Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anton Zvonko Gazvoda

Integracija podatkovnih shem na osnovi analize podatkov z algoritmom arhetipske analize za povzemanje podatkovnih množic

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2014

Izjava o avtorstvu magistrskega dela

Spodaj podpisani Anton Zvonko Gazvoda, z vpisno številko **63090075**, sem avtor magistrskega dela z naslovom:

Integracija podatkovnih shem na osnovi analize podatkov z algoritmom arhetipske analize za povzemanje podatkovnih množic

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaža Branka Juriča,
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki "Dela FRI".

V Ljubljani, 16. septembra 2014 Podpis avtorja:

Zahvaljujem se mentorju prof. dr. Matjažu Branku Juriču za vodenje in usmerjanje pri izdelavi magistrskega dela. Zahvaljujem se tudi dr. Sebastijanu Špragerju za vse strokovne nasvete.

Kazalo

Slike

Tabele

Seznam uporabljenih kratic

kratica	angleško	slovensko
AA	archetypal analysis	arhetipska analiza
PCA	principal component analysis	analiza glavnih komponent
NMF	non-negative matrix factorization	nenegativna matrična faktorizacija
ICA	independent component analysis	analiza neodvisnih komponent
EM	expectation-maximization algorithm	maksimizacija pričakovane vrednosti
MKL	math kernel library	jedrna matematična knjižnica
MDS	multidimensional scaling	večdimenzionalno skaliranje
GP	genetic programming	genetsko programiranje
SaaS	Software as a Service	programska oprema kot storitev
API	application programming interface	aplikacijski programski vmesnik

Povzetek

Ključna aktivnost v procesu integracije aplikacij na podatkovnem nivoju je iskanje preslikav med podatkovnimi shemami, kar je osnova za izvedbo ustreznih transformacij podatkov. V ta namen predlagamo novo metodo za integriranje shem, ki deluje na osnovi ocenjevanja podobnosti med podatkovnimi instancami. Metoda temelji na arhetipski analizi, s katero generiramo povzetke podatkov elementov sheme. Njihove približke opišemo s konveksnimi ovojnicami. Za izračun povzetkov definiramo različne pristope za transformacijo podatkov v vektorski prostor in metrike podobnosti. Preslikave iščemo s pomočjo dveh algoritmov za odkrivanje enostavnih in kompleksnih preslikav. Metodo smo ovrednotili na testnih podatkih, ki vključujejo pravilne preslikave med shemami, in jo primerjali z iskalnikom preslikav COMA CE. Uspešnost smo ocenili z občutljivostjo (91%), specifičnostjo (75%), točnostjo (87%) in natančnostjo (91%), pri čemer je naša metoda v povprečju za 20% boljša od COMA CE.

Ključne besede: integracija podatkovnih shem na osnovi instanc, iskanje preslikav, arhetipska analiza, konveksna ovojnica, povzetek podatkov.

Abstract

Schema mapping discovery is key activity while performing data-level integration process and represents the basis for proper data transformation. For this purpose, we introduce novel instance-based schema matching method by using archetypal analysis in order to generate data summary for each schema element. Summary approximations are represented by convex hulls. We define several approaches for data transformation to vector space, as well as summary-similarity metrics. Two algorithms were developed in order to determine simple and complex matches. Our method was evaluated on the test data including proper mappings between schemas and compared with COMA CE schema matcher. Efficiency of our method was evaluated with sensitivity (91%), specificity (75%), accuracy (87%) and precision (91%). Compared with COMA CE, our method performs on average 20% better.

Keywords: instance-based schema matching, schema mapping, archetypal analysis, convex hull, data summary.

Poglavje 1

Uvod

Potreba po integraciji aplikacij se pojavi, ko v nekem podjetju ali organizaciji za podporo poslovanja uporabljajo množico aplikacij, npr. sistem za upravljanje s strankami, sistem za upravljanje virov podjetja, sistem za upravljanje oskrbovalne verige itd. Skoraj vsaka aplikacija delež podatkov deli z neko drugo aplikacijo. To pomeni, da je v primeru, če aplikacije ne uporabljajo skupne podatkovne baze, potrebna integracija. Ravno tako se trend prehajanja na oblačne aplikacije (SaaS - *Software as a Service*) iz leta v leto povečuje, s tem se vzporedno povečuje tudi potreba po integraciji med SaaS in klasičnimi aplikacijami in SaaS med seboj.

Narava poslovnih okolij je, da za svoje poslovanje uporabljajo množico sistemov in aplikacij. Razlog za to je doseganje modularnosti, ki omogoča fleksibilno in učinkovito nadgradnjo posameznih modulov z najsodobnejšimi rešitvami. Vsak modul poslovnega okolja je podprt z aplikacijo ali sistemom. Posledica modularne arhitekture je, da zahteva medsebojno integracijo, ne glede na to, ali module predstavljajo oblačne ali klasične aplikacije, ki se izvajajo na nekem strežniku v podjetju. Integracijo aplikacij lahko izvajamo na več nivojih. Tedaj govorimo o integraciji uporabniških vmesnikov, poslovni, procesni in podatkovni integraciji. V tej magistrski nalogi se bomo osredotočili na integracijo podatkovnega nivoja aplikacij.

