uporabi obstoječih gradnikov, kot so sistemi za lematizacijo, slovarji sopomenk in podobno.

Mentor:

doc. dr. Janez Demšar

Dekan:

prof. dr. Nikolaj Zimic

Somentor:

prof. dr. Dunja Mladenić

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Anže Šuštar,

z vpisno številko 63050116,

sem avtor diplomskega dela z naslovom:

Leksikalna substitucija z učenjem iz uporabnikovega odziva

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  doc. dr. Janeza Demšarja
  in somentorstvom
  prof. dr. Dunje Mladenić
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 9.9.2011 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju doc. dr. Janezu Demšarju in somentorici prof. dr. Dunji Mladenić za vse nasvete in pomoč pri izdelavi diplomskega dela.

Zahvaljujem se tudi dr. Janezu Branku za vse predloge, s katerimi mi je pomagal pri učinkovitejšem pristopu k reševanju problema leksikalne substitucije.

Kazalo

Seznam uporabljenih kratic in simbolov

Ajax - *Asynchronous JavaScript and XML*
API - *Application Programming Interface*
CSS - *Cascading Style Sheets*
DOM - *Document Object Model*
HMM - *Hidden Markov Model*
HTML - *Hypertext Markup Language*
HTTP - *Hypertext Transfer Protocol*
JSON - *JavaScript Object Notation*
MVC - *Model-View-Controller*
PHP - *PHP: Hypertext Preprocessor*

Seznam prevedenih izrazov

Application programming interface - *Vmesnik za programiranje*
Controller - *Krmilnik*
Corpus - *Zbirka*
Document object model - *Objektni model dokumenta*
Drag and drop - *Povleci in spusti*
Event listener - *Detektor dogodkov*
Event-based programming - *Programiranje na osnovi dogodkov*
Full-text index - *Kazalo celotnega besedila*
Hidden Markov model - *Skriti markovski model*
Index - *Kazalo*
Lexeme - *Leksem*
Markup language - *Označevalni jezik*
Model - *Podatkovni model*
Style sheet language - *Stilski jezik*
Part-of-speech tagging - *Določanje besednih vrst*
Progress bar - *Indikator napredka*
Tag - *Oznaka*
Tag set - *Nabor oznak*
View - *Pogled*
Word-sense disambiguation - *Določanje pomena besede*

Povzetek

Diplomsko delo opisuje implementacijo sistema za izvajanje leksikalne substitucije v obliki spletne aplikacije. Leksikalna substitucija je proces iskanja ustreznih sopomenk za besede in besedne zveze, ki se pojavljajo v izvornem besedilu, s čimer lahko ustvarimo pomensko enakovredno besedilo, ki pa uporablja drugačen nabor besed - gradnikov jezika. Na kratko so opisane izbrane tehnologije PHP, MySQL in JavaScript, ki v kombinaciji z označevalnim jezikom HTML in stilskim jezikom CSS tvorijo ogrodje spletne aplikacije. Nadalje je podrobno predstavljen sam problem leksikalne substitucije, bralec pa spozna tudi teoretično zasnovo sistema za leksikalno substitucijo. Teoretični zasnovi sledi podroben prikaz dejanske implementacije sistema z uporabo navedenih tehnologij, pri čemer so predstavljeni razni tehnični izzivi. V zaključku diplomskega dela se srečamo s primeri delovanja dokončanega sistema ter z idejami za nadaljnje delo, ki bi sistem lahko še dodatno izboljšale.

Ključne besede:

leksikalna substitucija, obdelava besedil, spletna aplikacija

Abstract

The thesis focuses on the implementation of an advanced lexical substitution system that was designed as a web application. Lexical substitution is the task of identifying synonyms for words and phrases used in the original text, which enables us to create a new text that is using different words as units of speech while conveying the exact same message. A brief description of the technologies that were used to build the application (namely PHP, MySQL, JavaScript, HTML and CSS) is presented. The problem of lexical substitution itself is presented from a theoretical standpoint, along with a theoretically sound design of a lexical substitution system. The actual implementation of a lexical substitution system and the accompanying technical challenges are presented as the reader progresses. Final results and ideas that could further improve the system are explained in the conclusion.

Key words:

lexical substitution, text manipulation, web application

Poglavje 1

Uvod in podrobna predstavitev ideje

Leksikalna substitucija je tudi v tehnično dobro podkovanih krogih relativno slabo poznan pojem. Sam izraz *leksikalna substitucija* označuje proces določanja ustreznih sopomenk besedam in besednim zvezam, s katerim lahko besedilo prepišemo z uporabo novega nabora besed, pri čemer ohranimo prvotni pomen besedila. Gre za relativno zahteven problem, katerega učinkovita rešitev lahko prinese mnogo koristi.

Ena od teh koristi je na primer učinkovitejša interpretacija vnešenega niza v sistem za iskanje po podatkovni bazi. Uporabnik ob vnosu "dober tiskalnik" od sistema pričakuje, da mu bo vrnil vse zadetke iz semantično (pomensko) ustreznega področja. To pomeni, da sistem vnosa "dober tiskalnik" ne sme interpretirati le v strogem dobesednem smislu, temveč se mora zavedati, da uporabnik pravzaprav išče tudi "kvaliteten tiskalnik" in "zmogljiv printer."

Druga korist je možnost avtomatiziranega generiranja večih različic oglasnega teksta, ki v kombinaciji z avtomatiziranim merjenjem učinka (tako imenovani *A/B test*) omogoča hitro določitev tržno najučinkovitejšega oglasa.

Tretja korist je možnost iskanja pomensko sorodnih besedil, saj se zaradi pestrosti naravnega jezika pri tem opravilu ne kaže zanašati na naivno analizo frekvence uporabljenih ključnih besed, temveč je potreben temeljitejši in robustnejši pristop, ki nam ga ponuja leksikalna substitucija. Tako lahko po vnaprej določenem ključu besede iz dveh besedil menjamo z ustreznimi sopomenkami (npr. vsako besedo menjamo s tisto od ustreznih sopomenk, ki je prva po abecednem redu), s čimer par besedil z istega tematskega področja prevedemo v par besedil, ki načeloma za izražanje istega pomena uporablja iste besede. Na ta način učinkovito razpoznamo, kdaj neko besedilo parafra-

zira drugo besedilo.

Za učinkovito izrabo moči sistema za leksikalno substitucijo pa potrebujemo čim bolj dostopen in dobro premišljen uporabniški vmesnik. Iz tega razloga smo se odločili za implementacijo sistema v obliki spletne aplikacije, zato smo posegli po trenutno aktualnih tehnologijah za spletno programiranje, ki so podrobneje predstavljene v naslednjem poglavju. Sistem s centralizirano (strežniško) zasnovo v obliki spletne aplikacije namreč za uporabo ne postavlja drugih pogojev kot dostop do spleta ter zadovoljiv spletni brskalnik, s čimer se možnost uporabe iz osebnih računalnikov razširi tudi na mobilne telefone, TV vmesnike, igralne konzole in tako dalje.

V tretjem poglavju diplomskega dela se bralec podrobneje seznani s teoretično predstavitvijo samega problema leksikalne substitucije. Spoznamo nekaj ustaljenih pristopov k učinkovitemu reševanju zadanega problema, kar vključuje tudi pregled dela obstoječih prizadevanj na področju določanja besednih vrst, s čimer posamezne besede oziroma dele besedila označimo z ustreznimi besednimi vrstami (samostalnik, pridevnik, glagol itd.). Prav tako se srečamo s teoretično zasnovo sistema, kjer določimo posamezne faze algoritma ter definiramo učinkovito strukturo podatkovne baze za dano problemsko domeno. Dobro premišljena teoretična zasnova nam v nadaljevanju omogoča kakovostno realizacijo sistema.

V četrtem poglavju je podrobno opisan dejanski razvoj sistema za leksikalno substitucijo z vidika strežnika, pri čemer temeljito obdelamo vsako izmed faz algoritma. Najprej se dotaknemo semiotične obdelave besedila (obdelava besedila na nivoju materialne plati jezikovnega znaka), kjer pripravimo začetno besedilo na nadaljnjo obdelavo. Sledi semantična obdelava besedila (obdelava besedila na nivoju nematerialne, tj. pomenske plati jezikovnega znaka), se posvetimo lematizaciji (proces pripisovanja osnovne oblike danim besednim oblikam) in posameznim besedam določimo ustrezno besedno vrsto. Po semantični obdelavi sistem nadaljuje z rapoznavanjem dejanskega pomena posameznih gradnikov besedila ter iskanjem ustreznih sopomenk v prilagojeni podatkovni bazi. Nabor domnevno ustreznih sopomenk ponudi uporabniku, ki s svojim pozitivnim oziroma negativnim odzivom vpliva na delovanje sistema v prihodnje. Učenje iz uporabnikovega odziva je predstavljeno v zadnjem delu četrtega poglavja.

V petem poglavju je bralcu podrobneje predstavljen še uporabniški vmesnik spletne aplikacije ter koncept komunikacije med spletnim brskalnikom ter strežnikom v posameznih fazah obdelave izvornega besedila. Tako si ogledamo potek dejanske uporabe razvitega sistema za leksikalno substitucijo v fazi vnosa ter začetne analize besedila, v fazi interaktivnega določanja ustre-

znih sopomenk za posamezne gradnike besedila (bodisi besede bodisi besedne zveze) ter v fazi končne obdelave besedila, kjer sistem poskrbi za slovnično pravilnost končnega produkta. Bralcu je hkrati predstavljena komunikacija med spletnim brskalnikom ter strežnikom, ki poteka v ozadju delovanja našega sistema.

V zaključnem delu diplomskega dela si ogledamo rezultate delovanja sistema za leksikalno substitucijo na nekaterih praktičnih primerih ter naštejemo nekaj idej, ki bi ob učinkoviti realizaciji lahko še nadalje izboljšale delovanje razvitega sistema za leksikalno substitucijo.

Poglavje 2

Uporabljena orodja in metode pri razvoju spletne aplikacije

Ker smo sistem za leksikalno analizo zasnovali kot spletno aplikacijo, smo morali poseči po tehnologijah in pristopih, ki se uporabljajo za razvoj spletnih aplikacij in spletnih strani v splošnem. V nadaljevanju opisani pristopi in tehnologije nam zagotavljajo vse prednosti spletnega okolja ter kar se da preprosto vzdrževanje programske kode v ozadju.

2.1 Navodila za namestitev spletnega okolja

Za pravilno delovanje spletne aplikacije potrebujemo spletni strežnik z interpreterjem PHP kode ter strežnik MySQL za rokovanje s podatkovno bazo. Za razvoj in testiranje aplikacije smo uporabili brezplačni spletni strežnik Apache z modulom `mod_php`, ki vsebuje interpreter PHP kode.

Za namestitev vsega potrebnega v operacijskem sistemu Windows 7 smo uporabili priljubljen paket Xampp, ki ob privzetih nastavitvah vsebuje primerno nastavljen strežnik Apache s podporo za izvajanje PHP kode ter rokovanje z MySQL podatkovno bazo.

Po namestitvi moramo na strežniku ustvariti posebno mapo, ki vsebuje spletno aplikacijo, ki je v našem primeru seveda sistem za leksikalno substitucijo. V to mapo naložimo vse datoteke, ki jih aplikacija za svoje delovanje potrebuje, razen podatkovne baze, ki teče ločeno na strežniku MySQL. Podrobno datotečno strukturo mape si bomo ogledali v poglavju o softverski arhitekturi Model-View-Controller.

Nazadnje spletni strežnik zaženemo. Do naše spletne aplikacije dostopamo tako, da spletni brskalnik usmerimo na naslov, na katerem se aplikacija na

strežniku nahaja.

2.2 Uporabljene tehnologije pri razvoju uporabniškega vmesnika

Spletna aplikacija je zasnovana z mislijo na delovanje na čim večjem številu različnih platform. Posebno skrb smo namenili temu, da aplikacija nudi popolno funkcionalnost na vseh trenutno aktualnih spletnih brskalnikih:

- Mozilla Firefox (od vključno verzije 4);
- Internet Explorer (od vključno verzije 6);
- Safari (od vključno verzije 4);
- Opera (od vključno verzije 10);
- Google Chrome (od vključno verzije 10).

Brezhibno delovanje v naštetih brskalnikih nam posledično omogoča nemoteno uporabo aplikacije tako v operacijskih sistemih Windows (XP, Vista in 7) kot tudi v operacijskih sistemih Mac OS in raznih distribucijah sistema Linux.

Za postavitev elementov uporabniškega vmesnika se uporablja jezik HTML v povezavi s stilskim jezikom CSS. Sama manipulacija z uporabniškim vmesnikom (tj. manipulacija z elementi, definiranimi v jeziku HTML) je omogočena z uporabo skriptnega jezika JavaScript, ki se izvaja v uporabnikovem brskalniku. Za nemotečo komunikacijo uporabniškega vmesnika z jedrom aplikacije na strežniku smo uporabili nabor tehnologij Ajax, ki nam s pomočjo skriptnega jezika JavaScript in jezika za izmenjavo podatkov XML omogoča uporabniško izkušnjo, ki se na moč približa tisti iz namiznih aplikacij.

2.2.1 HTML, CSS

Kot že omenjeno smo za postavitev elementov uporabniškega vmesnika uporabili jezik HTML, natančneje XHTML™ 1.0, ki je razširitev jezika HTML4. HTML4 je standarden generaliziran označevalni jezik, ki je konformen veljavnemu mednarodnemu standardu ISO 8879, ki je trenutno najbolj razširjen standard med spletnimi vsebinami [1].

Dokumenti XHTML sledijo standardu XML [1], kar pomeni da je objektni model dokumenta (*ang. Document object model*, s kratico DOM) mogoče pregledovati, urejati in preverjati na povsem enak način kot pri XML dokumentih.

Ta lastnost nam bistveno olajša vsakršno manipulacijo (dodajanje ali urejanje elementov uporabniškega vmesnika) nad objektnim modelom dokumenta spletne aplikacije med samim izvajanjem.

```
<html>
  <head>
    <title>Naslov dokumenta</title>
  </head>
  <body>
    Uporabniško ime: <input type="text" name="ime" />
    <submit name="gumb" value="Pošlji!">
  </body>
</html>
```

Slika 2.1: Primer kode v jeziku HTML.

Izgled posameznih elementov uporabniškega vmesnika smo bodisi posredno bodisi neposredno definirali s stilskim jezikom CSS v dokumentih vrste CSS 2.1 (*ang. Cascading Style Sheets*). Določanje izgleda elementov poteka tako, da se posamezne stilske predloge dodajajo elementom na podlagi posebnih kriterijev, s katerimi izberemo enega ali več elementov v dokumentu [2]. Tako lahko npr. definiramo stilsko predlogo "besedilo naj bo podčrtano in modre barve" in jo dodamo vsem elementom tipa *spletna povezava*, medtem ko stilsko predlogo "element naj bo obdan s 3 piksle širokim črnim okvirjem" dodamo vsem elementom tipa *slika*.

2.2.2 JavaScript

Za manipulacijo objektnega modela dokumenta (dodajanje in urejanje elementov v dokumentu XHTML) ter večino komunikacije uporabniškega vmesnika s strežnikom smo uporabili skriptni jezik JavaScript, ki se v celoti izvaja v uporabnikovem brskalniku.

Programiranje v jeziku JavaScript temelji na osnovi dogodkov (*ang. event-based programming*), kar pomeni, da skriptni jezik lahko odreagira na dogodke, ki so se zgodili na vnaprej določenih elementih. Takšni dogodki vključujejo klikanje na vnaprej določene povezave, spremembo izbire v spustnem meniju, premikanje kazalca miške, pritiskanje izbranih tipk na tipkovnici in tako dalje.

Jezik JavaScript nam tudi omogoča izdelavo uporabniškega vmesnika, ki se približa tistemu iz namiznih aplikacij, saj lahko s pomočjo skriptnega jezika neopazno komuniciramo s strežnikom brez osveževanja celotne spletne strani

ob vsakem poslanem zahtevku. Več o tem bomo spregovorili v poglavju o naboru tehnologij Ajax (2.4.1).

2.2.3 Knjižnica jQuery, knjižnica jQuery UI

Pri učinkovitejšem pisanju programske kode v skriptnem jeziku JavaScript nam je lahko v izredno veliko pomoč knjižnica jQuery, saj prinaša 4 pomembne prednosti pri samem razvoju uporabniškega vmesnika aplikacije:

a) Izbiranje elementov v objektnem modelu dokumenta se s pomočjo knjižnice jQuery izredno poenostavi, saj nam ponuja možnost uporabe enakih kriterijev za izbiranje elementov, kot jih poznamo že iz stilskega jezika CSS. To nam omogoča, da izredno preprosto izberemo npr. vse gumbe z imenom "izvedi" ter nanje postavimo detektorje dogodkov (*ang. event listeners*).

b) Tudi manipulacija z objektnim modelom dokumenta je izredno poenostavljena, saj nam knjižnica nudi nabor ukazov za neposredno spreminjanje atributov elementov. To pomeni, da brez večjega truda lahko spremenimo določeno lastnost (barva, velikost besedila itd.) izbrani podmnožici elementov uporabniškega vmesnika.

c) Knjižnica omogoča tudi uporabo asinhrone komunikacije brskalnika s strežnikom, kar nam omogoča hkratno izvajanje večih algoritmov na strežniku. S takšno paralelizacijo lahko pogosto prihranimo levji delež časa, potrebnega za izvajanje določene naloge, obenem pa lahko komuniciramo s strežnikom brez osveževanja celotne spletne strani. Več o tem v poglavju o tehnologiji Ajax (2.4.1).

d) Zadnja pomembna prednost uporabe knjižnice jQuery je v tem, da je njeno delovanje izredno temeljito preizkušeno v vseh priljubljenih brskalnikih, kar pomeni, da ob uporabi ukazov knjižnice jQuery lahko računamo na brezhibno delovanje naše aplikacije preko velikega števila različnih platform.

Dotaknimo se še izredno koristnega dodatka h knjižnici jQuery, in sicer knjižnice jQuery UI. Gre za nabor vtičnikov za izgradnjo uporabniškega vmesnika s podobnim naborom funkcionalnosti, kot nam jih nudijo namizne aplikacije. Tako lahko izredno enostavno v aplikaciji uporabljamo modalna okna za interakcijo z uporabnikom, generiramo zavihke z različnimi nastavitvami, dodajamo drsnike, uporabljamo tehnologijo *povleci in spusti* (*ang. drag and drop*), prikazujemo indikatorje napredka (*ang. progress bar*) in tako dalje.

2.3 Uporabljeni pristopi in tehnologije za izvajanje spletne aplikacije

Izvajanje posameznih algoritmov v okviru spletne aplikacije poteka tako, da spletni brskalnik uporabnika (bodisi asinhrono s pomočjo JavaScript kode, bodisi s preusmeritvijo na drugo spletno stran v okviru iste spletne aplikacije) kliče izvajanje različnih skript PHP, ki so shranjene na strežniku. Te skripte izvedejo zadano nalogo ter spletnemu brskalniku vrnejo rezultat procesiranja, posledično pa lahko uporabnika preko uporabniškega vmesnika na to opozorimo ter mu ponudimo ustrezne možnosti za nadaljevanje z delom.

2.3.1 PHP

PHP je večnamenski odprtokodni skriptni programski jezik, ki večinoma teče na spletnih strežnikih. Sprva je bil namenjen dinamičnemu generiranju kode HTML (od tukaj pomen kratice *PHP: Hypertext Preprocessor*), vendar se je kasneje njegova uporaba zaradi relativne preprostosti, moči ter aktivne baze uporabnikov močno razširila.

Interpreter PHP lahko deluje kot program v ukazni vrstici ali kot del spletnega strežnika, ki ob vsakem prejetem zahtevku zažene novo instanco procesa PHP, izvede programsko kodo ter rezultat pošlje odjemalcu. V primeru izvajanja PHP skripte v okviru spletnega strežnika se celotno delo opravi na strežniku, odjemalec pa prejme le rezultat procesiranja.

```
<?php
  $vsota = 7 + 23;
  echo "Rezultat: " . $vsota;
?>
```

Slika 2.2: Primer kode v skriptnem jeziku PHP.

2.3.2 MySQL

MySQL je sistem za upravljanje z relacijskimi podatkovnimi bazami, ki je bil prvič predstavljen 23. maja 1995, od takrat pa je postal izredno pogosto izbran sistem za upravljanje s podatkovnimi bazami v kontekstu spletnih strani ter spletnih aplikacij. MySQL je ena od osrednjih komponent široko uporabljanega nabora programske opreme za izdelavo spletnih aplikacij *LAMP*: Linux, Apache, MySQL in Perl / PHP / Python.

Trenutno aktualna različica sistema MySQL nosi oznako 5.5 in je uporabnikom na voljo na vseh večjih platformah, ponuja pa tudi shranjene procedure, prožilce, poglede, neodvisne hrambene pogone (*MyISAM* za čim višjo hitrost branja podatkov, *InnoDB* za učinkovite transakcije, *MySQL Archive* za shranjevanje zgodovinskih podatkov ob strogih prostorskih zahtevah), shranjevanje poizvedb, uporabo kazal celotnega besedila (*ang. full-text index*), Unicode podporo itd.

```
SELECT * FROM uporabniki WHERE starost > 18 ORDER BY starost ASC;
```

Slika 2.3: Primer poizvedbe v sistemu MySQL, ki nam vrne vse uporabnike, starejše od 18 let, razvrščene po naraščajoči starosti.

2.3.3 Model-View-Controller arhitektura

Model-view-controller (pogosto označen krajše kot MVC) je softverska arhitektura, ki ločuje *domensko logiko* od *uporabniškega vmesnika*. Takšen pristop nam omogoča neodvisen razvoj, testiranje in vzdrževanje ločenih aspektov spletne aplikacije.

Arhitektura je razdeljena na 3 relativno neodvisne dele:

a) Podatkovni model (*ang. Model*): Skrbi za podatke aplikacije ter za upoštevanje domenske logike v ozadju, odziva se na zahtevke po podatkih ter na ukaze o spremembi trenutnega stanja.

b) Pogled (*ang. View*): Preoblikuje trenutno stanje podatkovnega modela v obliko, ki je primerna za interakcijo z uporabnikom. Vsak podatkovni model lahko uporablja večje število različnih pogledov, ki uporabniku omogočajo različne operacije nad podatkovnim modelom.

c) Krmilnik (*ang. Controller*): Sprejema ukaze preko uporabniškega vmesnika ter sproži obdelavo ukazov nad podatkovnim modelom.

V kontekstu spletne aplikacije si lahko uporabniški vmesnik (predstavljen z jezikom HTML) predstavljamo kot pogled, ki omogoča pošiljanje uporabniških zahtevkov preko GET in POST protokolov ustreznemu krmilniku. Krmilnik nato obdela zahtevke ter poskrbi za ustrezna nadaljnja navodila podatkovnemu modelu, ki jih obdela v skladu s trenutnim stanjem podatkov ter določili domenske logike.

Slika 2.4: Datotečna struktura spletne aplikacije, ki uporablja arhitekturo MVC.

2.4 Povezava uporabniškega vmesnika in strežniškega dela

Za učinkovito delovanje spletne aplikacije mora biti omogočena brezhibna komunikacija med uporabniškim vmesnikom, ki se nahaja v spletnem brskalniku uporabnika, ter strežniškim delom aplikacije, ki se navadno nahaja na povsem ločenem računalniku, do katerega uporabnik dostopa preko spleta.

2.4.1 Pristop Ajax (Asynchronous JavaScript and XML)

Za povezavo uporabniškega vmesnika in strežniškega dela aplikacije smo uporabili protokol HTTP. Vsak zahtevek lahko oblikujemo kot zahtevek tipa HTTP POST, ki se s pomočjo tehnologije JavaScript (in knjižnice jQuery, ki nam delo močno olajša in poskrbi za brezhibno delovanje na širokem spektru trenutno aktualnih spletnih brskalnikov) posreduje vnaprej določeni skripti PHP, ki je shranjena na strežniku.

Pri komunikaciji uporabniškega vmesnika s strežniškim delom aplikacije stremimo k temu, da se ob poslanih zahtevkih vsebina uporabniškega vmesnika ne osvežuje v celoti, kot je to v navadi pri običajnih spletnih straneh. Takšna osvežitev (ponovno nalaganje) uporabniškega vmesnika ob vsakem zahtevku bi pomenila bodisi izgubo trenutnega stanja uporabniškega vmesnika bodisi veliko količino dodatnega razvijalskega dela, s katerim bi zagotovili nemoteno delovanje aplikacije. Da se osveževanju v veliki meri izognemo, smo se poslužili skupka tehnologij Ajax (*Asynchronous JavaScript And XML*). Ajax [3] nam omogoča uporabo skript na strežniku in obdelavo vrnjenih rezultatov brez

osveževanja uporabniškega vmesnika, omogoča pa nam tudi sočasno izvajanje večjega števila skript (vsaka skripta se na strežniku izvede kot nova instanca interpreterja PHP).

Ajax je torej nabor tehnologij, ki uporablja kombinacijo tehnologij HTML in CSS za izgradnjo uporabniškega vmesnika ter tehnologije JavaScript za nadzor nad objektnim modelom dokumenta in ustrezno obdelavo akcij, ki jih sproži uporabnik, kar vključuje tudi obdelavo rezultatov, ki jih vrnejo skripte na strežniku. Skriptni jezik JavaScript in objekt XMLHttpRequest (ki predstavlja implementacijo asinhrone komunikacije s strežnikom v vseh sodobnih brskalnikih) nudita možnost za asinhrono izmenjavo podatkov med spletnim brskalnikom in strežnikom, s čimer se izognemo osvežitvi celotne spletne strani ob vsakem zahtevku in poskrbimo za mnogo prijaznejši uporabniški vmesnik.

Brez tehnologije Ajax bi bila razvoj in uporaba spletne aplikacije, predstavljene v tem diplomskem delu, zelo otežena, tako z vidika razvijalca kot tudi z vidika uporabnika (daljši nalagalni časi ob zahtevanih akcijah, nezmožnost sočasnega izvajanja opravil).

2.4.2 JSON za prenos podatkov

JSON je strojno nezahteven odprt standard za izmenjavo podatkov v berljivi obliki - zasnovan je namreč na izmenjavi podatkov v tekstovni obliki. Izpeljan je iz skriptnega jezika JavaScript, kjer se uporablja za predstavitev enostavnih podatkovnih struktur ter asociativnih polj, čemur lahko v okviru jezika JavaScript rečemo tudi objekti. Čeprav izvira iz jezika JavaScript, je od samega jezika neodvisen in se v praksi za izmenjavo podatokv lahko uporablja s poljubnim programskim jezikom.

V naši spletni aplikaciji bomo uporabljali JSON format za izmenjavo podatkov med uporabniškim vmesnikom ter strežniškim delom aplikacije. Enostavna berljivost podatkov med prenosom razvijalcu tudi močno olajša iskanje napak v programski kodi, saj lahko z ustreznim orodjem (npr. HttpFox vtičnikom za spletni brskalnik Mozilla Firefox) tekom izvajanja aplikacije spremljamo vso izmenjavo podatkov ter sproti odkrivamo morebitne nepravilnosti pri prenosu podatkov.

```
{
  "ime": "Anze",
  "priimek": "Sustar",
  "naslov":
  {
    "mesto": "Ljubljana",
    "drzava": "Slovenija"
  }
}
```

Slika 2.5: Primer JSON predstavitve objekta.

Poglavje 3

Zasnova sistema za leksikalno substitucijo

Pri izdelavi sistema za leksikalno substitucijo se bo za izredno pomembno izkazala dobro premišljena ter učinkovita zasnova samega sistema, ki bo v nadaljevanju nudila dobro osnovo za implementacijo željene funkcionalnosti.

3.1 Definicija leksikalne substitucije v kontekstu obdelave besedil

Leksikalna substitucija je proces iskanja ustreznih sopomenk za besede in besedne zveze, ki se pojavljajo v izvornem besedilu, s čimer lahko ustvarimo pomensko enakovredno besedilo, ki pa uporablja drugačen nabor besed in besednih zvez (skupno jim recimo kar *gradniki besedila*) [4]. V okviru tega diplomskega dela bomo izdelali sistem za izvajanje leksikalne substitucije nad besedili v angleškem jeziku, tako da si tudi primer oglejmo kar v angleškem jeziku. Vzemimo sledečo poved za naše izvorno besedilo:

John herded his sheep into a pen.

Izvorno besedilo lahko z uporabo leksikalne substitucije zapišemo z uporabo drugih gradnikov besedila, pri čemer ohranimo prvotni pomen besedila. Tako lahko besedilo z menjavo glagola “herd” z glagolom “guide” preoblikujemo v:

John guided his sheep into a pen.

V obeh primerih bi besedilo prevedli v slovenščino kot "John je nagnal svoje ovce v ograd." To nam potrjuje hipotezo, da besedilo kljub uporabi različnih besed nosi povsem enako sporočilo.

V nadaljevanju se bo izkazalo, da je za uspešen pristop k leksikalni substituciji potrebno mnogo dela na področju razlikovanja med možnimi pomeni posameznih gradnikov besedila (*ang. Word-sense disambiguation*) [5] [6], prav tako pa bomo po opravljeni leksikalni substituciji morali naknadno poskrbeti za ohranitev slovnične pravilnosti besedila.

3.2 Cilj sistema za leksikalno substitucijo

Naš sistem za leksikalno substitucijo bo kot vhod sprejel kakršnokoli besedilo poljubne dolžine v angleškem jeziku. S pomočjo razpoznavanja dejanskega pomena posameznih gradnikov besedila (besed in besednih zvez) bo sistem sposoben uporabniku samodejno ponuditi ustrezno preoblikovane sopomenke za posamezne gradnike besedila, z uporabo katerih bo besedilo ohranilo prvotni pomen, vendar bo za izražanje pomena uporabljalo drugačen nabor besed in besednih zvez.

Oglejmo si še enkrat sledečo poved:

John herded his sheep into a pen.

Sistem bo najprej razpoznal besedo "herded" kot glagol z osnovno obliko "(to) herd," ki pomeni "gnati trop živali v izbrano smer." Omenjeni glagol v trenutnem besedilu nastopa v pretekliku, zato bo sistem pripravil ustrezno preoblikovan seznam sorodnih glagolov, ki bodo prav tako nastopali v pretekliku

Nato bo razpoznal besedo "pen" kot samostalnik, ki predstavlja *ograjo za živali* in ne *pisalo*, čeprav se *pisalo* pojavlja kot mnogo pogostejši pomen besede "pen."

V tem primeru moramo torej razlikovati med vsaj dvema možnima pomenoma besede "pen": *ograda* ali *pisalo*. To nam pove, da je sam problem leksikalne substitucije tesno povezan s problemom razlikovanja pomenov besed, saj oba problema stremita k uspešnemu določanju dejanskega pomena zapisane besede. Vendar pa med problemoma obstaja ključna razlika: Medtem, ko problem razlikovanja pomenov besed pravzaprav predstavlja samodejno razvrščanje besed k enemu izmed vnaprej določenih možnih pomenov, problem leksikalne substitucije ni vnaprej omejen s končno množico izbir. Sistem lahko operira brez vnaprej podane končne množice možnih izbir, s čimer

se izognemu problemu razdrobljenosti posameznih pomenov, zaradi katerega imajo lahko tudi ljudje (in ne samo algoritmi) težave s povsem ustreznim razvrščanjem besed. Pri leksikalni substituciji moramo torej najti le sopomenko izvorne besede, ki ustreza istemu pomenu kot naša izvorna beseda.

Ko sistem pripravi ustrezno preoblikovane sezname sopomenk za posamezne gradnike prvotnega besedila, prvotnim gradnikom doda najustreznejše sopomenke s pomočjo posebne sintakse, s katero navedemo vsako množico sopomenk znotraj zavitih oklepajev ter jih med seboj ločimo z navpično črto.

Oglejmo si enega od možnih končnih rezultatov našega sistema za leksikalno substitucijo:

$$John\ \{herded \mid rallied \mid guided \mid drove \mid forced\}$$

$$his\ sheep\ into\ \{a\ pen \mid a\ sty \mid a\ corral \mid an\ enclosure\}.$$

Na tem mestu velja izpostaviti še dejstvo, da mora naš sistem pri uporabi sopomenk tudi samodejno poskrbeti za uporabo ustreznega nedoločnega člena "a" oziroma "an," s čimer doseže dejstvo, da je vsaka izmed možnih variacij prvotnega besedila slovnično pravilna.

Omenimo še, da sistem dopušča uporabo gnezdene sintakse za navedbo sopomenk, kot v sledečem primeru:

$$This\ is\ \{too\ \{expensive \mid pricey\} \mid overpriced\}.$$

Iz besedil, zapisanih z uporabo ravnokar opisane sintakse, zna sistem v zadnjem koraku leksikalne substitucije samodejno generirati množico različnih besedil, ki nosijo enak pomen.

3.3 Štirislojni pristop k leksikalni substituciji

Za učinkovitega se je izkazal pristop k leksikalni substituciji, pri katerem nad vhodnimi podatki operiramo v štirih korakih, začenši z določanjem besednih vrst.

3.3.1 Določanje besednih vrst

Določanje besednih vrst (*ang. part-of-speech tagging*) je v problemski domeni obdelave besedil poznano tudi pod imenoma *označevanje gramatike* ter *razlikovanje besednih kategorij*. Gre za proces označevanja gradnikov besedila (besed in besednih zvez) z ustreznimi besednimi vrstami, kar nam pove mnogo

tudi o samem pomenu posameznih delov besedila.

V slovenskem jeziku poznamo:

- predmetnopomenske besedne vrste;
- slovnične besedne vrste.

Predmetnopomenske besedne vrste nadalje delimo na:

- samostalniška beseda (samostalnik, samostalniški zaimek, posamostaljena pridevniška beseda);
- pridevniška beseda (pridevnik, števnik, pridevniški zaimek);
- glagol;
- povedkovnik;
- prislov (krajevni, časovni, vzorčni ali lastnostni).

Slovnične besedne vrste delimo na:

- predlog;
- veznik;
- členek;
- medmet.

3.3.2 Razpoznavanje pomena besed

Po uspešno zaključenem procesu določanja besednih vrst nadaljujemo z razpoznavanjem pomena posameznih besed oziroma besednih zvez. V tem koraku si lahko pomagamo z obstoječo podatkovno bazo gradnikov jezika, nad katerim operiramo. Za vsako besedo oziroma besedno zvezo opravimo poizvedbo, ki nam pove, ali ima določen gradnik besedila v danem jeziku več možnih pomenov. Če ima gradnik res več možnih pomenov, z nadaljnjo analizo sobesedila s pomočjo kontekstnih vektorjev, ki bodo opisani v nadaljevanju, presodimo o tem, kateri pomen lahko z največjo verjetnostjo pripišemo danemu gradniku.