Pri integraciji aplikacij na podatkovnem nivoju je glavni cilj vsem aplika-

cijam zagotoviti celovite in ažurne podatke. To pomeni, da se morajo spremembe na podatkih pojaviti v vseh integriranih aplikacijah. Kljub temu da si aplikacije lahko delijo vsebinsko iste podatke, jih redkokdaj strukturirajo na enak način. Ko govorimo o strukturi podatkov, imamo v mislih podatkovne sheme. Te opisujejo strukturo podatkov, elemente, njihove podatkovne tipe in zaloge vrednosti. Različne podatkovne sheme za predstavitev istih podatkov v različnih aplikacijah predstavljajo glavni izziv, ki ga je pri integriranju aplikacij potrebno rešiti. Prenos sprememb med podatki v aplikacijah zahteva ustrezno transformacijo podatkov v ustrezno obliko ciljne aplikacije. Transformacije med podatkovnimi shemami lahko definiramo ročno, vendar kmalu pridemo do spoznanja, da je ta način precej težaven. Če želimo napisati transformacijo med dvema shemama, moramo za vsak element točno vedeti, kakšne podatke hrani. Težave se pojavijo, ko so sheme zelo obsežne in vsebujejo zelo veliko elementov. Ne pozabimo tudi na to, da pogostokrat dokumentacije, ki bi razjasnila pomen posameznega elementa, ni na razpolago. Opisane težave poskušamo rešiti z uvajanjem samodejnih pristopov za integriranje podatkovnih shem. To področje je aktualno od pojavitve relacijskih baz do danes, saj še vedno ne obstaja rešitev, ki bi v vseh primerih učinkovito znala odkriti preslikave med elementi.

Iskanje podobnosti med podatkovnimi shemami in njenimi elementi imenujemo ujemanje shem (*schema matching*). Pojem moramo razlikovati od pojma preslikovanje shem (*schema maping*), saj ta opisuje transformacije iz ene sheme v drugo. Ena izmed temeljnih nalog podatkovne integracije je avtomatizacija opisanih pristopov, kar je hkrati tudi primarni cilj naše magistrske naloge. Posvetili se bomo zasnovi, evalvaciji in vpeljavi nove metode na področje samodejnega integriranja podatkovnih shem, ki temelji na principu iskanja podobnosti med podatki posameznih elementov. Z metodo, ki jo vpeljujemo, skušamo doseči visoko natančnost in zanesljivost. Pri tem predpostavljamo, da sistemi, ki jih integriramo, že hranijo podatke. Metoda, ki jo vpeljujemo, podobnosti med elementi išče na podlagi povzetkov podatkov elementov. Povzetke generiramo z metodo arhetipske analize.

Arhetipska analiza je pristop, s katerim med množico točk poiščemo tisto podmnožico, ki čim bolj natančno opisuje celotno podatkovno množico. Z matematičnega vidika gre za iskanje približka konveksne ovojnice. Navdih za uporabo te metode na področju samodejnega iskanja preslikav smo dobili v [1]. Avtorji članka opisujejo uporabo metode arhetipske analize za povzemanje množice tekstovnih dokumentov. Metoda se je v primerjavi z obstoječimi metodami za povzemanje dokumentov izkazala za eno izmed najučinkovitejših. Lastnost dobrih povzetkov, ki jih pridobimo z metodo arhetipske analize, želimo prilagoditi problemu samodejnega integriranja podatkovnih shem. Pristop predlagane metode temelji na analizi podatkov elementov, ki predstavlja velik potencial za doseganje zelo visoke stopnje samodejnosti.

V poglavju 2 bomo spoznali in primerjali obstoječe metode in pristope, ki se uporabljajo za samodejno integriranje podatkovnih shem. V poglavju 3 predstavimo analitično zasnovo predlagane metode arhetipske analize. Sledi poglavje 4, kjer predstavimo in opišemo posamezne komponente ogrodja za iskanje preslikav. Poglavje 5 prikazuje načine za predstavitev podatkov elementov sheme, da lahko iz njih izračunamo povzetke in podobnost med njimi. Poglavje vključuje tudi evalvacijo in primerjavo z znanimi uveljavljenimi rešitvami na področju integracije podatkovnih shem. Na koncu, v poglavju 6, predstavimo naše ugotovitve in sklepe.

Poglavje 2

Pregled obstoječih metod za samodejno integriranje podatkovnih shem

Proces integriranja aplikacij na podatkovnem nivoju zajema iskanje korespondenc med podatkovnimi shemami. Podatkovna shema je formalen zapis za predstavitev podatkov, ki jih hrani neka aplikacija. Shema določa, katere podatke neka aplikacija hrani za neko entiteto (npr. *Oseba*). Podatki so v shemi določeni z elementi. Za vsak element v shemi je definirano, kakšen tip podatkov hrani, kakšno ima zalogo vrednosti (katere podatke zavzema) in omejitve (npr. obvezen podatek, primarni ključ, tuji ključ itd.). Shema določa tudi hierarhijo in relacije med entitetami in elementi. Tipični primeri podatkovnih shem so sheme XML, sheme SQL ali entitetno-relacijski diagrami.

Pojem *ujemanje shem* označuje, kako se elementi ene sheme preslikajo v elemente druge sheme. Iskanje ujemanj izvedemo tako, da iščemo elemente, ki predstavljajo semantično iste podatke. Pri tem za posamezni element ni nujno, da sploh ima preslikavo v nekem elementu druge sheme. Preslikavo med enim ali več elementi ene sheme in enim ali več elementi druge sheme imenujemo korespondenca. Korespondenca nam pove, kateri elementi ene

sheme hranijo semantično iste podatke kot elementi druge sheme. Kot primer vzemimo dve shemi, označeni z S1 in S2, kjer ima shema S1 elementa *name* in *surname*, shema S2 pa element *fullname*. V tem primeru med temi elementi obstaja semantična korespondenca, saj elementi hranijo podatke o imenu in priimku. Preslikavo med elementi lahko v tem primeru opišemo: {S1.*name*, S1.*surname*} $\leftrightarrow$ {S2.$fullname$}.

Preslikave imajo največkrat številnost (kardinalnost) ena-proti-ena (1:1), kjer se natanko en element sheme slika v natanko en element druge sheme. Poleg te številnosti je možna tudi ena-proti-mnogo (1:*N*,*N*:1), kjer se en element sheme slika v več elementov druge sheme ali obratno, kot prikazuje prej omenjeni primer. Številnost preslikav mnogo-proti-mnogo (*M*:*N*) predstavlja preslikavo med dvema shemama, na nivoju elementa pa ne obstaja. Povezave številnosti 1:1 bomo v tem delu označili kot enostavne preslikave, preslikave številnosti (1:*N*, *N*:1 in *M*:*N*) pa kompleksne.