V primeru, da nam s pomočjo algoritma ne uspe z zadovoljivo gotovostjo ugotoviti dejanskega pomena besede ali besedne zveze, lahko o tem seveda povprašamo uporabnika spletne aplikacije ter mu kot pomoč pri izbiri ponudimo daljšo razlago vsakega od možnih pomenov. Treba pa se je zavedati, da vsakršna interakcija z uporabnikom zmanjšuje uporabnost celovitega sistema za leksikalno substitucijo, ki bi v idealnem svetu deloval brez vsakršnega posredovanja končnega uporabnika.

3.3.3 Določanje ustreznih sopomenk

Ko smo posameznim besedam oziroma besednim zvezam določili dejanski pomen, lahko s pomočjo informacij, ki so nam na voljo, pričnemo z iskanjem ustreznih sopomenk. Kot si bomo ogledali v nadaljevanju, obstajajo podatkovne baze, ki predstavljajo relativno dobro razdelano strukturo angleškega jezika in ki nam bodo pri tem v veliko pomoč.

Pri samem iskanju ustreznih sopomenk si bomo pomagali tako s seznami sopomenk kot tudi s seznami nadpomenk in podpomenk, ogledali si bomo sorodne besede ter besede s podobnim izvorom, včasih pa nam bodo pri leksikalni substituciji pomagale tudi protipomenke.

Ker lahko z interpretacijo podatkov iz obstoječih podatkovnih baz, ki se nanašajo na relacije med besedami znotraj angleškega jezika (npr. WordNet), sklepamo na dejansko stopnjo povezanosti posameznih sopomenk z izvorno besedo [13], bo naš sistem sopomenke vedno tudi razvrstil od najbolj primernih do najmanj primernih. Informacijo o ustreznosti izbranega pomena ter predlaganih sopomenk v določenih kontekstih bo s pomočjo uporabnikovega odziva nenehno dopolnjeval ter s tem izboljševal svoje delovanje.

3.3.4 Učenje iz uporabnikovega odziva

V ključnem koraku leksikalne substitucije bo uporabniku spletne aplikacije na voljo možnost označevanja predlaganih pomenov besed ter sopomenk z implicitnima oznakama *ustreza* oziroma *ne ustreza*, s čimer bo sistemu posredoval novo informacijo, ki je le-ta prej še ni imel na voljo. Na podlagi povratnih informacij (uporabnikovega odziva) se bo naš sistem kontinuirano učil ter prilagajal svojo delovno bazo pomenov ter sopomenk s ciljem, da bo kasneje uporabnikom sposoben v večini primerov ponuditi povsem ustrezne nabore sopomenk brez vsakršne zunanje pomoči.

3.4 Določanje besednih vrst

Določanje besednih vrst (*ang. part-of-speech tagging*) je, kot že rečeno, problem iz problemske domene razlikovanja posameznih kategorij, v katerih gradniki besedila lahko nastopajo.

Ker določene besede lahko nastopajo v različnih besednih vrstah in ker je pomen besedila lahko kompleksen ali celo deloma neizgovorjen, gre za zahteven problem s tehničnega vidika, ki so ga v preteklosti reševali z množico različnih pristopov [7].

Reševanje problema je tesno povezano z obdelavo večjih količin besedila v danem jeziku. Prva večja zbirka (*ang. corpus*) besedil v angleškem jeziku z namenom računalniške analize je *Brown Corpus*, ki so ga v šestdesetih letih prejšnjega stoletja razvili na Brown University [8]. Sestavlja ga približno 1.000.000 besed angleške proze, pri čemer so uporabili 500 naključno izbranih besedil. Vsako besedilo vsebuje dobrih 2.000 besed, saj se zaključi po prvi povedi, ki sledi 2.000. besedi - na ta način zbirka vsebuje samo zaključene povedi. Celotno zbirko so najprej obdelali z računalniškim programom, ki je z množico kompleksnih pravil o sobivanju določenih besednih vrst znotraj posameznih povedi dosegel približno 70-odstotno učinkovitost. Nadaljnja obdelava je bila izvedeno pretežno ročno in do konca sedemdesetih let prejšnjega stoletja je bila celotna zbirka obdelana skorajda v popolnosti.

Zbirka je bila kasneje uporabljena v mnogih nadaljnih študijah o pogostosti posameznih besed ter algoritmičnem določanju besednih vrst, posledično pa so nastale podobne zbirke tudi v drugih svetovnih jezikih. Statistični podatki, pridobljeni z analizami zbirke, so v nadaljnjih desetletjih predstavljali osnovo velikega števila avtomatiziranih sistemov za določanje besednih vrst, danes pa sistemi uporabljajo še popolnejše zbirke, ki so bile razvite kasneje in ki vsebujejo še večje število besed, npr. *British National Corpus* s 100 milijoni besed.

3.4.1 Zasnova algoritma za določanje besednih vrst

Najenostavnejši algoritem, ki si ga lahko zamislimo, je verjetno sledeč: Najprej analiziramo besede iz velike zbirke besedila, ki je bila poprej ustrezno označena, ter si zapomnimo, s kakšno verjetnostjo posamezna beseda nastopa v vlogi posamezne besedne vrste. Nato vsaki besede iz našega vhodnega besedila (ki je še neoznačeno) priredimo besedno vrsto, ki najpogosteje ustreza dani besedi. Na tak način lahko nekoliko presenetljivo pridemo do približno 90% zanesljivosti pri določanju besednih vrst, kar priča o tem, da se v besedilih

pojavlja obvladljivo število besed z večjim številom pomenov, ter da se tudi pri besedah z večjim številom pomenov eden od teh pomenov navadno pojavlja mnogo pogosteje od ostalih.

Vendar pa lahko tako osnoven pristop definitivno izboljšamo z naprednejšimi prijemi. V osemdesetih letih prejšnjega stoletja so tako evropski znanstveniki pričeli z uporabo skritih markovskih modelov (*ang. Hidden Markov Model*, s kratico HMM) za razlikovanje med posameznimi deli besedila. Takšen pristop pravzaprav analizira besede ter prirejene besedne vrste (npr. iz zbirke *Brown Corpus*) ter izgradi tabelo verjetnosti posameznih zaporedij delov besedila.

Kot primer si lahko ogledamo stanje, ko naletimo na besedo "the" - hipotetično tej besedi v 50% primerov sledi samostalnik, v 40% primerov pridevnik in v 10% primerov številnik. Na podlagi te informacije lahko program zaključi, da je beseda "can" v zaporedju "the can" najverjetneje samostalnik in ne modalni glagol, saj se glagoli v takšnem zaporedju praktično nikoli ne pojavljajo.

Skriti markovski modeli višjih redov ne analizirajo zgolj verjetnosti posameznih zaporedij dveh besednih vrst, temveč analizirajo tudi trojice ali celo večje sklope besed. Tako na primer pridejo do zaključka, da po zaporedju predloga in glagola zelo redko nastopi še en glagol, mnogo pogosteje pa nastopi samostalnik.

Ko naletimo na zaporedje besed, pri katerih ne moremo zanesljivo določiti besedne vrste, si lahko pomagamo z relativnimi verjetnostmi posameznih zaporedij. S tem pristopom se lahko odločimo za najverjetnejšo kombinacijo besednih vrst, kar se v praksi izkaže za učinkovit pristop, ki omogoča izdelavo algoritma za določanje besednih vrst s 93 do 95-odstotno učinkovitostjo.

Pristop k reševanju problema s pomočjo skritih markovskih modelov pa pesti izredna časovna potratnost in kmalu se je pokazala potreba po učinkovitejših algoritmih. S tem v mislih sta leta 1987 Steven DeRose [9] in Ken Church [10] neodvisno razvila metode dinamičnega programiranja, ki rešujejo problem v mnogo krajšem času in prav tako na nek način temeljijo na skritih markovskih modelih. Njun pristop je bil podoben Viterbijevemu algoritmu, ki je algoritem dinamičnega programiranja za iskanje najbolj verjetnih zaporedij skritih stanj (temu pravimo "Viterbijeva pot"), ki rezultira v zaporedju opazovanih dogodkov v kontekstu skritih markovskih modelov.

DeRose je uporabil tabelo parov zaporednih besednih vrst, Church pa celo tabelo trojčkov. Obe metodi sta dosegli nadvse zadovoljivo učinkovitost (nad 95%) ob bistveno hitrejšem izvajanju od prejšnjih pristopov.

Ta odkritja so pomenila velik miselni preskok na področju procesiranja in analize naravnega jezika. Izkazalo se je, da za določanje besednih vrst izredno sofisticirani in potratni algoritmi za razpoznavanje dejanskega pomena

besedila niso nujno potrebni, in da lahko določanje besednih vrst smatramo kot povsem ločen problem. Ločitev določanja besednih vrst od ostalih manipulacij nad naravnim jezikom je pomenila bistveno poenostavitev teorije in prakse računalniške obdelave naravnega jezika ter spodbudila druge raziskovalce, da na podoben način ločeno razdelali tudi preostala področja analize naravnega jezika.

3.4.2 Nabor oznak "Brown Corpus"

Omenjali smo že zbirko besedil v angleškem jeziku, poznano pod imenom *Brown Corpus*, ki so jo raziskovalci z Brown University dodobra razdelali ter posamezne gradnike besedila označili z ustreznimi besednimi vrstami.

Seveda pa so pri tem naleteli na fundamentalen problem - določiti je bilo treba posamezne besedne vrste, v katerih se gradniki besedila pravzaprav sploh lahko pojavljajo. Vsaki besedi je namreč treba prirediti točno eno, najbolj ustrezno besedno vrsto. V ta namen so izdelali izredno obširen nabor oznak (*ang. tag set*), ki vsebuje kar 226 oznak.

Ker je nabor oznak mnogo preobširen za objavo v tem diplomskem delu, si ga lahko bralec v celoti ogleda na sledečem spletnem naslovu:

http://www.comp.leeds.ac.uk/ccalas/tagsets/brown.html

3.4.3 Nabor oznak "Penn Treebank"

Nabor oznak *Brown Corpus* se je v določenih primerih s svojimi 226 oznakami izkazal za preobsežnega in kmalu se je pokazala potreba po okleščenem, bolj funkcionalnem naboru oznak.

V ta namen so na univerzi UPenn razvili *Penn Treebank* nabor oznak [11], ki vsebuje le 36 oznak, s katerimi lahko ravno tako prekrijemo vsako besedilo.

Oznaka	**Definicija v angleškem jeziku**
CC	coordinating conjunction
CD	cardinal number
DT	determiner
EX	existential there
FW	foreign word
IN	preposition or subordinating conjunction
JJ	adjective
JJR	adjective, comparative
JJS	adjective, superlative
LS	list item marker
MD	modal
NN	noun, singular or mass
NNS	noun, plural
NP	proper noun, singular
NPS	proper noun, plural

PDT	predeterminer
POS	possessive ending
PP	personal pronoun
PP$	possessive pronoun
RB	adverb
RBR	adverb, comparative
RBS	adverb, superlative
RP	particle
SYM	symbol
TO	to
UH	interjection
VB	verb, base form
VBD	verb, past tense
VBG	verb, gerund or present participle
VBN	verb, past participle
VBP	verb, non-3rd person singular present
VBZ	verb, 3rd person singular present
WDT	wh-determiner
WP	wh-pronoun
WP$	possessive wh-pronoun
WRB	wh-adverb

3.4.4 Poenostavitev "Penn Treebank" nabora oznak

Za potrebe leksikalne substitucije v okviru tega diplomskega dela smo nabor oznak *Penn Treebank* še dodatno skrčili na vsega 10 oznak.

Oznaka	**Penn Treebank ekvivalent**	**Definicija v angleškem jeziku**
JJ	JJ, JJR, JJS	adjective
AV	RB, RBR, RBS	adverb
CC	CC	conjunction
DT	DT, PDT	determiner
UH	UH	interjection
NN	CD, FW, NN, NNS, NP, NPS, SYM	noun
PP	IN	preposition
PR	PP, PP$	pronoun
VB	VB, VBD, VBG, VBN, VBP, VBZ	verb
ZZ	/	unknown-other

Poenostavljeni nabor se bo v nadaljevanju izkazal za dovolj učinkovitega za potrebe leksikalne substitucije, predvsem pa bo olajšal iskanje sopomenk v podatkovni bazi, saj so posamezne besedne vrste z njim bolje združene kot z bolj razdrobljenim modelom.

3.5 Podatkovna baza sistema

Za učinkovito delovanje sistema za leksikalno substitucijo potrebujemo učinkovito zasnovano podatkovno bazo, s katero si bomo pomagali pri preoblikovanju besedila, ki bo ob uporabi drugačnih besed in besednih zvez (gradnikov jezika) ohranilo identično sporočilo.

Za dober pristop se je izkazala uporaba prilagojene relativno dobro poznane leksikalne podatkovne baze angleškega jezika WordNet, ki jo je na Princeton University razvil profesor psihologije George A. Miller s svojo ekipo [12].

3.5.1 Predstavitev leksikalne podatkovne baze WordNet

Leksikalna podatkovna baza WordNet združuje angleške besede v skupine sopomenk ter uporabnikom ponuja dostop do definicij posameznih besed ter medsebojnih relacij med besedami oziroma skupinami sopomenk. Osrednji vzgib, ki je botroval razvoju leksikalne podatkovne baze, je bila združitev slovarja definicij ter slovarja sopomenk v uporabnejšo celoto, ki bi omogočala nov pristop k analizi besedil ter olajšala nekatere vidike umetne inteligence.

Leksikalna podatkovna baza WordNet je izdana pod licenco BSD in je prosto dostopna vsem potencialnim uporabnikom. Trenutno aktualna različica (3.0) vsebuje 155.287 besed, ki so združene v 117.659 sklopov sopomenk. Vseh povezav posameznih besed s posameznimi sklopi sopomenk je 206.981 - nekatere besede namreč pripadajo večim sklopom sopomenk, saj imajo več različnih pomenov.

V izvirniku je celotna kolekcija na voljo kot aplikacija z vgrajeno podatkovno bazo, ki deluje na operacijskih sistemih Windows in Unix, sama podatkovna baza pa je dostopna tudi v obliki ANSI Prolog datotek.

Iz podatkovne baze v Prologovem formatu je z nekaj prilagoditvami mogoča interpretacija podatkov, ki ustrezajo predstavitvi v relacijski podatkovni bazi. Ker bo naša spletna aplikacija za leksikalno substitucijo uporabljala sistem MySQL, ki operira izključno z relacijskimi podatkovnimi bazami, je bila tovrstna transformacija zapisa podatkov nujno potrebna. Po opravljeni transformaciji nam je torej na voljo podatkovna baza WordNet v relacijski obliki.

3.5.2 Struktura podatkov WordNet v relacijski podatkovni bazi

Relacijska podatkovna baza WordNet vsebuje 19 tabel, ki sestojijo bodisi iz parov, ki povezujejo posamezne sklope sopomenk (takšnemu sklopu pravimo "synset," predstavlja pa nam skupino besed z enakim ali vsaj zelo podobnim pomenom) ali iz večih atributov, ki podajajo določeno dodatno informacijo o izbrani besedi.

Tabele relacijske podatkovne baze WordNet so sledeče:

- *antonym* - protipomenke;
- *attr_adj_noun* - pogosto povezani pridevniki;
- *cause* - X povzroča Y;
- *class_member* - X spada v pomenski razred Y;
- *derived* - besede s podobnim izvorom;
- *entails* - pogojenost X z Y;
- *gloss* - slovar s podrobnejšo razlago besed;
- *hypernym* - nadpomenke;
- *hyponym* - podpomenke;
- *mbr_meronym* - X opisuje člana celotnega Y;
- *participle* - deležniki;
- *part_meronym* - X opisuje delež celotnega Y;
- *pertainym* - besede s sorodnim pomenom;
- *see_also* - glej tudi;
- *similar* - podobne besede;
- *subst_meronym* - X opisuje substanco celotnega Y;
- *synset* - osnovni pomeni, določeni z besedo;
- *verb_frame* - kazalci glagolov na okvire;
- *verb_group* - skupine kazalcev na glagole.

3.5.3 Prilagoditev WordNetove strukture za praktično uporabo

Sistem za leksikalno analizo bo zelo pogosto izvajal isto fundamentalno operacijo - za dano besedo bo iskal čim večjo množico čim bolj ustreznih sopomenk. To pomeni, da je zasnova leksikalne podatkovne baze z 19 tabelami neoptimalna ter jo lahko nadalje optimiziramo v obliko, v kateri si bomo lahko za čim hitrejše delovanje pomagali z vgrajenimi funkcijami sistema MySQL.

Kot primer si oglejmo iskanje sopomenk za dano besedo (označeno s *$keyword*) v relacijski reprezentaciji podatkovne baze WordNet:

```
SELECT * FROM synset WHERE synset_id IN (
 SELECT synset_id_2 FROM similar WHERE synset_id_1 IN (
  SELECT synset_id FROM synset WHERE word = '{$keyword}'
 )
);
```

Slika 3.1: Trikratno gnezdena SQL poizvedba, ki nam vrne množico sopomenk za dano besedo.

Čeprav tabele podatkovne baze uporabljajo ustrezna kazala (*ang. index*) za pohitritev iskanja, je celotna poizvedba precej zahtevna in časovno potratna. Poizvedba na najvišjem nivoju v povprečju vzame 0,0011 sekunde časa na povprečnem sistemu, poizvedba na drugem nivoju vzame 3,2193 sekunde, poizvedba na najnižjem nivoju pa dodatne 1,6623 sekunde. Tako v povprečju poizvedba potroši približno 4,9 sekunde časa, kar je v praksi povsem nesprejemljivo.

Poizvedbo lahko seveda nekoliko optimiziramo in jo zapišemo v sledeči obliki:

```
SELECT SYN2.word
   FROM synset SYN1, synset SYN2, similar SIM
   WHERE SYN1.synset_id = SIM.synset_id_1
     AND SYN2.synset_id = SIM.synset_id_2
     AND SYN1.word = '{$keywords}';
```

Slika 3.2: Optimizirana SQL poizvedba, ki nam vrne množico sopomenk za dano besedo.

Takšna poizvedba v povprečju potrebuje le 0,0021 sekunde in se izkaže za mnogo učinkovitejšo. Vendar ne pozabimo, da smo za dano besedo z na sliki

3.2 prikazano poizvedbo preiskali le tabelo *similar*. Ustrezne sopomenke pa lahko iščemo tudi v tabelah *derived*, *hypernym*, *hyponym*, *pertainym*, *see_also* in tako dalje, kar ustrezno poveča število potrebnih poizvedb in tudi časovno zahtevnost same operacije iskanja sopomenk. Upoštevajmo še, da lahko z informacijami o logičnih relacijah, ki so opisane v tabelah *class_member*, *entails*, *mbr_meronym*, *part_meronym*, *subst_meronym*, *verb_frame* in *verb_group* ustvarimo boljši pregled nad tem, iz katerega dela podatkovne baze lahko učinkovito izvlečemo še dodatne sopomenke za dano besedo. Tako nam hitro postane jasno, da bo ob upoštevanju vsega naštetega potreben čas za iskanje sopomenk eksponentno narastel, ter da trenutna reprezentacija podatkov v relacijski podatkovni bazi z 19 tabelami ni dovolj učinkovita za potrebe našega sistema za leksikalno substitucijo.

Ob upoštevanju dejstva, da bi dobro sestavljena množica poizvedb za iskanje sopomenk pravzaprav vsakič iskala sopomenke po istih tabelah, upoštevala iste logične relacije med skupinami ter prišla do zelo podobnih zaključkov o sorodnosti posameznih besed, hitro pridemo do dejstva, da lahko vse naštete operacije nad leksikalno podatkovno bazo opravimo vnaprej za vsako izmed obstoječih besed ter ustvarimo podatkovno bazo z zgolj dvema učinkovitima tabelama, po katerih bo iskanje neprimerljivo bolj učinkovito.

Najprej ustvarimo osnovno tabelo "roots," ki vsebuje nabor pomenov, ter jo sestavljajo sledeči atributi:

- *id* - indeks zapisa o danem pomenu;
- *root* - beseda ali besedna zveza v osnovni obliki, ki nosi dani pomen;
- *pos* - besedna vrsta, v kateri dani pomen nastopa;
- *definition* - daljša razlaga danega pomena;
- *num_suggested* - kolikokrat je bil dani pomen predlagan uporabnikom;
- *num_accepted* - kolikokrat je bil dani pomen potrjen s strani uporabnikov kot ustrezen;
- *c_weight* - utež, izračunana na podlagi razmerja med prejšnjima dvema atributoma z uporabo formule *num_accepted*/*num_suggested*.

Atribut *id* nam predstavlja primarno kazalo (*ang. index*) tabele. Na atributu *root* ustvarimo dodatno kazalo, ki nam bo omogočalo hitro iskanje po podatkovni bazi, saj bo sistem MySQL zgradil razpršeno tabelo vseh vrednosti, ki se pojavljajo v atributu *word*. Posledično bo vsako iskanje določene besede v

podatkovni bazi izvedeno zelo hitro in naš sistem za leksikalno substitucijo bo dobil celotno informacijo o dani besedi v dovolj kratkem času, da bo leksikalna substitucija možna v realnem času.

Tabeli "roots" dodamo še tabelo "synonyms," ki vsebuje nabor sopomenk za vsakega izmed pomenov iz prejšnje tabele, ter jo sestavljajo sledeči atributi:

- *id* - indeks zapisa o dani sopomenki;
- *root_id* - indeks povezanega pomena iz tabele "roots;"
- *root* - sopomenka v osnovni obliki;
- *num_suggested* - kolikokrat je bila sopomenka predlagana uporabnikom;
- *num_accepted* - kolikokrat je bila sopomenka potrjena s strani uporabnikov kot ustrezna;
- *c_weight* - utež, izračunana na podlagi razmerja med prejšnjima dvema atributoma z uporabo formule *num_accepted*/*num_suggested*.

Atribut *id* nam tudi tokrat predstavlja primarno kazalo (*ang. index*) tabele. Na atributih *root_id* ter *root* ustvarimo dodatni kazali, ki nam zopet omogočata hitro iskanje po podatkovni bazi, kadar je eden od iskalnih kriterijev vezan na enega od teh atributov. Najpogostejši primer iskanja po tabeli "synonyms" bo iskanje ustreznih sopomenk za dani indeks prvotnega pomena iz tabele "roots." Ker ima tabela "synonyms" na atributu *root_id* ustvarjeno kazalo, bo poizvedba izvedena zelo hitro.

3.5.4 Ostale nujno potrebne tabele podatkovne baze

Za potrebe določanja pomena, v katerem se dana beseda ali besedna zveza trenutno nahaja, bomo za vsako besedo, ki lahko nosi več različnih pomenov, ustvarili tako imenovane kontekstne vektorje [13]. Vsak kontekstni vektor bo vseboval seznam besed v osnovni obliki, ki se pogosto pojavljajo v sobesedilu besede, kadar le-ta nosi enega od možnih pomenov.

Zamisel je relativno preprosta. Oglejmo si zopet sledečo poved:

John herded his sheep into a pen.

Omenili smo že, da se beseda "pen" lahko pojavlja kot nosilka vsaj dveh različnih pomenov - predstavlja nam lahko bodisi *ograd*o bodisi *pisalo*. Pri določanju pomena se bo bralec verjetno strinjal s sledečo predpostavko:

Beseda	Pomen	Kontekstni vektor
pen	ograda	cattle (živina), sheep (ovca), animal (žival), herd (gnati), saddled (osedlan)
pen	pisalo	story (zgodba), book (knjiga), paper (papir), write (pisati), writer (pisatelj)

S pomočjo premišljeno izgrajenih kontekstnih vektorjev za vsako izmed večpomenk bo sistem lahko zelo učinkovito razpoznaval dejanski pomen, ki ga opazovana beseda v danem trenutku nosi.

Za hranjenje kontekstnih vektorjev uporabimo tabelo "context," ki jo sestavljajo sledeči atributi:

- *id* - indeks zapisa o danem gradniku sobesedila;
- *root_id* - indeks povezanega pomena iz tabele "roots;"
- *root* - gradnik sobesedila (beseda, besedna zveza) v osnovni obliki;
- *num_found* - kolikokrat se dana beseda dejansko pojavi v sobesedilu povezanega pomena;
- *num_sought* - kolikokrat smo dano besedo iskali v sobesedilu povezanega pomena;
- *c_weight* - utež, izračunana na podlagi razmerja med prejšnjima dvema atributoma z uporabo spodaj opisane formule.

Zopet nam atribut *id* predstavlja primarno kazalo (*ang. index*) tabele, dodatno kazalo pa ustvarimo še na atributu *root_id*, saj bo najpogostejši primer iskanja po tabeli "context" iskanje ustreznih gradnikov sobesedila za dani indeks pomena iz tabele "roots."

Za izračun uteži *c_weight* posameznega gradnika sobesedila (elementa kontekstnega vektorja) uporabljamo sledečo formulo:

$$c\_weight = \frac{num\_found^{1.4}}{num\_sought}$$

Z uporabo zgornje preproste formule si zagotovimo, da je gradnik sobesedila, ki smo ga v stotih poskusih iskanja v sobesedilu trenutno opazovanega pomena našli petdesetkrat, ocenjen nekoliko bolje od gradnika sobesedila, ki smo ga v dveh poskusih iskanja našli le enkrat. Na ta način zmanjšamo tveganje, da bi sistem polagal preveliko težo na morebitne kontaminirane podatke ter posledično dobimo robustnejši sistem.

V nadaljevanju se bo izkazalo tudi, da ob lematizaciji potrebujemo podatek o nepravilnih glagolih, ki se pojavljajo v angleškem jeziku. Vse nepravilne glagole shranimo v tabelo s sledečo strukturo:

- *id* - indeks zapisa o dani trojki glagolov;
- *base* - glagol v sedanjiku;
- *past_simple* - glagol v pretekliku;
- *past_participle* - glagol v preteklem deležniku.

Na dani tabeli ustvarimo kazala na atributih *base*, *past_simple* in *past_participle*, saj bo sistem v okviru svojega delovanja moral pretvarjati glagole tako iz sedanjika v eno od preteklih oblik kot tudi obratno.

Na koncu se dotaknimo še tabele, v kateri bomo hranili vsa besedila, ki jih bodo uporabniki vnesli v sistem za leksikalno substitucijo. Tabela "structures" ima sledečo strukturo, v nadaljevanju pa bomo podrobneje spoznali, zakaj:

- *id* - indeks zapisa o obdelanem besedilu;
- *original_text* - prvotno besedilo;
- *data_content* - podatkovna struktura, ki predstavlja analizirano besedilo;
- *data_content_finished* - podatkovna struktura, ki predstavlja besedilo, obdelano s strani uporabnika;
- *finished_text* - končno besedilo, ki nastane kot rezultat leksikalne substitucije;
- *used_for_learning* - ali smo dano besedilo že uporabili za nadaljnjo izboljšavo sistema.

Atribut *id* nam tudi tokrat predstavlja primarno kazalo tabele.

3.5.5 Semantično smiselna interpretacija podatkov WordNet

Na tej točki moramo podatke iz obstoječe podatkovne baze WordNet v relacijski obliki semantično smiselno prenesti v našo novo - za potrebe sistema za leksikalno substitucijo močno optimizirano - podatkovno bazo.

Najprej iz WordNetove "synset" tabele razberemo vse osnovne pomene, ki se pojavljajo v angleškem jeziku, vključno z besedo v osnovni obliki, ki se pojavlja kot nosilka določenega pomena, ter besedno vrsto, v kateri dani pomen nastopa. V navezi s tabelo "gloss" vsakemu od osnovnih pomenov dodamo še podrobnejšo definicijo, ki jo bomo lahko ob uporabi sistema za leksikalno

substitucijo ponudili uporabniku, kadar sistem ne bo uspel samodejno izbrati ustreznega pomena. Tako smo uspešno prenesli podatke v novo tabelo "roots," ki smo jo podrobno predstavili nekoliko prej. Tabela še ne vsebuje uporabnih podatkov o tem, kako pogosto se v procesu leksikalne substitucije posamezen pomen dejansko izkaže za uporabnega, vendar bomo za to poskrbeli v nadaljevanju.

Opremljeni s tabelo vseh osnovnih pomenov "roots" se lahko lotimo izgradnje tabele "synonyms." Preko dobro definiranih relacij v podatkovni bazi WordNet izluščimo sopomenke iz obstoječe tabele "similar" ter v omejenem obsegu tudi iz tabel "see_also," "hyponym" in "hypernym."

Tokrat naša novo ustvarjena tabela "synonyms" že vsebuje nekaj podatkov o ustreznosti posameznih sopomenk v določenih kontekstih, saj smo si pomagali z informacijo o tem, v kateri tabeli smo našli posamezno sopomenko. Sopomenke iz tabele "similar" se namreč izkažejo za ustreznejše kandidate od sopomenk iz tabele "hypernym."

S pripravljenima tabelama "roots" in "synonyms" se lahko posvetimo izgradnji kontekstnih vektorjev za posamezne osnovne pomene. Pri tem si bomo pomagali z uporabo obstoječih WordNetovih relacij za določanje semantične sorodnosti konceptov [13]. V ta namen najprej ustvarimo začasno tabelo vseh besed, ki nosijo več kot en pomen. Nato se premikamo preko posameznih primerkov takšnih besed ter pri vsakem lematiziramo podrobnejšo definicijo pomena, ki ji vnaprej odstranimo besede, ki ne igrajo glavne vloge pri določanju pomena (vezniki, predlogi itd.). Na ta način pridobimo množico lem številnih besed, ki so semantično zelo sorodne trenutni besedi, kadar se pojavlja v danem pomenu.

Oglejmo si primer na že obravnavni besedi "pen" kot nosilki pomena *ograda*. WordNet definira ograd kot "an enclosure for confining livestock." Po izločitvi manj pomembnih besed ter lematizaciji preostalega dobimo sledeči seznam besed: "enclosure, confine, livestock."

Za vsako od semantično sorodnih besed nadalje raziščemo vse ustrezne tabele podatkovne baze WordNet:

- antonym;
- attr_adj_noun;
- cause;
- class_member;
- derived;

- entails;
- hypernym;
- hyponym;
- mbr_meronym;
- participle;
- part_meronym;
- pertainym;
- see_also;
- similar;
- subst_meronym;
- verb_group.

Na ta način pridobimo nov seznam sorodnih besed. V našem primeru to pomeni, da med drugim za besedo "enclosure" pridobimo sorodni besedi "pen" in "sty," za besedo "confine" pridobimo glagol "enclose," za besedo "livestock" pa besedi "sheep" in "cattle."

Za vsako besedo teh novo pridobljenih seznamov ponovimo postopek. V nabor semantično sorodnih besed torej najprej dodamo kar trenutno opazovano besedo, nato pa dodamo še množico lem, ki jo z opisanim postopkom pridobimo iz podrobnejše definicije trenutne besede.

Če se vrnemo k našemu primeru, lahko predpostavimo, da bomo v naslednjem koraku preko glagola "enclose," ki smo ga izpeljali iz glagola "confine" pridobili sorodni glagol "herd."

Po zadostnem številu korakov (v okviru diplomskega dela smo se odločili za dva koraka) pridobljeni seznam besed v osnovni obliki shranimo kot kontekstni vektor prvotnega pomena, pri čemer je vsaka od novo pridobljenih besed ustrezno utežena glede na sorodnost tabele podatkovne baze WordNet, v kateri smo jo našli.

Programsko kodo za določanje semantične sorodnosti konceptov si bralec lahko ogleda v dodatku A.

Na tej točki imamo za vsako besedo, ki se lahko pojavlja kot nosilka več različnih pomenov, izgrajen relativno kakovosten kontekstni vektor, torej seznam besed, ki se najpogosteje pojavljajo v sobesedilu, kadar opazovana beseda nastopa v danem pomenu. Vendar lahko z nekaterimi prijemi kontekstne vektorje še nadalje izboljšamo.

Poglejmo si spet primer, kjer beseda "pen" nastopa kot nosilka pomena *ograda*. Tabela sopomenk za besedo "pen" kot nosilko pomena *ograda* nam pove, da bi lahko besedo "pen" nadomestili z besedami "sty," "corral," "enclosure," "fence," "hutch" itd.

Po kratkem premisleku vidimo, da obstaja velika verjetnost, da se bodo v besedilu ob dani besedi torej pojavljale tudi sopomenke dane besede, saj sodijo na semantično zelo sorodno področje. Pisci besedil navadno načrtno uporabljajo sopomenke med samim ustvarjanjem besedila kot stilsko orodje, saj se s tem izognejo ponavljanju besed ter dosežejo lepšo berljivost besedila. Ob tem zaključku lahko torej kontekstnim vektorjem posameznih pomenov dodamo še sopomenke besed, ki se pojavljajo kot nosilke teh pomenov.

Poglavje 4

Strežniški del sistema za leksikalno substitucijo

Potek samega procesa leksikalne substitucije lahko smiselno razdelimo na del, ki se izvaja na strežniku, ter uporabniški vmesnik, ki domuje v uporabnikovem brskalniku.

Oglejmo si najprej potek leksikalne substitucije iz perspektive strežnika. Naloga strežnika je temeljita analiza vnešenega besedila, določitev pomenov posameznih gradnikov besedila, priprava seznamov ustreznih sopomenk, slovnična obdelava dokončanega besedila ter učenje iz uporabnikovega odziva.

S tem v mislih bomo za potrebe našega sistema za leksikalno substitucijo najprej razvili relativno preprost sistem za določanje besednih vrst. Pri tem ne bomo uporabljali markovskih verig, temveč poenostavljen sistem za iskanje po podatkovni bazi besed v osnovni obliki s pripadajočimi besednimi vrstami, ki bo v določeni meri upošteval tudi medsebojno odvisnost besednih vrst v posameznih povedih.

V ta namen moramo besedilo obdelati tako s semiotičnega kot tudi s semantičnega vidika.

4.1 Semiotična in semantična obdelava besedila

Semiotika je jezikoslovna veda, ki se ukvarja z jezikovnimi znaki (simboli) in sistemi znakov [14]. V nasprotju s semantiko se ukvarja z materialno platjo jezikovnega znaka, torej z njegovo obliko. V praksi to pomeni, da pri semiotični obdelavi besedila ne bomo pazili na dejanski pomen (nematerialno plat) jezika,

temveč se bomo posvetili predvsem sami strukturi besedila z materialne plati.

4.1.1 Razdelitev besedila na posamezne povedi

Besedilo moramo pred vsakršno nadaljnjo analizo (npr. določanjem besednih vrst) najprej razčleniti na posamezne povedi. Poved je definirana kot pomenska enota, ki je sestavljena iz enega ali več stavkov (vsak stavek, razen tako imenovanih pastavkov, vsebuje glagol) in jo predstavlja besedilo od velike začetnice do končnega ločila. Enostavčne povedi vsebujejo le en stavek, medtem ko večstavčne povedi vsebujejo več stavkov, ki so lahko povezani bodisi podredno, priredno ali soredno.