Poleg same številnosti lahko upoštevamo še lastnost globalnosti. O globalni kardinalnosti preslikave govorimo takrat, ko preslikava velja na globalni ravni sheme. Kot primer vzemimo preslikavo številnosti 1:1 med elementoma S1.*ime* in S2.*ime*. Če ne obstaja nobene druga preslikava iz S2, ki bi se slikala v S1.*ime*, in nobena druga preslikava iz S1, ki bi se slikala v S2.*ime*, je kardinalnost globalna. V nasprotnem primeru je kardinalnost lokalna. Večina obstoječih pristopov za reševanja problema iskanja preslikav med podatkovnimi shemami podpira preslikave številnosti 1:1 na lokalni in globalni ravni, v določenih primerih pa tudi številnosti 1:*N*. Preslikav številnosti *N*:1 in *N*:*M* večina rešitev ne obravnava z obzirom na dejstvo, da se take preslikave v praksi zelo redko pojavijo.

Enostavne preslikave so praviloma neposredne. Gre za preslikave, kjer je element, ki hrani enake podatke, drugače poimenovan ali se nahaja na drugem nivoju glede na izvorno shemo. Na drugi strani imajo kompleksne preslikave tipično številnost 1:*N* in predstavljajo preslikavo, kjer se en element preslika v strukturo elementov ciljne sheme ali obratno. Tudi preslikave številnosti 1:1 so lahko kompleksne. To so tiste preslikave, kjer moramo presli-

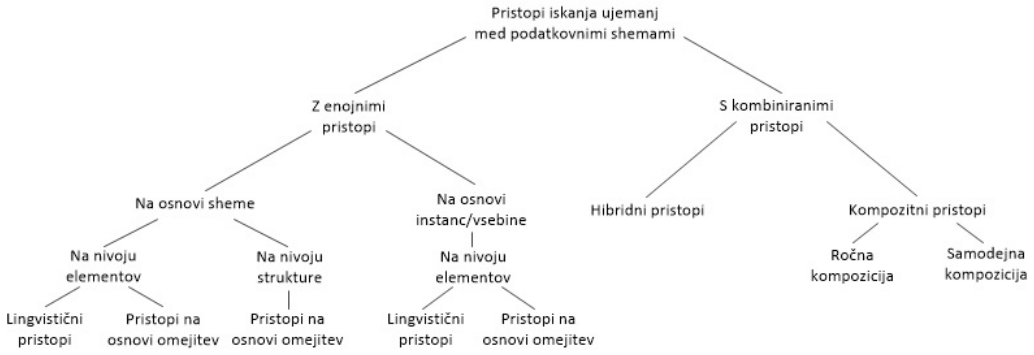
kati le podmnožico podatkov elementa, zamenjati zaporedje v zapisu podatka (npr. drugačen format datuma: iz *dd/mm/llll* v *mm:dd:llll*) ali pa pretvoriti vrednosti (kjer npr. S1 vsebuje element *status* z vrednostmi [*naročilo sprejeto, naročilo obdelano, naročilo poslano*]; S2 pa element *stat* z vrednostmi [*1, 2, 3*], ki so definirane v šifrantu, in kjer 1 prestavlja sprejeto, 2 obdelano, 3 odposlano). Z našo metodo želimo podpreti preslikave vseh številnosti.

Iskanje ujemanj med podatkovnimi shemami lahko apliciramo na mnoga področja, v prvi vrsti pri upravljanju podatkovnih baz, kjer je potrebno zagotoviti sinhronizacijo in integracijo med njimi. Drugi primer uporabe je na področju aplikacij, ki probleme rešujejo na osnovi baze znanja in temeljijo na iskanju ujemanj med ontologijami. Kot primer navedimo področje medicine, kjer je ujemanje med shemami pomembno za iskanje povezav med pacientovo kartoteko in preostalimi medicinskimi poročili z namenom združevanja podatkov iz več virov. Še več, če želimo podatke v aplikaciji predstaviti na nekoliko drugačen način, kot so shranjeni v podatkovni bazi, potrebujemo ponovno preslikavo med dvema shemama. Preslikovanje potrebujemo tudi pri elektronskemu poslovanju, kjer sistemi med sabo komunicirajo s sporočili in vsak sistem zahteva svojo obliko.

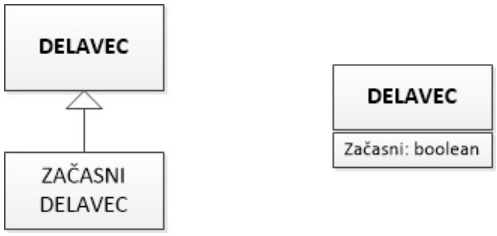
2.1 Delitev metod za iskanje preslikav

Metode za iskanje preslikav v osnovi delimo na dve skupini: na enojne pristope in kombinirane pristope, kot je predstavljeno na sliki 2.1. Enojni pristopi za iskanje uporabljajo le eno metodo, ki deluje na osnovi podatkov sheme ali pa na osnovi podatkovnih instanc. Po drugi strani kombinirani pristopi uporabljajo več metod hkrati, kot je navedeno v [5, 6].

Enojni pristopi lahko za iskanje preslikav uporabljajo podatke sheme, in sicer: imena elementov, podatkovne tipe, strukture shem in relacije med shemami in podshemami. V prvi vrsti te pristope delimo na tiste, ki delujejo na nivoju enega elementa, in tiste, ki delujejo na nivoju strukture elementov. Pri ujemanju na nivoju elementa govorimo o preslikavah številnosti 1:1, tj. o



Slika 2.1: Delitev metod za iskanje ujemanj med podatkovnimi shemami.



Slika 2.2: Primer kompleksne strukturne preslikave.

nivoju atoma. Ujemanja na nivoju strukture opisujejo preslikave med dvema kombinacijama elementov (vsaka iz svoje sheme), ki sta predstavljeni s poljubno strukturo. V idealnem primeru je ujemanje med dvema strukturama popolno, če se vse komponente (elementi) strukture iz ene in druge sheme popolnoma ujemajo. Drugi, nasprotni primer, predstavlja nepopolno ujemanje. Oba primera sta prikazana na primeru v tabeli 2.1. Iskanje strukturnih ujemanj lahko razširimo z uporabo poznanih enakovrednih vzorcev, ki jih lahko hranimo v zunanji shrambi. Primer preslikave kompleksnega strukturnega vzorca predstavlja shema za začasnega delavca, prikazana na sliki 2.2. *Začasni delavec* je lahko shema, ki razširja generično shemo *Delavec* ali je to shema *Delavec* z elementom *Začasni* (tip *boolean*).

Druga vrsta enojnih iskalnikov deluje na osnovi podatkovnih instanc. Te metode delujejo tako, da iz samih podatkov poskušajo ugotoviti, ali dva

elementi sheme S1	elementi sheme S2	
Ime	S2_ime	Popolno ujemanje med S1 in S2.
Priimek	S2_priimek	
DatumRojstva	S2_dr	
Spol	S2_s	
Ulica_in_hisna_st	S2_Ulica	Nepopolno ujemanje med S1 in S2.
Mesto	S2_Histna_st	
Drzava	S2_Mesto	
GEO_lokacija		

Tabela 2.1: Primerjava med popolno in nepopolno preslikavo med dvema strukturama.

elementa hranita iste podatke.

Podatkovni tipi elementov lahko ravno tako pomagajo pri iskanju enojnih ujemanj, pri čemer to informacijo uporabimo za zožitev nabora elementov druge sheme, v katere se lahko element preslika. Za element podatkovnega tipa decimalno število (*decimal*) lahko tako trdimo, da se zagotovo ne preslika v podatkovni tip datum (*date*). Težje trdimo, da se podatkovni tip niz (*string*) ne more slikati v podatkovni tip celo število (*integer*), ali obratno, saj nas nič ne omejuje, da število ali datum shranimo kot niz. Obstajajo torej podatkovni tipi, ki lahko hranijo različne tipe podatkov.

Ujemanja na nivoju elementov lahko identificiramo z algoritmi, ki so podobni algoritmom za procesiranje operacije združevanja (*join*) pri relacijskih podatkovnih bazah. Glede na tip iskalnika lahko primerjave izvajamo glede na ime, opis ali podatkovni tip elementa sheme. Vsak element sheme S1 se primerja z vsemi elementi sheme S2; tako algoritem pridobi seznam vseh podobnih elementov. Za vse elemente se na podlagi neke metrike izračuna podobnost, pri čemer so kandidati za preslikavo tisti elementi, ki dosežejo podobnost nad vnaprej določenim pragom (*threshold*). Učinkovitejši algoritmi pri integriranju shem za učinkovitejše iskanje enakih vrednosti uporabljajo

razpršilne funkcije (*hash functions*), ali ujemanja iščejo na podlagi več lastnosti elementa hkrati (npr. ime in podatkovni tip) [5].

Kombinirani pristopi se v praksi izkazujejo za najboljše, saj z njimi najdemo več ujemanj, kot če uporabimo le eno metodo, kot je navedeno v [5]. Kombinirane pristope delimo na hibridne in kompozitne, katere bomo natančneje opisali nekoliko kasneje.

Slabost metod, ki delujejo na podlagi podatkov sheme, je ta, da so omejene s podatki, ki so jim na razpolago. Količino podatkov sicer lahko razširimo z dodatnimi podatki, npr. z uporabo zunanje podatkovne baze vseh možnih imen za element, ki npr. hrani imena neke osebe (*name, n, firstname, fn, given name* itd.). V določenih primerih samih imen elementov ne moremo uporabiti, saj ni rečeno, da element dejansko hrani podatke, ki jih opisuje ime elementa, ali metoda odpove že v primeru, ko shemi za poimenovanje elementov uporabljata različen jezik (npr. angleški in slovenski). Primer, s katerim se lahko srečamo v praksi, je sistem SAP (*Systems Applications Products*), kjer je iz imen elementov nemogoče ugotoviti vsebino elementa. Po drugi strani metode, ki za ujemanje uporabljajo podatkovne instance, ne trpijo za to slabostjo, saj v primeru podobnih podatkov skoraj zagotovo najdejo ujemanje. Naprednejše metode temeljijo na pristopih iz področja razpoznavanja vzorcev (*pattern recognition*) in so s tem manj odvisne od podobnosti samih podatkov.

2.2 Primeri metod za integracijo podatkovnih shem

Naslednje metode so tipični predstavniki posameznih skupin, ki smo jih opredelili v predhodnem poglavju:

1. **Metode na osnovi podatkov sheme**: lingvistične metode na osnovi imen, opisov in imenskih prostorov; metode na osnovi omejitev kot podatkovni tipi, ključi in zaloge vrednosti; metode za iskanje ujemanj

grafov.

2. **Metode na osnovi podatkov instanc**: lingvistične metode, ki delujejo na osnovi tehnik pridobivanja informacij (*information retreival techniques*), metode na osnovi podatkovnega rudarjenja in strojnega učenja.

3. **Kombinirane metode**: hibridne in kompozitne metode, ki so sestavljene iz več metod, na osnovi podatkov shem in instanc.