Besedilo na posamezne povedi razčlenimo z analizo uporabljenih znakov. Pristop ni povsem enostaven, saj ločila, ki se navadno pojavljajo kot končna ločila posameznih povedi, relativno pogosto nastopajo tudi v drugih vlogah. Tako se na primer pika kot ločilo pojavlja tudi pri okrajšavah, kraticah, decimalnih številih itd. Pozabiti ne smemo niti dejstva, da lahko ista pika hkrati nastopa kot uporabljeno ločilo pri okrajšavi ter kot končno ločilo, če je bila okrajšava uporabljena povsem na koncu določene povedi.

Potrebujemo torej postopek, ki nam omogoča učinkovito razdelitev besedila na posamezne povedi tudi v zahtevnih primerih, npr. v primeru s slike 4.1.

```
Mr John J. Johnson purchased the domain expensive.com for
7.5 million dollars, i.e. he spent way too much money on it.

But did he mind? It certainly seems that he didn't...At all!

This brings us to this conclusion ... he must be filthy rich!
Is he really? Hard to say; but it sure seems so.
```

Slika 4.1: Testno besedilo z velikim številom zahtevnih ločil.

Besedilo bomo torej razdelili na posamezne povedi s pomočjo upoštevanja pojavljanja velikih začetnic, ločil, prehodov v novo vrsto besedila in tako dalje. Opravila se lahko lotimo s sledečim postopkom:

Besedilu najprej povsem na začetek ter na konec dodamo znak za prehod v novo vrstico. S tem dosežemo, da se na obeh skrajnih točkah besedila nahaja znak, ki nas opozarja na prehod v novo poved.

Nato se premikamo preko posameznih znakov. Če naletimo na eno od možnih končnih ločil (pika, vprašaj, klicaj ter znak za novo vrstico), preverimo, če gre res za konec povedi. Če gre za piko, si ogledamo besedilo, ki se nahaja

med prejšnjim ločilom (tukaj upoštevamo tudi presledke) in opazovano piko. Če besedilo obstaja in je hkrati krajše od 2 znakov ali pa vsebuje le številke, potem nam pika ponazarja le prisotnost okrajšave, kratice ali decimalnega števila. Če piki neposredno sledi mala črka, potem ne gre za končno ločilo, temveč verjetno za spletni naslov (npr. *www.anze.eu*).

Če pika ne ustreza zgornjim kriterijem in se torej verjetno pojavlja kot končno ločilo ali če gre za eno izmed preostalih ločil, pa si ogledamo še besedilo, ki sledi danemu ločilu. Dokler znaki besedila sestojijo iz ločil, jih preprosto dodajamo trenutni povedi. Na ta način učinkovito obdelamo ločila kot npr. "..." in "?!" ter jih priredimo ustrezni povedi.

Ob končanem postopku si le še zabeležimo, če povedi sledi prehod v novo vrstico, da bomo kasneje lahko obnovili strukturo po odstavkih, ter poved shranimo.

Rezultat obdelave je prikazan na sliki 4.2.

```
[0] => Mr John J. Johnson purchased the domain expensive.com for 7.5
       million dollars, i.e. he spent way too much money on it.
[1] => But did he mind?
[2] => It certainly seems that he didn't...
[3] => At all!
[4] => This brings us to this conclusion ...
[5] => he must be filthy rich!
[6] => Is he really?
[7] => Hard to say; but it sure seems so.
```

Slika 4.2: Rezultat razdelitve besedila na povedi.

Programsko kodo funkcije *parseSentences()*, ki je bila spisana v namen razdelitve besedila na posamezne povedi, si lahko bralec ogleda v dodatku C.

4.1.2 Tokenizacija in standardizacija črkovanja in sloga

Ko imamo besedilo enkrat razdeljeno na povedi, moramo narediti še analizo po posameznih besedah. Pri tem si ob vsaki besedi zapomnimo še podatek o tem, če je beseda zapisana z veliko začetnico, če je v celoti zapisana z velimi črkami, če ji sledi ločilo in podobno.

Programsko kodo funkcije *tokenize()*, ki je bila spisana v ta namen, si lahko bralec prav tako ogleda v dodatku C.

4.1.3 Lematizacija

Z lematizacijo se premaknemo na področje semantične obdelave besedila.

Lematizacija v okviru lingvistike predstavlja proces združevanja različnih oblik določene besede z namenom, da lahko različne pojavne oblike besede hkrati obravnavamo kot eno besedo.

Lematizacija v okviru strojne obdelave naravnega jezika predstavlja algoritem, s katerim določamo *lemo* dane besede. Proces določanja ustrezne leme navadno vsebuje kompleksna opravila kot so razumevanje konteksta ter poznavanje pripadajoče besedne vrste, kar ga posledično napravi relativno zahtevnega za implementacijo.

V večini naravnih jezikov besede nastopajo v različnih pojavnih oblikah. Kot primer si lahko ogledamo glagol "hoditi," ki lahko nastopa v raznih pojavnih oblikah: hodim, hodita, hodijo, hodili (smo). Osnovna oblika tega glagola, torej beseda "hoditi," predstavlja *lemo* dane besede, ki bi jo npr. uporabili tudi za iskanje po slovarju. Kombinaciji leme ter podatka o besedni vrsti navadno rečemo leksem (*ang. lexeme*).

V okviru sistema za leksikalno subsitucijo bomo za potrebe diplomskega dela uporabili relativno preprost pristop k lematizaciji. Ker nam je na voljo podatkovna baza z besedami, ki se pojavljajo v angleškem jeziku, se bomo pri določanju lem posameznih besed večinoma zanašali na pomoč podatkovne baze.

Uporabili bomo tudi tabelo podatkovne baze, kamor smo shranili vse nepravilne glagole, ki se pojavljajo v angleškem jeziku. Vsak zapis v tabeli vsebuje atribute *base*, *past_simple* in *past_participle*, ki nam ponazarjajo glagol v sedanjiku, glagol v pretekliku ter glagol v preteklem deležniku.

Ob premikanju po posameznih gradnikih danega besedila (posameznih besedah) tako preverjamo prisotnost trenutne besede v podatkovni bazi besed, ki nastopajo v angleškem jeziku. Če besede v dani obliki ne najdemo, poskusimo z lematizacijo - besedi določimo najverjetnejšo osnovno obliko. To pomeni, da ji bodisi odstranimo končno črko "s" (s čimer morebitni samostalnik v množini spremenimo v samostalnik v ednini, morebitni glagol pa iz tretje osebe ednine postavimo v nedoločnik) bodisi glagolu odstranimo končni niz črk "ed" (s čimer ga iz preteklika postavimo v nedoločnik; pri spreminjanju glagolskega časa si pomagamo tudi s tabelo nepravilnih glagolov).

Glede na uspešnost oz. neuspešnost iskanja besede v podatkovni bazi angleških besed sklepamo na to, kdaj smo besedo postavili v pripadajočo osnovno obliko.

4.1.4 Določanje besednih vrst

Tudi pri določanju besednih vrst bomo za potrebe našega sistema za leksikalno substitucijo uporabili relativno enostaven pristop, ki nam ga omogoča dejstvo, da imamo vhodno besedilo po izvedenih prejšnjih korakih obdelave že razdeljeno na posamezne povedi, povedi so razdeljene na besede, besedam pa so določene najbolj verjetne osnovne oblike.

Pri pomikanju preko besed v izbrani povedi si zopet pomagamo s podatkovno bazo besed, ki nastopajo v angleškem jeziku, saj podatkovna baza vsebuje tudi podatek o tem, v kakšnih besednih vrstah posamezna beseda lahko nastopa. Če beseda nastopa le v eni besedni vrsti, si lahko označimo, da tudi v našem besedilu nastopa v tej besedni vrsti.

Če naletimo na besedo, ki se lahko pojavlja v različnih besednih vrstah, si naš algoritem pomaga s kontekstom danega stavka. Če beseda lahko nastopa bodisi kot samostalnik bodisi kot glagol, vendar smo v danem stavku glagol že nedvoumno določili, jo označimo kot samostalnik - tudi v ostalih primerih ravnamo s podobno logiko. Besede, katerim besedne vrste ne uspemo določiti, označimo kot samostalnike, s čimer navadno učinkovito pokrijemo lastna imena (Anže, Ljubljana itd.).

4.2 Iskanje ustreznih sopomenk za besede in besedne zveze

Na tej točki je besedilo razdelano tako s semiotičnega kot tudi s semantičnega vidika. Algoritem je tekom izvedbe posameznih korakov analize vhodnega besedila zgradil reprezentacijo besedila, ki celotno besedilo najprej razdeli na posamezne povedi, znotraj povedi pa na posamezne besede - gradnike besedila. Vsaka beseda vsebuje tudi podatek o svoji osnovni obliki (lemi) ter o najverjetnejši besedni vrsti, v kateri se v okviru dane povedi pojavlja.

Ko je besedilo obdelano do te mere, strežnik dobi nalogo, da besedam določi ustrezne pomene ter določi verjetnost ustreznosti posameznih pomenov. Prav tako mora strežnik generirati tudi sezname sopomenk za vsakega izmed pomenov po padajoči ustreznosti.

Proces lahko torej razdelimo na 2 dela: določanje ustreznega pomena vsakemu izmed gradnikov besedila ter sestavljanje seznama sopomenk v ustrezni obliki.

4.2.1 Določanje pomena gradnikom besedila

Pri določanju pomena posameznih gradnikov besedila se algoritem premika preko vseh povedi, na katere smo besedilo uspešno razčlenili.

Na nivoju vsake povedi izračuna kontekstni vektor, ki ga bo znotraj trenutne povedi uporabljal za določanje pomena pri tistih besedah, ki nastopajo kot nosilke več različnih pomenov. Pri izdelavi kontekstnega vektorja upoštevamo zgolj leme besed, ki navadno nosijo večji del pomena (samostalnike, glagole, pridevnike itd.), uporabljamo pa pristop, ki leme besed obteži glede na oddaljenost od trenutne povedi.

Koda za izdelavo kontekstnega vektorja je razvidna na sliki 4.3.

Oglejmo si izdelavo kontekstnega vektorja na primeru besedila "John herded his sheep into a pen." Ker sosednje povedi ne obstajajo, algoritem preišče le trenutno poved. Utež za leme, ki jih najdemo znotraj trenutne povedi, izračunamo s pomočjo sledeče formule:

$$(4 - \mathit{oddaljenost\ povedi})^3 = (4 - 0)^3 = 4^3 = 64$$

Ko je kontekstni vektor trenutne povedi uspešno izračunan, se algoritem premakne preko vseh besed znotraj povedi.

Pri vsaki izmed besed najprej izračunamo dodaten kontekstni vektor, ki vsebuje leme besed znotraj iste povedi v neposredni bližini trenutne besede. Tudi tokrat elemente kontekstnega vektorja utežimo glede na oddaljenost od trenutne besede. Tako ima v našem primeru pri obdelavi besede "pen" beseda "sheep" večjo težo od besede "herd," medtem ko beseda "John" ne spada več v dovolj bližnjo okolico ter sploh ni vključena v kontekstni vektor.

Ko sta nam na voljo tako kontekstni vektor na nivoju povedi kot kontekstni vektor na nivoju sosednjih besed, algoritem nadaljuje z delom. Najprej poskusi z grajenjem besedne zveze, ki se začenja s trenutno besedo. Tega se loti tako, da postopno lemi trenutne besede dodaja še leme besed, ki ji v besedilu neposredno sledijo, dokler celotna besedna zveza ne vsebuje 5 besed. Za vsako izmed tako pridobljenih besednih zvez preveri, če v podatkovni bazi zanjo obstaja definicija.

Če definicija obstaja in se posledično pridobljena besedna zveza pojavlja kot nosilka določenega pomena, algoritem analizira leme, ki jih je uporabil za izgradnjo nove besedne zveze. Z analizo uporabljenh lem lahko novo pridobljeni besedni zvezi določimo najverjetnejšo besedno vrsto, pri čemer upoštevamo hiearhijo dominantnosti posameznih besednih vrst.

Seznam besednih vrst po padajoči prioriteti izgleda takole: glagol, samostalnik, pridevnik, prislov, zaimek, predlog, veznik, medmet, členek.

```
// build a weighed context vector with relevant lemmas
$context_lemmas = array();
$omit_pos_types = array("conjunction", "determiner", "preposition");
for ($i = $sentence_key - 3; $i <= $sentence_key + 3; $i++) {
 // does this sentence actually exist?
 if (array_key_exists($i, $tagged_text)) {
  // calculate the weight of lemmas in this sentence
  $weight_of_lemmas = pow(4 - abs($i - $sentence_key), 3);
  foreach ($tagged_text[$i] as $tagged_word) {
   $pos = $tagged_word['pos'];
   $word_pos_simple = $converters[$pos];
   // does this lemma meet our criteria?
   if (
    strpos($stopwords, "\n{$tagged_word['lemma']}\n") === false &&
    !in_array($word_pos_simple, $omit_pos_types) &&
    strtolower($tagged_word['lemma']) != strtoupper($tagged_word['lemma'])
   ) {
    // this word meets our criteria
    $context_lemma_found = false;
    foreach ($context_lemmas as $cs_lemma_key => $cs_lemma) {
     if ($tagged_word['lemma'] == $cs_lemma['root']) {
      // we already have this lemma, let's just increase its weight
      $context_lemmas[$cs_lemma_key]['weight'] += $weight_of_lemmas;
      $context_lemma_found = true;
     }
    }
    if (!$context_lemma_found) {
     // we don't have this context lemma yet, let's add it
     $context_lemmas[] = array(
      "root" => $tagged_word['lemma'],
      "weight" => $weight_of_lemmas
     );
    }
   }
  }
 }
}
```

Slika 4.3: Koda za izgradnjo kontekstnega vektorja na nivoju posamezne povedi.

Oglejmo si pravkar opisani algoritem še na praktičnem primeru. Če novo pridobljeno besedno zvezo sestavljata pridevnik in samostalnik (npr. "short story"), ji določimo vlogo samostalnika, saj ima ta višjo prioriteto od pridevnika.

Na tem mestu podrobno analiziramo vse aktualne gradnike besedila, torej lemo trenutne besede ter vse morebitne novo pridobljene besedne zveze, ki smo jih uspeli izdelati začenši z lemo trenutne besede.

Zdaj za vsak gradnik preverimo seznam vseh pomenov v naši podatkovni bazi. Če gradnika ne najdemo, se z analizo premaknemo na naslednjo besedo znotraj trenutne povedi.

Če se gradnik na seznamu pomenov v podatkovni bazi pojavlja vsaj enkrat, vsakemu izmed možnih pomenov dodamo začasna podatka o prioriteti ter zanesljivosti, ki sta v začetku nastavljena na ničelno vrednost.

Oglejmo si to na primeru besede "pen" v povedi "John herded his sheep into a pen." Informacija o možnih pomenih, s katero operira naš sistem za leksikalno substitucijo, je razvidna iz naslednje tabele:

Beseda	**Besedna vrsta**	**Definicija**	**Prioriteta**	**Zanesljivost**
pen	samostalnik	ograda (His sheep are in the pen.)	0	0
pen	samostalnik	pisalo (He couldn't write without his pen.)	0	0
pen	glagol	pisati (He quickly penned a short note.)	0	0

Če se gradnik na seznamu pomenov pojavlja točno enkrat, smo zanj že našli ustrezen pomen. Prioriteto in zanesljivost mu nastavimo na 1000 točk, nakar zanesljivosti dodamo še akumulirano zanesljivost najdenega pomena v preteklosti. To pomeni, da določimo višjo zanesljivost gradnikom, ki so bili v preteklosti s strani uporabnikov večkrat označeni za pravilne ter uporabljeni pri določanju sopomenk.

Če se gradnik na seznamu pomenov pojavlja večkrat (torej lema trenutnega gradnika nosi več različnih pomenov, kot npr. v našem primeru), najprej za vsakega izmed najdenih pomenov preverimo ujemanje besedne vrste pomena z besedno vrsto našega gradnika. Če se besedni vrsti ujemata, prioriteto ter zanesljivost povečamo na zgoraj opisan način, sicer pa ju pustimo na ničelni vrednosti. Na tem mestu vse najdene pomene sortiramo po padajoči prioriteti, kar v praksi pomeni, da so tisti od pomenov, ki nastopajo v pravilni besedni vrsti, najvišje na seznamu ustreznih pomenov.

Ker v našem primeru besedna vrsta gradnika "pen" (samostalnik) ustreza dvema izmed možnih pomenov in ne ustreza tretjemu, je informacija sistema o možnih pomenih sledeča:

Beseda	Besedna vrsta	Definicija	Prioriteta	Zanesljivost
pen	samostalnik	ograda (His sheep are in the pen.)	1000	1023
pen	samostalnik	pisalo (He couldn't write without his pen.)	1000	1042
pen	glagol	pisati (He quickly penned a short note.)	0	0

Če sta v tej točki vsaj dva možna pomena po vrednosti atributa *prioriteta* izenačena na prvem mestu našega seznama, si bomo morali za določitev ustreznega pomena pomagati s kontekstnimi vektorji, ki smo jih pripravili vnaprej. Iz podatkovne baze najprej pridobimo obstoječi podatek o kontekstnem vektorju za vsakega izmed možnih pomenov, ki so izenačeni na prvem mestu našega seznama, ter ga primerjamo s kontekstnim vektorjem trenutnega gradnika besedila na nivoju povedi ter kontekstnim vektorjem trenutnega gradnika besedila na nivoju sosednjih besed.