Pri posamezni skupini smo našteli standardne pristope, a obstajajo še številni drugi [7]. V nadaljevanju magistrske naloge podrobneje predstavimo dve metodi iz skupine metod na osnovi podatkovnih instanc, in sicer metodo na osnovi regularnih izrazov in evolucijsko metodo na osnovi razdvojevanja podatkov. V skupino metod na osnovi podatkovnih instanc spada tudi metoda, ki jo predlagamo v okviru te magistrske naloge. To je tudi razlog, da podrobneje preučimo obstoječe metode in primerjamo rešitve. Metode ostalih skupin bomo na kratko predstavili zaradi bolj nazornega prikaza razlik med pristopi.

2.2.1 Lingvistične metode

Lingvistične metode ali metode na osnovi jezikovnih lastnosti iščejo semantična ujemanja na podlagi podobnosti med besedami in besednimi zvezami. Primera metod, ki delujeta na osnovi lingvistike, sta metoda za iskanje podobnosti na osnovi imen elementov in metoda za iskanje podobnosti na osnovi njihovih opisov.

Iskanje ujemanj na podlagi podobnosti imen elementov sheme

Ta metoda išče ujemanje z iskanjem elementov druge sheme, ki imajo isto ali podobno ime. Podobnost med imeni lahko merimo z različnim pristopi:

- **Podobnost imen**: Če imata dva elementa zelo podobno ime, obstaja velika verjetnost, da hranita iste podatke. Tudi elementi iz istih imen-

skih prostorov (*namespace*) z veliko verjetnostjo predstavljajo isto semantično vsebino, npr. *datum kreiranja* in *datum* kot primera elementa iz imenskega prostora *.../faktura*.

- **Podobnost sinonimov**: posredno iskanje podobnosti elementov na podlagi sinonimov, npr. proizvajalec in znamka.
- **Podobnost korena imena**: procesiranje besed z algoritmi za luščenje korenov besed; primerno v primerih simboličnih predpon in pripon, npr. *StZaposlenega - Številka Zaposlenega* ali pa *DRojstva - Datum Rojstva*.
- **Podobnost nadpomenk**: iskanje nadpomenk imen elementov. Preslikave med elementi odkrivamo na podlagi enakosti nadpomenk, npr. gsm telefon - kontakt, mobilni telefon - kontakt → gsm telefon - mobilni telefon.
- **Podobnost podnizov**: podobnost imen na podlagi skupnih podnizov ali razdalje urejanja (*edit distance*), npr. domači naslov - naslov za vročanje.

Uporaba metod za iskanje podobnosti na osnovi sinonimov in nadpomenk zahteva uporabo dodatnih slovarjev. V splošnem lahko uporabimo slovarje naravnih jezikov, npr. slovensko-angleški slovar, če ena shema uporablja slovenski in druga angleški jezik. Poleg naravnih slovarjev lahko iskalne metode uporabljajo tudi domensko specifične slovarje, ki vsebujejo sinonime, skupna imena, opise elementov, okrajšave itd. Pri tem je potrebno vložiti nekoliko več truda, saj moramo pri gradnji slovarjev zagotoviti konsistentnost.

Lingvistične metode so zelo občutljive na enakozvočnice, ki lahko zavedejo algoritem za iskanje preslikav. Enakozvočnice so besede, ki ob izgovoru zvenijo enako ali se tudi enako črkujejo kot neka druga beseda, a imajo drugačen pomen. Lahko so del naravnega jezika ali so specifične za neko domeno. Primer enakozvočnice naravnega jezika predstavlja beseda *list*, ki predstavlja del drevesa ali pa *list* kot kos papirja. Primer domensko specifične enakozvočnice je besedna zveza *vrsta dela*, ki v domeni književnosti

predstavlja, za kakšen žanr knjižnega dela gre, v domeni zaposlovanja *vrsta dela* predstavlja, kakšno delo oz. opravilo je potrebno opravljati. Algoritmi za iskanje ujemanj na osnovi imen lahko do neke mere izboljšajo odpornost na enakozvočnice z dodatnimi slovarji, ki jih definirajo uporabniki. V najenostavnejšem primeru lahko algoritem opozori uporabnika, da je našel element z dvoumnim imenom. V takih primerih lahko uporabimo kontekst, v katerem se pojavi element. Kontekst predstavlja shema, v kateri se nahaja element, npr. *Knjiga.VrstaDela* in *ZaposlitveniOglas.VrstaDela*. V obeh primerih se *vrsta dela* pojavi v drugačnih kontekstih. V tem primeru kombiniramo lingvistični pristop s strukturnim.

Algoritme za integriranje podatkovnih shem lahko razširimo s slovarji sopomenk in nadpomenk. Procesiranje v praksi tako vključuje še slovar D, s čimer lahko problem predstavimo na naslednji način:

S1 (ime, ...)
S2 (ime, ...)
D (ime1, ime2, podobnost),

kjer ima podobnost zalogo vrednosti $\mathbb{R} = [0, 1]$. V tem primeru se algoritem za iskanje ujemanj izvaja na način, kot se procesirajo dvosmerne operacije združevanja (glej kodo 2.1).

Koda 2.1: Operacija združevanja

```
Select S1.ime, S2.ime, D.podobnost
From S1, S2, D
Where (S1.ime = D.ime1) and (D.ime2 = S2.ime) and
    (D.podobnost > mejna vrednost)
```

V tem primeru predpostavljamo, da slovar D vsebuje vse potrebne pare. Zapisi parov so oblike <*ime_elementa* >- <*ime_elementa* >- σ, kjer σ predstavlja oceno podobnosti. Recimo, da D vsebuje zapisa *A-B-0,9* in *B-C-0,8*, potem pričakujemo, da D vsebuje tudi *B-A-0,9* in *C-B-0,8*, hkrati pričakujemo, da D vsebuje tudi *A-C-*σ in *C-A-*σ. Najenostavnejša možnost

za izračun podobnosti na podlagi tranzitivnih povezav je produkt, pri čemer je v tem primeru podobnost med A in C enaka $0{,}9 \cdot 0{,}8 = 0{,}72$. Način izračuna podobnosti je odvisen od tipa preslikave, uporabe enakozvočnic in drugih faktorjev, zato je običajno bolj kompleksen kot produkt dveh števil.