Vsak izmed možnih pomenov dobi nov podatek o kontekstualni ustreznosti, ki ima v začetku ničelno vrednost. Za vsako ujemanje z obstoječim kontekstnim vektorjem trenutnega gradnika tej vrednosti dodamo zmnožek utežene vrednosti posameznega elementa kontekstnega vektorja trenutnega gradnika ter ocene zanesljivosti elementa kontekstnega vektorja možnega pomena iz podatkovne baze.

Poglejmo si zgoraj opisani postopek še na praktičnem primeru. Pri analizi besede "pen" sta nam na voljo dva kontekstna vektorja. Vsak element kontekstnega vektorja ima v oklepaju navedeno utež.

Kontekstni vektor na nivoju povedi: John (64), herd (64), sheep (64), pen (64)
Kontekstni vektor na nivoju sosednjih besed: herd (1), sheep (8)

Kontekstni vektor v uporabi: John (64), herd (65), sheep (72)

V podatkovni bazi imamo za pomen *ograda* shranjen sledeči kontekstni vektor, kjer je ob vsakem elementu vektorja navedena ocena zanesljivosti:

cattle (17), sheep (21), animal (7), herd (12), saddled (5)

Zaradi ujemanj v lemah "herd" (65 krat 12 točk) in "sheep" (72 krat 21 točk) pridobi pomen *ograda* 780 ter 1512 točk, skupno torej njegova vrednost podatka o kontekstualni ustreznosti znaša 2292 točk.

Pomen *pisalo* je v podatkovni bazi opremljen s sledečim kontekstnim vektorjem:

story (15), book (12), paper (6), write (23), writer (9)

Pri tem pomenu ne zaznamo nobenih ujemanj s kontekstnim vektorjem trenutno opazovane besede, kar pomeni, da vrednost podatka o kontekstualni ustreznosti ostane ničelna.

Ko imamo na ta način za vsakega izmed pomenov izračunan podatek o kontekstualni ustreznosti, podatke normaliziramo na interval od 0 do 500 ter dobljeno vrednost dodamo prioriteti pomena. Nato pomene zopet sortiramo po padajoči prioriteti, s čimer se tisti od pomenov, ki so kontekstualno ustreznejši, znajdejo na vrhu seznama ustreznih pomenov.

V našem primeru je informacija sistema o možnih pomenih v tem trenutku sledeča:

Beseda	**Besedna vrsta**	**Definicija**	**Prioriteta**	**Zanesljivost**
pen	samostalnik	ograda (His sheep are in the pen.)	1500	1077
pen	samostalnik	pisalo (He couldn't write without his pen.)	1000	1084
pen	glagol	pisati (He quickly penned a short note.)	0	0

Sedaj podatku o prioriteti posameznih možnih pomenov začasno dodamo še podatek o pretekli zanesljivosti uporabljenega pomena iz podatkovne baze, s čimer preferiramo pomene, ki so jih v preteklosti uporabniki pogosteje označili za pravilne ter so bili pogosteje uporabljeni za menjavo prvotnega gradnika s sopomenkami.

Če se znajdemo v situaciji, kjer sta dva pomena po vrednosti atributa *prioriteta* relativno izenačena navkljub preverjanju ustreznosti besedne vrste ter kontekstualne ustreznosti, v primeru zares velike razlike med preteklo zanesljivostjo posameznih pomenov preferiramo tistega, ki se je v preteklosti izkazal za zanesljivejšega. Posledično zanesljivejšemu pomenu povečamo podatek o prioriteti ter seznam možnih pomenov znova uredimo po padajoči prioriteti.

Na ta način za vsakega izmed gradnikov besedila poiščemo ustrezen pomen (če le-ta obstaja) ter izdelamo tudi seznam besednih zvez, ki se pojavljajo v besedilu in so nosilke pomena same po sebi. Besedilo je torej na tej točki opremljeno z informacijo o točnem pomenu velike večine gradnikov, kar pomeni, da lahko pričnemo s procesom izbiranja ustreznih sopomenk.

4.2.2 Sestavljanje seznama sopomenk za posamezne gradnike

Sistem je zdaj soočen z nalogo sestavljanja urejenih seznamov sopomenk v pravilni obliki za vsak možni pomen vsakega gradnika. To pomeni, da so vse sopomenke posameznega pomena urejene po padajoči oceni ustreznosti,

postavljene pa so v ustrezen čas in spregatev (v primeru glagola), število (v primeru samostalnika) itd. Pri ustvarjanju seznama sopomenk moramo prav tako upoštevati uporabo velikih in malih črk pri zapisu prvotnega gradnika.

Pri obdelavi gradnikov, ki jim je bila določena besedna vrsta *modalni glagol* (npr. beseda "can" v "John can win this game."), moramo še nadalje raziskati sobesedilo, da ugotovimo ustrezno spregatev glavnega glagola, ki se veže na modalni (pomožni) glagol. Določiti moramo namreč, če gre za glagol v prvi osebi (v neposrednem sobesedilu najdemo zaimek "jaz"), glagol v ednini (v neposrednem sobesedilu najdemo edninski zaimek ali samostalnik v ednini) oziroma glagol v množini (v neposrednem sobesedilu najdemo množinski zaimek ali samostalnik v množini oziroma večje število zaimkov in samostalnikov, npr. "John and I"). S pomočjo na ta način pridobljenega podatka lahko modalni glagol "can" zamenjamo s "has the ability to" oziroma "have the ability to" v pravilni obliki.

Pri obdelavi samostalnikov določimo, ali se samostalnik pojavlja v ednini ali množini. Pri obdelavi glagolov razlikujemo med osnovno obliko glagola v prvi osebi, osnovno obliko glagola v ednini, osnovno obliko glagola v množini, preteklikom, preteklim deležnikom in gerundijem.

Zapis z ozirom na uporabo velikih in malih črk določimo z uporabo naslednjih dveh pravil. Če prvotni gradnik uporablja veliko začetnico (torej je prvi znak gradnika ena od velikih črk), bomo sopomenke prav tako zapisovali z veliko začetnico. Nadalje, če je zadnji znak gradnika prav tako ena od velikih črk, bomo sopomenke zapisovali v celoti z uporabo velikih črk.

Zdaj iz podatkovne baze pridobimo seznam vseh sopomenk ter jih uredimo po pretekli zanesljivosti, torej po vrednosti že omenjenega atributa *c_weight*. Za vsako izmed sopomenk najprej preverimo, če bomo z njeno uporabo podvojili eno od sosednjih besed (besede "can" v "John can probably win this game." ne želimo nadomestiti s "can probably") in jo po potrebi zavržemo.

Nato z ozirom na obliko prvotnega gradnika prilagodimo vsako od sopomenk. Programska koda, s katero dosežemo primerno prilagoditev, je prikazana v dodatku B.

Ustrezno preoblikovane sopomenke vključno s podatkom o zgodovinski zanesljivosti dodamo na seznam sopomenk trenutnega pomena.

4.3 Končna obdelava besedila

Na tej točki sistem vstopi v fazo končne obdelave besedila za prikaz uporabniku. Zaradi implementacije pristopa k zamenjevanju osnovnih gradnikov

besedila z ustreznimi sopomenkami v okviru para zavitih oklepajev, moramo besedilo po opravljeni leksikalni substituciji zaradi slovnične pravilnosti še enkrat obdelati z vidika uporabe nedoločnih členov "a" oziroma "an."

Še enkrat si torej oglejmo že uveljavljeni primer besedila po opravljeni leksikalni substituciji:

$$John\ herded\ his\ sheep\ into\ a\ \{pen \mid enclosure\}.$$

Kot je razvidno že na prvi pogled, lahko z uporabo sintakse za zapis množic ustreznih sopomenk sistem ustvari besedilo "John herded his sheep into a enclosure," ki je pomensko sicer enakovredno prvotnemu besedilu, vendar pa je slovnično nepravilno, saj napačno uporablja nedoločni člen "a" namesto nedoločnega člena "an."

Ker sistem omogoča uporabo poljubno gnezdene sintakse za navajanje množic ustreznih sopomenk, postane problem pravilnega določanja nedoločnih členov relativno zahteven. Algoritem se problema loti s klicem funkcije *fixIndefiniteArticles()* na sledeči način:

- besedilo najprej razdeli po zavitem oklepaju, zavitem zaklepaju ter navpični črti;
- vsakemu od tako pridobljenih delov besedila določi številsko vrednost nivoja gnezdene sintakse, na katerem se besedilo nahaja (0, 1, 2, ...);
- vsakič, ko ob prehodu preko delov besedila naleti na nedoločni člen, ki mu sledi del besedila z višjo številsko vrednostjo nivoja gnezdene sintakse, si zapomni uporabo velike začetnice za zapis nedoločnega člena;
- nedoločni člen odstrani s trenutnega mesta ter kliče rekurzivno funkcijo *fixIndefiniteArticlesR()*, ki bo nedoločni člen postavila na pravo mesto.

Rekurzivna funkcija *fixIndefiniteArticlesR()* sprejme podatek o trenutni poziciji v besedilu, trenutnem nivoju gnezdenja ter zapisu nedoločnega člena z uporabo velike začetnice. Nato nadaljuje z delom na sledeči način:

- premika se po delih besedila, dokler ne naleti na del besedila z nižjo številsko vrednostjo nivoja gnezdenja;
- če množica ustreznih sopomenk ne vsebuje nadaljnjega gnezdenja, vsakemu od elementov množice določi ustrezen nedoločni člen glede na prvo črko elementa, pri čemer upošteva tudi izjeme, kot sta "a unicorn" in "an hour;"

- če množica vsebuje nadaljnje primere gnezdenja, se na globljem nivoju zopet kliče rekurzivna funkcija *fixIndefiniteArticlesR()*.

Algoritem torej v tej fazi odpravi še slovnične napake, ki so se prikradle v besedilo tekom same obdelave s strani sistema za leksikalno substitucijo, nakar zopet posodobi zapis v tabeli "structures" in v polje *finished_text* shrani končno verzijo obdelanega besedila.

4.4 Uporabniški odziv ter učenje iz odziva

Pri učenju, cilj katerega je nadaljna izboljšava sistema za leksikalno substitucijo, je sistemu na voljo obdelana podatkovna struktura, ki predstavlja analizirano prvotno besedilo z naborom vseh možnih pomenov ter predlaganih sopomenk, vključno z informacijo o tem, kateri pomeni so bili s strani uporabnika označeni za pravilne ter katere sopomenke so bile dejansko uporabljene v procesu leksikalne substitucije.

4.4.1 Izboljševanje informacije o možnih pomenih besed

Sistem zopet prične s premikanjem po vseh gradnikih prvotnega besedila, ki jim je bil določen vsaj en možen pomen. Pri vsakem gradniku najprej preveri, kateri od možnih pomenov je bil dejansko izbran s strani uporabnika za pravilnega. Pomen se smatra za pravilnega le v primeru, da je bila v besedilu uporabljena vsaj ena njegova sopomenka, sicer ga smatramo kot nepravilnega oziroma - bolje rečeno - neuporabljenega.

Za vsak pomen iz podatkovne baze pridobimo podatek *num_suggested* o tem, kolikokrat je bil pomen v preteklosti predlagan uporabnikom, ter podatek *num_accepted* o tem, kolikokrat so ga uporabniki dejansko uporabili. Oba podatka ustrezno posodobimo (vrednost *num_suggested* v vsakem primeru povečamo za 1) ter ju skupaj z novo izračunano vrednostjo *c_weight* shranimo nazaj v podatkovno bazo k ustreznemu pomenu. Na ta način sistem kontinuirano izboljšuje svojo informacijo o splošni ustreznosti posameznih pomenov.

V primeru, da je bil trenutni pomen s strani uporabnika označen za pravilnega ter gradnik, na katerega je bil vezan, lahko nosi tudi kakšen drug pomen, seveda posodobimo še kontekstni vektor trenutnega pomena.

Na praktičnem primeru si lahko predstavljamo, da je uporabnik pri besedi "pen" izbral pomen *ograda* ter uporabil sopomenke "sty," "corral" in "enclosure." Na ta način smo lahko prepričani, da je bil besedi "pen" določen pravi

pomen, kar pomeni, da lahko obdelamo sobesedilo in posodobimo kontekstni vektor izbranega pomena v podatkovni bazi. Posledično bo sistemu prihodnjič, ko bo preučeval kontekst besede "pen," na voljo popolnejša informacija o verjetnem kontekstu, kadar beseda nastopa kot nosilka pomena *ograda*.

To storimo tako, da zopet izdelamo utežen kontekstni vektor na podlagi sobesedila (na podlagi lem besed, ki so vsebovane v bližnjih povedih, pri čemer uporabimo le besede, ki navadno nosijo večji del pomena) ter ga primerjamo s kontekstnim vektorjem trenutnega pomena v podatkovni bazi. Za vsako ujemanje izboljšamo oceno elementov kontekstnega vektorja v podatkovni bazi, za vsako neujemanje jo znižamo, obenem pa ustvarimo tudi seznam še neobstoječih elementov kontekstnega vektorja, ki so se v trenutnem sobesedilu trenutnega pomena pojavili prvič.

Teh elementov ne shranimo neposredno v podatkovno bazo kot elemente kontekstnega vektorja trenutnega pomena, temveč jih začasno shranimo v posebno tabelo podatkovne baze vključno s podatkom, da so bili v kontekstu trenutnega pomena najdeni enkrat. Tako lahko v primeru, da se v kontekstu trenutnega pomena tudi v prihodnje pojavljajo enake leme, te leme dokončno dodamo v primarni kontekstni vektor pomena v podatkovni bazi. S pomočjo na ta način določene omejitve za vključitev besed v primarni kontekstni vektor si zagotovimo, da bodo v kontekstnem vektorju vedno vključene le pomensko sorodne besede.

4.4.2 Izboljševanje informacije o možnih sopomenkah besed

Sedaj se sistem sprehodi še preko vseh izbranih sopomenk različnih pomenov. Ker uporabniški vmesnik omogoča vnos novih sopomenk (ki torej v podatkovni bazi še ne obstajajo), najprej poišče takšne sopomenke, ki so ustrezno označene z odsotnostjo kazalca na zapis v podatkovni bazi. Sveže dodane sopomenke najprej postavi v osnovno obliko (torej jih lematizira), nakar jih shrani kot sopomenke k ustreznemu pomenu.

Nato se algoritem sprehodi še preko sopomenk, ki so že prisotne v sistemu, kar vidimo po tem, da so opremljene s kazalcem na zapis v podatkovni bazi. Vsaki izmed njih ustrezno poveča atribute *num_suggested*, *num_accepted* in *c_weight* ter posodobi zapis v podatkovni bazi. Na ta način sistem preko učenja iz uporabnikovega odziva pridobiva vedno boljšo informacijo o pretekli zanesljivosti sopomenk posameznih pomenov.

Ko so na ta način obdelane vse sopomenke, sistem označi strukturo trenutnega besedila kot obdelano in zaključi s procesom učenja.

Poglavje 5

Uporabniški vmesnik sistema za leksikalno substitucijo

Oglejmo si potek procesa leksikalne substitucije še z vidika uporabnika, ki je v svojem brskalniku zagnal spletno aplikacijo za leksikalno substitucijo.

Sam proces leksikalne substitucije lahko s tega vidika smiselno razdelimo na tri ločene faze.

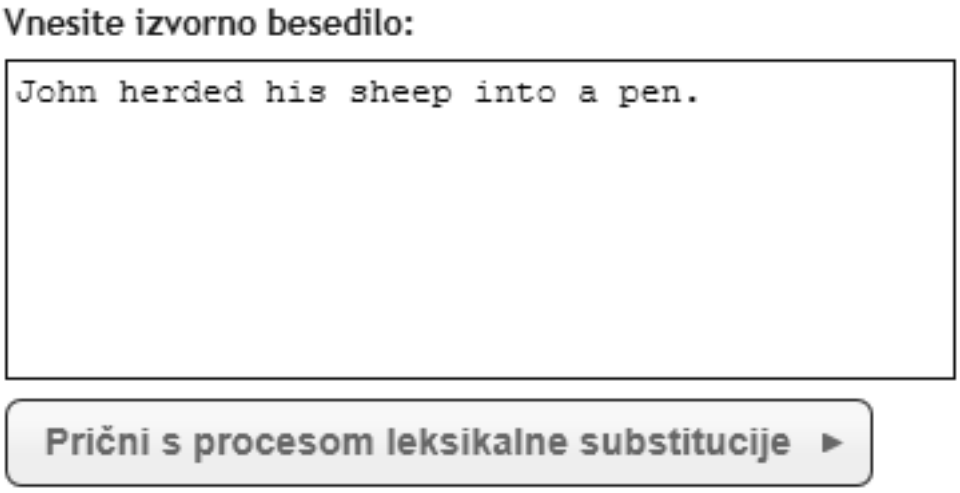
5.1 Analiza besedila

V prvem koraku uporabniku ponudimo spletno stran z vnosnim poljem za besedilo. Po vnosu besedila uporabnik pritisne gumb, s katerim sproži prehod v naslednjo fazo procesa leksikalne substitucije. Celoten uporabniški vmesnik prvega koraka je prikazan na sliki 5.1.