Iskanje ujemanj na podlagi opisov elementov shem

Podatkovne sheme pogosto vsebujejo komentarje, ki vsebujejo opise posameznih elementov v naravnem jeziku. Z uporabo lingvističnih metod lahko opise uporabimo za detekcijo preslikav med elementi shem. Kot primer vzemimo:

S1.dlim *//ime delavca*
S2.imed *//ime delavca*,

kjer sta opisa elementov enaka, kar pomeni, da lahko zelo enostavno najdemo ujemanje. V realnosti težko pričakujemo tako čiste opise elementov, zato potrebujemo razširjene lingvistične pristope. Ena izmed enostavnejših je metoda za iskanje ključnih besed iz opisov. Iskanje ujemanj potem izvajamo tako, kot bi izvajali ujemanja na podlagi imen elementov, le da tu namesto imen uporabljamo ključne besede. Uporabimo lahko tudi bolj napredne metode, kot so npr. tehnike za razpoznavanje pomena naravnega jezika, kjer iščemo izraze z istim semantičnim pomenom.

2.2.2 Iskanje ujemanj na podlagi omejitev

Sheme v večini primerov vsebujejo omejitve na nivoju elementov, kamor spadajo podatkovni tipi, zaloga vrednosti, enoličnost, izbirnost, kardinalnost relacij itd. Če obe shemi, med katerima želimo poiskati ujemanja, vsebujeta te informacije, jih lahko uporabimo pri procesu integriranja shem. Podobnost med njima lahko ocenimo na podlagi enakosti podatkovnih tipov, zaloge vrednosti elementov, karakteristik ključa (enoličnost, primarni, tuji ključ) in številnosti relacij. Tudi v tem primeru lahko uporabimo algoritem, ki deluje

elementi sheme S1	elementi sheme S2
Zaposleni	Osebje
StZap [Integer, Primary key]	OSt [Integer, unique]
ImeZap [varchar(50)]	OIme [string]
StOddelka [Integer, references Oddelek]	Oddelek [string]
Plača [Decimal]	Datum_rojstva [Date]
Rojstni_datum [Date]	
Oddelek	
StOddelka [Integer, Primary key]	
ImeOddelka [varchar(40)]	

Tabela 2.2: Primer iskanja preslikav na podlagi omejitev.

na osnovi procesiranja operacij združevanja, le da sedaj enakosti iščemo na podlagi podatkovnih tipov, strukture in njihovih omejitev. Za reševanje problema različnih poimenovanj podatkovnih tipov (*string*, *varchar*) in omejitev lahko uporabimo dodatne slovarje sopomenk.

V tabeli 2.2 je prikazan primer, kjer vidimo, da se na podlagi podatkovnega tipa element S1.$Rojstni\_dan$ ujema z elementom S2.$Datum\_rojstva$, druge preslikave pa ni, saj ne obstaja noben element, ki bi ravno tako imel podatkovni tip datum (*date*). Če poleg podatkovnega tipa upoštevamo še podatke o ključu, vidimo, da se element S2.OSt preslika v element S1.$StZap$ ali v element S1.$StOddelka$. V shemi S2 ostaneta le še dva elementa, in sicer S2.$OIme$ in S2.$Oddelek$, oba tipa *string*, kar pomeni, da se lahko slikata ali v element S1.$ImeZap$ ali S1.$ImeOddelka$. Natančnejše preslikave na podlagi teh podatkov ni moč dobiti. Slabost metod, ki delujejo samo na podlagi informacij omejitev, je ta, da vračajo nepopolne rezultate pri številnosti *N*:*M*. Kljub temu so take metode še vedno uporabne za omejitev števila elementov, v katere se nek drug element lahko preslika. Če bi poleg ujemanj na podlagi omejitev v drugi fazi uporabili še iskalnik na podlagi ujemanj imen, bi zelo verjetno prišli do rešitve, ki bi za rezultat vrnila preslikavo iz

S1.$ImeOddelka$ v S2.$Oddelek$. Iskalnik namreč v tem primeru izbira le med elementoma S2.$OIme$ in S2.$Oddelek$. To pomeni, da tako posredno najdemo tudi preslikavo med S1.$ImeZap$ in S2.$OIme$ kot edino preostalo možnost. To se neposredno navezuje na hibridne pristope, o katerih bomo govorili nekoliko kasneje.

Poleg pravkar navedenih eksplicitnih omejitev poznamo tudi implicitne, ki izhajajo iz same strukture sheme. Govorimo o referencah znotraj shem (npr. tuji ključ) in podatkih o sosednosti elementov. Te informacije nam povedo, kateri elementi spadajo pod isti element na višjem nivoju, do česar pridemo preko tranzitivnih relacij. Omejitve lahko interpretiramo kot strukture, ki nam omogočajo iskanje ujemanja na podlagi njihove podobnosti. Večina podatkovnih shem ima hierarhično strukturo, če jih gledamo z vidika gnezdenja podshem. Algoritmi, ki iščejo ujemanja med hierarhičnimi strukturami, lahko delujejo na dva načina: od spodaj navzgor (*bottom-up*) in obratno, od zgoraj navzdol (*top-down*). Pristopi od zgoraj navzdol so običajno bolj optimalni, saj algoritmi na višjem nivoju omejijo število primerjav, ki jih morajo izvesti na najnižjem nivoju - na nivoju elementa. Algoritem tako zoži nabor možnih kombinacij. Slabost tega pristopa se izkaže v primeru, če je struktura shem na višjih nivojih precej drugačna. V tem primeru algoritem lahko spregleda rešitev, čeprav je ujemanje na najnižjem nivoju lahko visoko. Pristop od spodaj navzgor deluje tako, da primerja vse kombinacije elementov na najnižjem nivoju, pri čemer v nasprotju s prej omenjeno metodo algoritem najde ujemanje, četudi se strukturi shem na višjih nivojih precej razlikujta.

2.2.3 Iskanje ujemanj na osnovi podatkovnih instanc

Podatkovne instance shem predstavljajo pomemben vir podatkov, na podlagi katerih lahko iščemo ujemanje. Vsebino elementov shem lahko uporabimo za identifikacijo elementov, ki hranijo semantično podobne podatke. Obstajajo primeri, ko podatki sheme niso na voljo ali so omejeni, predvsem v primerih, ko se ukvarjamo z nestrukturiranimi podatki. Metode na osnovi podatkovnih instanc lahko uporabimo za odkrivanje napačnih preslikav, ki so bile prido-

bljene na osnovi podatkov sheme. V primeru, ko obstaja več možnosti za preslikavo nekega elementa, se lahko odločimo na podlagi podobnosti podatkovnih instanc. Za iskanje ujemanj na nivoju podatkovnih instanc lahko uporabimo metode, ki smo jih omenili že na nivoju shem:

- Pri elementih, ki vsebujejo **tekstovne podatke**, lahko uporabimo lingvistične metode. Najboljše pristope predstavljajo tehnike za iskanje ključnih besed in tematske analize na podlagi frekvenc pojavitev besed in besednih zvez.

- Za **strukturirane podatke**, kot so števila in nizi, se uporabljajo tehnike, ki delujejo na osnovi karakteristik omejitev. Primeri karakteristik za števila so povprečje, srednja vrednost, zaloga vrednosti (interval), ali znakovni vzorci pri nizih. Vzorci so zelo uporabni pri podatkih, kot so telefonske številke, poštne številke, naslovi, IBAN, EMŠO, davčna številka itd.

Prednost analize podatkovnih instanc je natančna karakterizacija vsebine nekega elementa. To na nivoju sheme ni možno, saj se zanašamo na opisne podatke. Karakterizacijo vsebine lahko uporabimo na dva načina. En način je, da jo uporabimo za razširitev metod, ki delujejo na osnovi podatkov sheme. Iskalnik preslikav, ki deluje na osnovi omejitev, lahko razširimo z uporabo analize podatkov. To omogoči natančnejšo identifikacijo preslikav med elementi istega podatkovnega tipa na podlagi zaloge vrednosti podatkov in znakovnih vzorcev. Pri drugem načinu gre pri integraciji shem le za uporabo karakterizacije vsebine. V tem primeru identifikacija preslikav poteka tako, da se v prvem koraku izvede karakterizacija vsebine elementov na podlagi instanc sheme S1. V drugem koraku se nato izvede iskanje preslikav med množico instanc sheme S2 tako, da se posamezna instanca primerja s shemo S1. Rezultati preslikav posameznih instanc se združijo in izvede se abstrakcija na nivoju sheme, na podlagi česar se generirajo kandidati preslikav med elementi S1 in elementi S2. Na osnovi tega pristopa so predlagane različne tehnike za klasificiranje kot so odločitvena pravila, strojno učenje in

nevronske mreže [6].

Pristopi za integriranje shem na osnovi instanc lahko za iskanje uporabljajo tudi zunanje informacije, npr. informacije iz zgodovine izvedenih preslikav. Algoritmi lahko uporabljajo baze, ki hranijo podatke o tem, kateri podatki se najpogosteje pojavijo pri izbrani ključni besedi. Kot primer navedimo besedo "Apple", ki je pogosta pri elementih z imenom *Proizvajalec*, *Podjetje*, *Organizacija* itd. Če element sheme S2 z imenom X vsebuje podatek "Apple", bo algoritem preveril, če pri tem podatku v bazi vsebuje ime elementa, ki se hkrati pojavi v shemi S1. Če to velja, pomeni, da je našel ustrezno preslikavo. V tem primeru je pri takem načinu integracije dovolj že to, da se "Apple" pojavi le v eni instanci S2 in ni potrebno, da se neka vsebina pojavi večkrat.

Pristopi za integriranje shem na osnovi instanc so najprimernejši pri iskanju preslikav številnosti 1:1. Iskanje preslikav za množice elementov zahteva karakterizacijo kombinacij elementov, pri čemer problem predstavlja veliko število kombinacij, ki jih je potrebno ovrednotiti.

Ker se v naši magistrski nalogi osredotočamo na iskanje ujemanj med shemami na osnovi podatkovnih instanc, si nekoliko podrobneje poglejmo dva primera iskalnikov, ki ju bomo uporabili za primerjavo konceptov. Prvi primer iskalnika deluje na osnovi regularnih izrazov [2], kjer avtorji predlagajo uporabo regularnih izrazov za opis formata podatkov, ki jih hrani nek element. Druga metoda je zasnovana na osnovi razdvojevanja podatkov in uvaja evolucijski algoritem za iskanje kompleksnih preslikav; to so preslikave številnosti 1:N in N:1. Pristopa si bomo pogledali zaradi primerjave konceptov z našo metodo.

Integriranje shem z uporabo regularnih izrazov

Regularni izraz je sekvenca znakov, ki definira vzorec. Regularne izraze običajno uporabljamo za iskanje tekstovnih vzorcev v nizu. Vzorec je zapisan z dvema tipoma znakov, z opisnimi (metaznaki: *?, *, +, .* itd.) znaki, ki imajo poseben pomen in regularnimi znaki, ki imajo dobesedni pomen

in predstavljajo vsebino vzorca. Primer enostavnega regularnega izraza je *[a-zA-Z]+*, ki definira vzorec teksta, ki vsebuje znake iz intervala angleške abecede, kateri se zaporedoma pojavi vsaj enkrat. Predstavljen primer regularnega vzorca je ustrezen za opis formata vsebine elementa, npr. *Ime*, *Priimek* ali *Podjetje*, saj v primeru, da elementi vsebujejo enobesedne vrednosti, vzorec ne vsebuje presledka. V primeru, da regularni izraz preveč ohlapno definira vsebino, lahko pride do tega, da zadostuje vsebini več elementov; tako primer, ki smo ga navedli, ustreza trem elementom. V bistvu želimo doseči, da vsak element v shemi opišemo z enoličnim regularnim izrazom, tako da se izognemo dvoumnim preslikavam.