Ob kliku na gumb se s pomočjo tehnologije Ajax na strežnik asinhrono pošlje zahtevek za izvedbo procesa določanja besednih vrst na vnešenem besedilu. V podatkovni bazi se ustvari nov zapis znotraj tabele "structures," ki v polju *original_text* hrani pravkar vnešeno prvotno besedilo.

Ko strežnik zaključi s procesom določanja besednih vrst, shrani PHP podatkovno strukturo, ki predstavlja označeno prvotno besedilo, v podatkovno bazo in sicer v polje *data_content* sveže ustvarjenega zapisa iz tabele "structures." Nato odgovori prejetemu zahtevku s sporočilom, da je nalogo uspešno opravil.

Uporabnikov brskalnik prejme sporočilo o uspešno opravljeni nalogi ter uporabnika obvesti, da je bila prva faza analize besedila uspešno zaključena. Istočasno sproži nov asinhroni zahtevek, s katerim naroči strežniku, naj besedam (gradnikom besedila) določi najverjetnejši pomen ter za vsakega izmed



Slika 5.1: Uporabniški vmesnik prvega koraka procesa leksikalne substitucije.

možnih pomenov ustvari seznam sopomenk, ki naj bo razvrščen od najbolj primernih do najmanj primernih kandidatov.

Strežnik navedeno nalogo opravi (točen potek tega dela algoritma je bil predstavljen v prejšnjem poglavju) ter posodobi PHP podatkovno strukturo, shranjeno v polju *data_content*, ki zdaj vključuje tudi sezname pomenov ter sopomenk za vsako izmed prvotnih besed. Nato zopet odgovori prejetemu zahtevku s sporočilom, da je nalogo uspešno opravil.

Uporabnikov brskalnik je seznanjen s pozitivnim izidom obeh faz prvotne analize besedila ter uporabnika preusmeri na novo spletno stran, ki mu bo omogočila urejanje ravnokar analiziranega besedila.

5.2 Določanje ustreznih sopomenk s strani uporabnika

Ko uporabnik prispe na spletno stran za urejanje analiziranega besedila, se takoj sproži asinhroni zahtevek, ki od serverja zahteva podatkovno strukturo, ki vsebuje podatke o besedilu, možnih pomenih posameznih besed ter seznamih sopomenk za posamezne pomene. Strežnik strukturo prevede v JSON format ter jo v tekstovni obliki (stisnjeno s programom GZIP) pošlje uporabnikovemu brskalniku.

Brskalnik iz dobljenega besedila, ki predstavlja veljaven JSON zapis podatkovne strukture, ustvari podatkovno strukturo za uporabo v skriptnem jeziku JavaScript. Zasnova strukture je razvidna s slike 5.2.

Ko je struktura pripravljena, jo z uporabo istega skriptnega jezika analiziramo ter ustvarimo verno HTML reprezentacijo prvotnega besedila. Na ta način je uporabniku besedilo prikazano v svoji prvotni obliki, vendar je opremljeno s HTML oznakami, ki so nam pri kasnejši obdelavi besedila močno v pomoč.

Na tej točki lahko uporabnik prične s procesom leksikalne substitucije. K problemu lahko pristopi na dva načina:

- *samodejna leksikalna substitucija* - s klikom na gumb sistem samodejno uporabi najbolj verjetne pomene posameznih besed ter najbolj primerne kandidate za sopomenke;
- *ročna leksikalna substitucija* - z izbiro posamezne besede ali besedne zveze uporabnik sam določi pomen ter ustrezne sopomenke.

V primeru, da se uporabnik odloči za samodejno leksikalno substitucijo, aplikacija v skriptnem jeziku JavaScript stori naslednje:

- sprehodi se preko vseh besed, ki sestavljajo besedilo;
- za dano besedo preveri, če ji je z dovolj veliko zanesljivostjo določen ustrezen pomen;
- če za dano besedo obstaja dovolj zanesljiv pomen, se sprehodi preko vseh možnih sopomenk za dani pomen;
- za vsako sopomenko preveri oceno pretekle zanesljivosti in aktivira tiste, ki presegajo določen prag;
- posodobi uporabniški vmesnik, da le-ta odraža novo stanje.

Ko je delo opravljeno, je uporabnik obveščen o uspešno izvedenem procesu leksikalne substitucije in se lahko odloči za prehod v zadnji korak obdelave besedila. Po potrebi lahko tudi nadaljuje z ročnim urejanjem besedila, torej z ročno leksikalno substitucijo.

V primeru, da se uporabnik odloči za ročno leksikalno substitucijo, lahko izbere vsako besedo ali besedno zvezo v besedilu tako, da jo označi z uporabo miške. Brskalnik preko skriptnega jezika JavaScript ta dogodek zazna, v podatkovni strukturi poišče vse možne pomene za izbrani del besedila ter

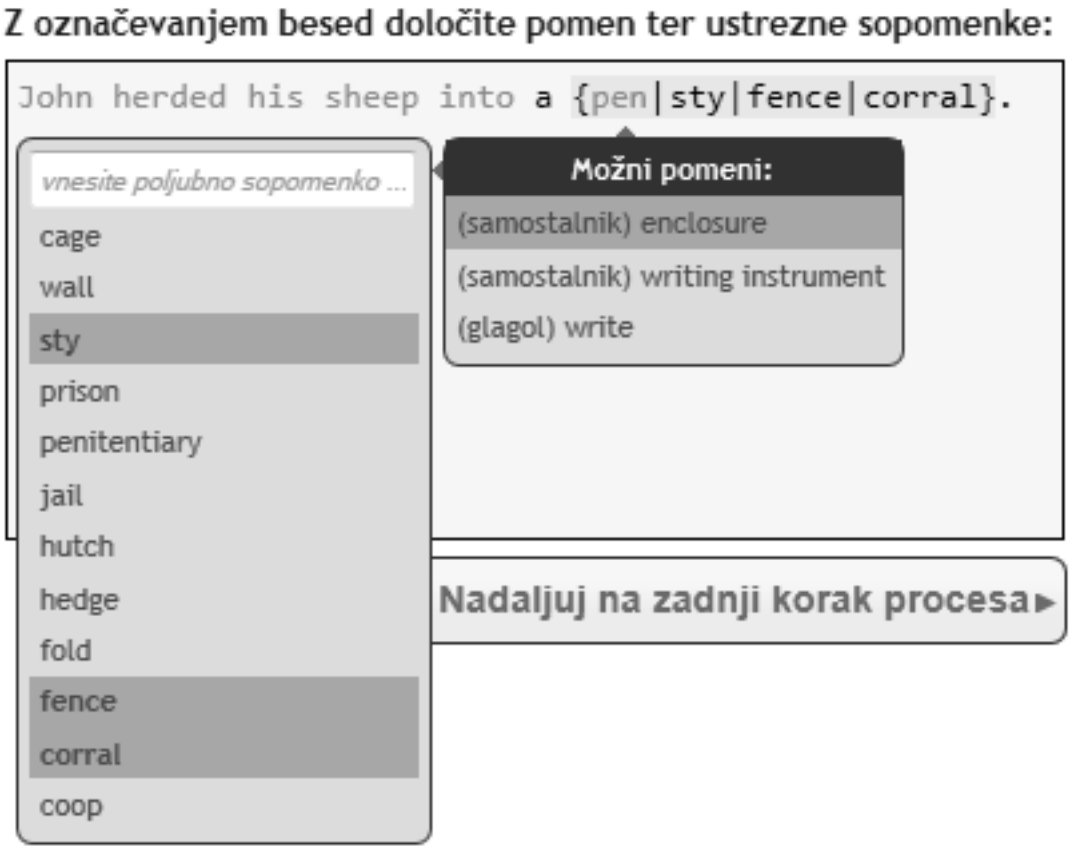
```
status: "OK"
id: kazalec na trenutno besedilo v podatkovni bazi
text: {
  (id prve besede, dejanska beseda, način zapisa),
  (id druge besede, dejanska beseda, način zapisa),
  ...
}
suggestions: {
 (
  id besede,
  seznam možnih pomenov: {
   (
    kazalec na trenutni pomen v podatkovni bazi,
    podrobna definicija pomena za prikaz uporabniku,
    ocena zanesljivosti trenutnega pomena,
    izbranost trenutnega pomena (DA ali NE),
    seznam sopomenk za trenutni pomen: {
     (
      kazalec na trenutno sopomenko v podatkovni bazi
      dejanska sopomenka
      ocena zanesljivosti trenutne sopomenke
      izbranost trenutne sopomenke (DA ali NE)
     )
     ...
    }
   )
  ...
 }
 ...
}
```

Slika 5.2: Zasnova podatkovne strukture analiziranega besedila.

uporabniku prikaže nov element uporabniškega vmesnika, kjer so možni pomeni navedeni po padajoči oceni ustreznosti. Vsak od pomenov je opremljen tudi s seznamom ustreznih sopomenk, ki je prav tako urejen po padajoči oceni zanesljivosti posameznih sopomenk.

Na tem mestu lahko uporabnik izbere pravilni pomen izbranega dela besedila ter aktivira oziroma deaktivira posamezne sopomenke. Ob vsaki akciji uporabnika se uporabniški vmesnik posodobi, da verno odraža trenutno stanje.

Uporabniški vmesnik drugega koraka procesa je razviden s slike 5.3.



Slika 5.3: Uporabniški vmesnik drugega koraka procesa leksikalne substitucije.

Uporabnik lahko posameznemu pomenu trenutno izbranega dela besedila po lastnih željah doda tudi nove sopomenke, ki se ustrezno shranijo v podatkovno strukturo, ki nam predstavlja analizirano trenutno besedilo.

Ko uporabnik konča z delom, s klikom na gumb sproži prehod v zadnji korak leksikalne substitucije. Na tem mestu brskalnik s pomočjo skriptnega jezika JavaScript stori sledeče:

- iz podatkovne strukture odstrani odvečne podatke (definicije, ocene ustreznosti itd.);
- okleščeno podatkovno strukturo pretvori v format JSON;
- tekstovno reprezentacijo strukture v formatu JSON stisne z algoritmom LZW [15];
- na ta način stisnjeno podatkovno strukturo pošlje strežniku.

5.3 Končna obdelava besedila

Ko strežnik prejme na ta način obdelano podatkovno strukturo, jo najprej ekstrahira s pomočjo LZW algoritma, nakar jo iz dobljene tekstovne reprezentacije v formatu JSON pretvori v uporabno PHP podatkovno strukturo. Na tej točki jo shrani v podatkovno bazo k ustreznemu zapisu iz tabele "structures" v polje *data_finished*.

Zavedati se je treba, da imamo sedaj na strežniku shranjeno tako podatkovno strukturo, ki nam predstavlja začetno stanje analiziranega besedila, ter podatkovno strukturo, ki nam predstavlja dokončano besedilo po izvedenem procesu leksikalne substitucije, nadzorovanem s strani uporabnika. Slednjo podatkovno strukturo sistem za leksikalno substitucijo uporablja v procesu učenja, saj se lahko z njeno pomočjo nadalje izpopolni ter pridobi nove informacije o ustreznosti posameznih pomenov ter sopomenk v različnih kontekstih.

Na tej točki vstopimo v fazo končne obdelave besedila za prikaz uporabniku, saj besedilo potrebuje še nekaj popravkov na področju uporabe nedoločnih členov, kar smo podrobneje razdelali v prejšnjem poglavju.

Sistem torej v tej fazi s pomočjo v poglavju 4.3 opisanega algoritma odpravi še slovnične napake, ki so se prikradle v besedilo tekom same obdelave. Nato odgovori na zahtevek, ki ga je prejel od uporabnikovega brskalnika, ter s tem sproži prehod uporabnika na novo spletno stran, kjer ga čaka končano besedilo.

Poglavje 6

Sklepne ugotovitve in ideje za nadaljnje delo

V diplomskem delu je opisan razvoj spletne aplikacije za reševanje problema leksikalne substitucije. Spletno aplikacijo sestavlja pregleden ter učinkovit uporabniški vmesnik, katerega implementacija je bila zaradi količine obdelovanih podatkov (pri obdelavi povprečnega besedila v skriptnem jeziku JavaScript operiramo z nekaj megabajtov velikimi podatkovnimi strukturami) relativno zapletena ter strežniški del, ki s pomočjo splošno sprejetih pristopov na področju obdelave naravnega jezika poskuša določiti dejanski pomen posameznih gradnikov besedila ter ponuditi čim bolj relevantne sopomenke.

Končno delovanje spletne aplikacije za leksikalno substitucijo je nadvse zadovoljivo, seveda pa obstaja še prostor za nadaljnje izboljšave.

V primeru, da bi bila spletna aplikacija na voljo uporabnikom, za katere ne moremo biti prepričani, da jo bodo uporabljali v skladu z našimi pričakovanji, bi bila potrebna implementacija dodatnih algoritmov za zaščito obstoječega nabora podatkov o pomenih ter ustreznih sopomenkah posameznih pomenov. S tem mislim predvsem, da bi moral sistem vsako novo vnešeno sopomenko določenega pomena shraniti v začasno tabelo podatkovne baze ter jo dokončno dodati na nabor primernih sopomenk šele, ko bi jo neodvisno drug od drugega predlagalo dovolj veliko (vnaprej določeno) število uporabnikov. Enako velja za posodabljanje informacije o kontekstnih vektorjih posameznih pomenov - gradnike sobesedila bi dodali v primarni kontekstni vektor pomena šele, ko bi večja množica uporabnikov potrdila njihovo ustreznost.

V skladu z možnostjo uporabe sistema za leksikalno substitucijo za namene čim boljše interpretacije vnešenega iskalnega niza, ki smo jo opisali v samem uvodu v pričujoče diplomsko delo, bi veljalo razmisliti tudi o izdelavi

vmesnika za programiranje (*ang. application programming interface*, s kratico API), preko katerega bi lahko ostali razvijalci programske opreme dostopali do sistema za leksikalno substitucijo mimo obstoječega uporabniškega vmesnika, ki je predviden predvsem za ročno uporabo sistema.

Problem leksikalne substitucije se je sicer izkazal za izredno zahtevnega, saj je področje obdelave naravnega jezika še vedno do neke mere neraziskano, vendar sem z razvitim sistemom za leksikalno substitucijo zelo zadovoljen in verjamem, da se s problemom spopada z zadovoljivo učinkovitostjo ter dosega dobre rezultate v doglednem času.

Dodatek A

Semantična sorodnost besed

```
<?php
require_once("../_includes/initialization.php");
require_once("anzeLemmatizer.php");
require_once("stopwords.php");

$lemmatizer = new anzeLemmatizer();

$counter = 0;
$result_homographs = mysql_query("SELECT * FROM homographs ORDER BY id ASC;");
while ($line_homograph = mysql_fetch_array($result_homographs)) {

 $root_esc = escape_string($line_homograph['root']);
 $result_roots = mysql_query("SELECT * FROM roots WHERE root = '{$root_esc}';");

 while ($l_root = mysql_fetch_array($result_roots)) {

  // process the definition of the word
  $mpl = 50;
  $lemmas = array();
  $words = $lemmatizer->lemmatizeText($l_root['definition']);
  foreach ($words as $word) {
   $lemmas[] = $word['lemma'];
   $lemmas[] = $word['lemma'];
  }
  updateContextData($l_root, $lemmas, $mpl);

  // process WordNet data for this word
  $result = mysql_query("SELECT * FROM root_synset WHERE root_id = {$l_root['id']};");
  while ($line = mysql_fetch_array($result)) {

   // process WordNet antonyms
   $mpl = 10;
   processWordNetData($l_root, $line, "antonyms", "antonym", true, $mpl);

   // process WordNet attributes & adjectives & nouns
   $mpl = 20;
   processWordNetData(
    $l_root,
    $line,
    "attributes & adjectives & nouns",
```

```
 "attr_adj_noun",
 false,
 $mpl
);

// process WordNet causes
$mpl = 15;
processWordNetData($l_root, $line, "causes", "cause", false, $mpl);

// process WordNet classes
$mpl = 15;
processWordNetData($l_root, $line, "classes", "class_member", false, $mpl);

// process WordNet derived words
$mpl = 30;
processWordNetData($l_root, $line, "derived", "derived", true, $mpl);

// process WordNet entails
$mpl = 20;
processWordNetData($l_root, $line, "entails", "entails", false, $mpl);

// process WordNet hypernyms
$mpl = 35;
processWordNetData($l_root, $line, "hypernyms", "hypernym", false, $mpl);

// process WordNet hyponyms
$mpl = 35;
processWordNetData($l_root, $line, "hyponyms", "hyponym", false, $mpl);

// process WordNet meronyms
$mpl = 30;
processWordNetData($l_root, $line, "meronyms", "mbr_meronym", false, $mpl);

// process WordNet participles
$mpl = 10;
processWordNetData($l_root, $line, "participles", "participle", true, $mpl);

// process WordNet partial meronyms
$mpl = 30;
processWordNetData($l_root, $line, "part. meronyms", "part_meronym", false, $mpl);

// process WordNet pertainyms
$mpl = 10;
processWordNetData($l_root, $line, "pertainyms", "pertainym", true, $mpl);

// process WordNet see-also
$mpl = 30;
processWordNetData($l_root, $line, "see-also", "see_also", true, $mpl);

// process WordNet similar words
$mpl = 15;
processWordNetData($l_root, $line, "similar words", "similar", false, $mpl);

// process WordNet substitute meronyms
$mpl = 30;
processWordNetData($l_root, $line, "subst. meronyms", "subst_meronym", false, $mpl);

// process WordNet verb groups
$mpl = 15;
```

```
   processWordNetData($l_root, $line, "verb groups", "verb_group", false, $mpl);
  }

  $context = "";
  $result_context_vector = mysql_query("SELECT * FROM context
   WHERE root_id = {$l_root['id']} ORDER BY c_weight DESC;");
  while ($lcv = mysql_fetch_array($result_context_vector)) {
   $context = $context . $lcv['root'] . " (" . round($lcv['c_weight'], 4) . "), ";
  }
  if (substr($context, -2) == ", ") $context = substr($context, 0, strlen($context) - 2);
  echo "Context data: {$context}<br /><br />";

 }

}

/**
 * processes all available (and useful) WordNet data for a given synset (meaning)
 * @param $l_root
 * @param $line
 * @param $name
 * @param $table_name
 * @param $incl_w_num
 * @param $mpl
 */
function processWordNetData($l_root, $line, $name, $table_name, $incl_w_num = false, $mpl = 1) {
 $result_wn = mysql_query("SELECT * FROM {$table_name}
  WHERE synset_id_1 = {$line['synset_id']};");
 $num_wn = mysql_num_rows($result_wn);
 if ($num_wn > 0) {
  $synset_ids = array();
  while ($line_wn = mysql_fetch_array($result_wn)) {
   if ($incl_w_num) {
    $synset_ids[] = array(
     "id" => $line_wn['synset_id_2'],
     "w_num" => $line_wn['wnum_2']
   );
   } else {
    $synset_ids[] = $line_wn['synset_id_2'];
   }

  }
  $lemmas = mineWordNet($synset_ids, $l_root['root']);
  $context = updateContextData($l_root, $lemmas, $mpl);
 }
}

/**
 * mines the WordNet database for relevant lemmas
 * @param array $synset_ids
 * @param string $original_word
 * @return array
 */
function mineWordNet($synset_ids, $original_word) {
 global $lemmatizer;
 $lemmas = array();
 if (!is_array($synset_ids)) $synset_ids[0] = $synset_ids;
 foreach ($synset_ids as $synset_id) {
  if (is_array($synset_id)) {
```

```
   $w_num = $synset_id['w_num'];
   $query_add = " AND w_num = '{$w_num}'";
   $synset_id = $synset_id['id'];
  }
  $result = mysql_query("SELECT * FROM synset WHERE synset_id = '{$synset_id}'{$query_add};");
  while ($line = mysql_fetch_array($result)) {
   if ($line['word'] != $original_word && strpos($line['word'], "_") === false) {
    if (strpos($line['word'], "(") !== false) {
     $bracket_pos = strpos($line['word'], "(");
     $line['word'] = substr($line['word'], 0, $bracket_pos);
    }
    $lemmas[] = $line['word'];
    $lemmas[] = $line['word']; // 2x :: count the actual words (compared to the words from gloss)
   }
   $result_wn = mysql_query("SELECT * FROM gloss WHERE synset_id = {$line['synset_id']};");
   while ($line_wn = mysql_fetch_array($result_wn)) {
    $words = $lemmatizer->lemmatizeText($line_wn['gloss']);
    foreach ($words as $word) {
     $lemmas[] = $word['lemma'];
    }
   }
  }
 }
 return $lemmas;
}

/**
 * creates new valid contents of the 'context' field
 * @param string $line
 * @param array $lemmas
 * @param int $mpl (optional)
 * @return string
 */
function updateContextData($l_root, $lemmas, $mpl = 1) {
 $context_items = array();
 $result_context = mysql_query("SELECT * FROM context WHERE root_id = {$l_root['id']};");
 while ($line_context = mysql_fetch_array($result_context, MYSQL_ASSOC)) {
  $context_items[] = $line_context;
 }

 foreach ($lemmas as $lemma) {
  if ($lemma != $l_root['root']) {
   $found = false;
   foreach ($context_items as $key => $context_item) {
    if ($context_item['root'] == $lemma) {
     $context_items[$key]['num_sought'] += $mpl;
     $context_items[$key]['updated'] = true;
     $found = true;
     break;
    }
   }
   if (!$found) {
    $context_items[] = array(
     "root_id" => $l_root['id'],
     "root" => $lemma,
     "num_sought" => $mpl,
     "inserted" => true
    );
   }
```

```
  }
 }

 foreach ($context_items as $key => $context_item) {
  $context_item['root'] = escape_string($context_item['root']);
  $fraction = 30 + ($mpl / 100);
  if ($fraction > 90) $fraction = 90;
  $context_item['num_found'] = round($context_item['num_sought'] * ($fraction / 100));
  $context_item['c_weight'] = calculateContextWeight($context_item['num_found'], $context_item['num_sought']);

  if ($context_item['inserted']) {
   // we added a new context item, let's update the database
   $result_insert = mysql_query("INSERT INTO context SET root_id = {$l_root['id']},
    root = '{$context_item['root']}', num_found = '{$context_item['num_found']}',
    num_sought = '{$context_item['num_sought']}', c_weight = '{$context_item['c_weight']}',
    time_added = NOW();");
  } else if ($context_item['updated']) {
   // we updated information about this context item, let's update the database
   $result_update = mysql_query("UPDATE context SET root = '{$context_item['root']}',
    num_found = '{$context_item['num_found']}', num_sought = '{$context_item['num_sought']}',
    c_weight = '{$context_item['c_weight']}' WHERE id = {$context_item['id']};");
  }
 }
}
?>
```

Dodatek B

Obdelava sopomenk za uporabo v besedilu

```
<?php
/**
 * applies the necessary changes to the given lemma
 * @param $text
 * @param $manipulation
 */
function manipulateWord($text, $manipulation) {
 /*
  * $manipulation can be one of the following:
  * PLURAL, PAST_SIMPLE, PAST_PARTICIPLE, GERUND,
  * PRESENT_1ST_PERSON, PRESENT_PLURAL, PRESENT_SINGULAR
  */

 $words = explode(" ", $text);
 $manipulate_key = 0;

 if ($manipulation != "PLURAL") {
  // if a verb contains a modal auxiliary verb, don't change anything
  foreach ($words as $word) {
   if (in_array($word, $this->modal_auxiliary)) {
    return $text;
   }
  }
 }

 // select the key of the word that we need to reformat
 $words_temp = $words;
 $ss_type = "v";
 if ($manipulation == "PLURAL") {
  // nouns tend to be at the end of the words in the suggestion
  $words_temp = array_reverse($words, true);
  $ss_type = "n";
 }

 foreach ($words_temp as $key => $word) {
  $word_esc = escape_string($word);
  $result = mysql_query("SELECT synset_id FROM synset WHERE
```

```
  word = '{$word_esc}' AND ss_type = '{$ss_type}';");
 $num = mysql_num_rows($result);
 if ($num > 0) {
  $manipulate_key = $key;
  break;
 }
}

$word = $words[$manipulate_key];

// take care of first person verbs
if (
 $manipulation == "PRESENT_1ST_PERSON" &&
 ($word == "be" || $word == "is" || $word == "are")
) {
 $word = "am";
}
else if ($manipulation == "PRESENT_SINGULAR" && $word == "be") {
 $word = "is";
}
else if ($manipulation == "PRESENT_SINGULAR" && $word == "have") {
 $word = "has";
}
else if (
 $manipulation == "PRESENT_PLURAL" &&
 ($word == "be" || $word == "is" || $word == "are")
) {
 $word = "are";
}

// add the -s ending
else if ($manipulation == "PLURAL" || $manipulation == "PRESENT_SINGULAR") {
 if ($word == "have" && $manipulation == "PRESENT_SINGULAR") $word = "has";
 else if ($word == "be" && $manipulation == "PRESENT_SINGULAR") $word = "is";
 else if (
  substr($word, -1, 1) == "y" &&
  in_array(substr($word, -2, 1), array("a", "e", "i", "o", "u"))
 ) {
  $word = substr($word, 0, strlen($word) - 1) . "s";
 } else if (substr($word, -1, 1) == "y") $word = substr($word, 0, strlen($word) - 1) . "ies";
 else if (in_array(substr($word, -1, 1), array("h", "o", "s", "x"))) $word = $word . "es";
 else $word = $word . "s";
}

// convert the verb to past tense (simple or participle)
else if ($manipulation == "PAST_SIMPLE" || $manipulation == "PAST_PARTICIPLE") {
 $array_of_endings = array(
  "b", "c", "d", "f", "g", "h", "i", "j", "k", "l", "m",
  "n", "o", "p", "r", "s", "t", "u", "v", "w", "x", "z"
 );
 $word_esc = escape_string($word);
 $result = mysql_query("SELECT * FROM irregular_verbs WHERE base = '{$word_esc}';");
 $num = mysql_num_rows($result);
 if ($num > 0) {
  $line = mysql_fetch_array($result);
  $word = ($manipulation == "PAST_SIMPLE") ? $line['past_simple'] : $line['past_participle'];
 } else {
  if (
   substr($word, -1, 1) == "y" &&
```

```
    in_array(substr($word, -2, 1), array("a", "e", "i", "o", "u"))
   ) {
    $word = $word . "ed";
   } else if (substr($word, -1, 1) == "y") {
    $word = substr($word, 0, strlen($word) - 1) . "ied";
   } else if (
    in_array(substr($word, -1, 1), array("b", "g", "m", "p", "t", "v")) &&
    in_array(substr($word, -2, 1), array("a", "e", "i", "o", "u"))
   ) {
    $word = $word . substr($word, -1, 1) . "ed";
   } else if (in_array(substr($word, -1, 1), $array_of_endings)) {
    $word = $word . "ed";
   } else {
    $word = $word . "d";
   }
  }
 }

 // add the -ing ending
 else if ($manipulation == "GERUND") {
  if (
   in_array(substr($word, -1, 1), array("b", "g", "m", "p", "t", "v")) &&
   in_array(substr($word, -2, 1), array("a", "e", "i", "o", "u"))
  ) {
   $word = $word . substr($word, -1, 1) . "ing";
  } else if (substr($word, -1, 1) == "e") {
   $word = substr($word, 0, strlen($word) - 1) . "ing";
  } else {
   $word = $word . "ing";
  }
 }

 $words[$manipulate_key] = $word;
 $text = implode(" ", $words);
 return $text;
}
?>
```

Dodatek C

Semiotična obdelava besedila

```
<?php
/**
 * splits the text into separate sentences
 */
function parseSentences() {
 if ($this->structure['phase'] != "initialized") return false;
 $sentences = array();
 $text = "\n" . $this->structure['text'] . "\n";
 $text_length = strlen($text);
 $sentence = "";
 $sentence_delimiters = array(".", "?", "!", "\n");
 $delimiters = array(" ", ".", "?", "!", "\n");
 for ($i = 0; $i < $text_length; $i++) {
  $sentence .= $text[$i];
  if (in_array($text[$i], $sentence_delimiters) && trim($sentence)) {
   if ($text[$i] == ".") {
    // are we dealing with an abbreviation, an acronym or a decimal number?
    $prev_part = "";
    $temp_i = $i - 1;
    while (!in_array($text[$temp_i], $delimiters)) {
     $prev_part = $text[$temp_i] . $prev_part;
     $temp_i--;
    }
    if ($prev_part &&
     (strlen($prev_part) <= 2 || is_numeric($prev_part) ||
     (!in_array($text[$i + 1], $delimiters) &&
     $text[$i + 1] == strtolower($text[$i + 1])))
    ) {
     continue;
    }
   }
   // are there other punctuation marks following this character?
   while (in_array($text[$i + 1], $sentence_delimiters)) {
    $sentence .= $text[$i + 1];
    $i++;
   }
   // are there any newlines present at the beginning of this sentence?
   for ($j = 0; $j < strlen($sentence); $j++) {
    if ($sentence[$j] == "\n") {
     $sentences[] = "\n";
```

```
    } else if (!in_array($sentence[$j], $delimiters)) {
     break;
    }
   }
   $sentences[] = array("text" => trim($sentence));
   // are there any newlines present at the end of this sentence?
   for ($j = strlen($sentence) - 1; $j >= 0; $j--) {
    if ($sentence[$j] == "\n") $sentences[] = "\n";
     else if (!in_array($sentence[$j], $delimiters)) break;
   }
   $sentence = "";
  }
 }
 for ($i = 0; $i < count($sentences); $i++) {
  if ($sentences[$i] == "\n") unset($sentences[$i]);
   else break;
 }
 $sentences = array_values($sentences);
 for ($i = count($sentences) - 1; $i >= 0; $i--) {
  if ($sentences[$i] == "\n") unset($sentences[$i]);
   else break;
 }
 $this->structure['sentences'] = array_values($sentences);
 $this->structure['phase'] = "parsed_sentences";
}

/**
 * splits the sentences into separate words and remember semiotic data
 */
function tokenize() {
 if ($this->structure['phase'] != "parsed_sentences") return false;
 foreach ($this->structure['sentences'] as $key => $sentence) {
  if ($sentence == "\n") continue;
  $tokens = array();
  $words = explode(" ", $sentence['text']);
  foreach ($words as $word) {
   // are there any punctuation marks present at the beginning of this word?
   $pre_characters = "";
   $actual_word = "";
   $special_state = true;
   for ($i = 0; $i < strlen($word); $i++) {
    $ord = ord($word[$i]);
    if (
     ($ord >= 48 && $ord <= 57) ||
     ($ord >= 65 && $ord <= 90) ||
     ($ord >= 97 && $ord <= 122)
    ) {
     if ($special_state) $special_state = false;
     $actual_word .= $word[$i];
    } else {
     if ($special_state) $pre_characters .= $word[$i];
      else $actual_word .= $word[$i];
    }
   }
   $post_characters = "";
   $final_word = "";
   $special_state = true;
   for ($i = strlen($actual_word) - 1; $i >= 0; $i--) {
    $ord = ord($actual_word[$i]);
```

```
    if (
     ($ord >= 48 && $ord <= 57) ||
     ($ord >= 65 && $ord <= 90) ||
     ($ord >= 97 && $ord <= 122)
    ) {
     if ($special_state) $special_state = false;
     $final_word = $actual_word[$i] . $final_word;
    } else {
     if ($special_state) $post_characters .= $actual_word[$i];
      else $final_word = $actual_word[$i] . $final_word;
    }
   }
   $upper_first = 0;
   if (strtoupper($final_word[0]) == $final_word[0]) $upper_first = 1;
   $upper_all = 0;
   if (strtoupper($final_word) == $final_word) $upper_all = 1;
   $final_word = strtolower($final_word);
   $tokens[] = array(
    "pre" => $pre_characters,
    "word" => $final_word,
    "post" => $post_characters,
    "upper_first" => $upper_first,
    "upper_all" => $upper_all
   );
  }
  $this->structure['sentences'][$key]['tokens'] = $tokens;
 }
 $this->structure['phase'] = "parsed_tokens";
}
?>
```

Slike

Literatura

[1] S. Pemberton et al., "XHTML™1.0 The Extensible HyperText Markup Language (Second Edition): A Reformulation of HTML 4 in XML 1.0," *W3C Recommendation (http://www.w3.org/TR/2002/REC-xhtml1-20020801)*, avg. 2002.

[2] B. Bos, H. W. Lie, C. Lilley, I. Jacobs, "Cascading Style Sheets, level 2 (CSS2) Specification," *W3C Recommendation (http://www.w3.org/TR/CSS2/)*, sept. 2009.

[3] C. W. Smullen III, S. A. Smullen, "Agnostic AJAX: Asynchronous JavaScript and Data," *Proceedings of Xtech 2008*, Dublin, Irska, maj 2008.

[4] D. McCarthy, R. Navigli, "The English Lexical Substitution Task," *Language Resources and Evaluation*, št. 43, zv. 2, str. 139-159, feb. 2009.

[5] R. Mihalcea, T. Pedersen, "Advances in Word Sense Disambiguation," *American Association for Artificial Intelligence (AAAI 2005)*, Pittsburgh, PA, ZDA, jul. 2005.

[6] D. Yarowsky, "Word-Sense Disambiguation by Using Statistical Models, Trained on Large Corpora," *Proceedings of the 14th conference on Computational linguistics*, Nantes, Francija, avg. 1992.

[7] E. Brill, "A Simple Rule-Based Part of Speech Tagger," *HLT '91: Proceedings of the workshop on Speech and Natural Language*, str. 112-116, 1992.

[8] Kirsten Malmkjaer, *The Linguistics Encyclopedia*, London: Routledge, str. 87, 2002.

[9] S. J. DeRose, "Grammatical Category Disambiguation by Statistical Optimization," *Computational Linguistics*, št. 16, zv. 1, str. 31-39, 1988.

[10] K. Church, P. Hanks, "Word Association Norms, Mutual Information and Lexicography," *Computational Linguistics*, št. 16, zv. 1, str. 22-29, 1991.

[11] M. P. Marcus, B. Santorini, M. A. Marcinkiewicz, "Building a Large Annotated Corpus of English: The Penn Treebank," *Computational Linguistics*, št. 19, zv. 2, str. 313-330, 1994.

[12] G. A. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM*, št. 38, str. 39-41, nov. 1995.

[13] S. Patwardhan, T. Pedersen, "Using WordNet-based Context Vectors to Estimate the Semantic Relatedness of Concepts," *EACL 2006 Workshop Making Sense of Sense - Bringing Computational Linguistics and Psycholinguistics Together*, str. 1-8, avg. 2007.

[14] B. B. Rieger, "Semiotics and Computational Linguistics. On Semiotic Cognitive Information Processing," *Computing with words in information/intelligent systems I. foundations*, str. 93-118, 1999.

[15] T. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer*, št. 17, zv. 6, str. 8-19, jun. 1984.