Uporaba regularnih izrazov v primerjavi z drugimi metodami, ki smo jih omenili pri iskanju preslikav na osnovi podatkovnih instanc, prinaša naslednje prednosti:

- metoda je relativno hitra in ne zahteva predhodnega obdobja treniranja in učenja kot pristopi s strojnim učenjem ali pa nevronskimi mrežami;
- regularni izrazi predstavljajo učinkovit pristop za obravnavo uporabnikovega znanja o neki domeni.

Regularne izraze lahko uporabimo kot formaliziran pristop za karakterizacijo vsebine elementov sheme. Na podlagi vzorca podatkov lahko z analizo identificiramo vzorec oziroma format podatkov, ki jih hrani element. Iskanje preslikav z drugo shemo izvedemo tako, da vzamemo instanco podatka iz sheme in jo primerjamo z regularnim izrazom. Če se vsebina ujema z izrazom, potem smo identificirali preslikavo. Pristop, ki ga predstavljajo avtorji v [2], je sestavljen iz dveh faz. Prva faza je namenjena generiranju regularnih izrazov vhodne sheme za vsak element. Postopek za generiranje regularnih izrazov je sestavljen iz naslednjih korakov. Najprej za vsak element naključno prebere podatek in identificira njegov tip. Predlagana metoda ignorira posebne znake, kot recimo '@'. V naslednjem koraku metoda generira regularni izraz za vsako vrednost, razen za vrednosti podatkovnega tipa niz. Metoda nize obravnava tako, da jih razdeli v žetone, pri čemer se v drugi fazi upo-

rabi le prvi žeton niza. Rezultat prve faze je tako seznam regularnih izrazov za vse elemente sheme. Tak seznam služi kot vhod v drugo fazo algoritma. V tej fazi iskanja ujemanj se vsak regularni izraz uporabi za testiranje ujemanja z naključno vrednostjo elementa ciljne sheme. Če se vrednost ujema z regularnim izrazom, potem se element izvorne sheme ujema z elementom ciljne sheme. Nizi se pri tej metodi obravnavajo drugače, in sicer se primerjajo žetoni iz prve faze in podatki tipa niz v ciljni shemi. Ujemanje med elementoma obstaja, če niz vsebuje isti žeton.

Avtorji v članku ne predstavijo primerov, v katerih metoda na osnovi regularnih izrazov odpove, zato to poskušamo identificirati sami, in sicer zaradi primerjave predlaganega koncepta z našo metodo. Predpostavimo, da znamo generirati regularni izraz za neko množico podatkov nekega elementa ne glede na podatkovni tip. Kot primer vzemimo sheme in regularne izraze za posamezne elemente, prikazane v tabeli 2.3. Naj vhod v metodo predstavlja shema S1, kar pomeni, da za shemo S2 v tem primeru ne potrebujemo regularnih izrazov, vendar smo jih vseeno predstavili zaradi lažje razlage. En način iskanja preslikav bi lahko izvedli s primerjanjem regularnih izrazov, vendar je bolj priročno, če uporabimo podatek iz druge sheme in preverimo, ali se ta ujema s katerim izmed elementov sheme S1. Recimo, da imamo v entiteti zapisano osebo z imenom “Janez”. To vrednost tedaj primerjamo z vsemi regularnimi izrazi v shemi S1. Algoritem bo našel ujemanje z elementom *ImeOddelka*, kar ni pravilno. Problem se namreč pojavi pri regularnem izrazu elementa *ImeZap*, za katerega vemo, da ima pravilno preslikavo. Regularni izraz za element *ImeZap* opisuje niz dveh besed, kjer sta besedi ločeni z vsaj enim presledkom. Omenjeni vzorec je primeren izključno tedaj, ko element hrani ime in priimek, in sicer ob postavki, da nimamo dvojnih imen in priimkov. Algoritem ravno tako ne bo našel preslikave med rojstnima datumoma zaradi različnih formatov. Edini preslikavi, ki ju bomo našli s to metodo, sta preslikava med *Oddelek* in *ImeOddelka* ter z veliko verjetnostjo preslikava med *StZap* in *OSt*.

Metodo lahko razširimo z zunanjimi podatki, ki vsebujejo regularne izraze

elementi sheme S1	
Zaposleni	
StZap	[0-9]{1,10}
ImeZap	([a-zA-Z]+){3,}\s+([a-zA-Z\-]+){3,}
StOddelka	[0-9]{1,2}
Plača	[0-9]+(\.[0-9][0-9]?)?
Rojstni_datum	[0-9]{1,2}-[0-9]{1,2}-[0-9]{4}
Oddelek	
StOddelka	[0-9]{1,2}
ImeOddelka	[A-Za-z]{3,10}
elementi sheme S2	
Osebje	
OSt	[0-9]{1,10}
OIme	([a-zA-Z\-]+){3,}
Oddelek	[A-Za-z]{3,10}
Datum_rojstva	[0-9]{1,2}/[0-9]{1,2}/[0-9]{4}

Tabela 2.3: Primer iskanja preslikav z regularnimi izrazi.

za določene elemente, npr. za telefonske številke, kjer hranimo regularne izraze za vse poznane formate. Podobno velja tudi za datume. Tako bi v prejšnjem primeru našli tudi ujemanje med elementoma *Rojstni_datum* in *Datum_rojstva*.

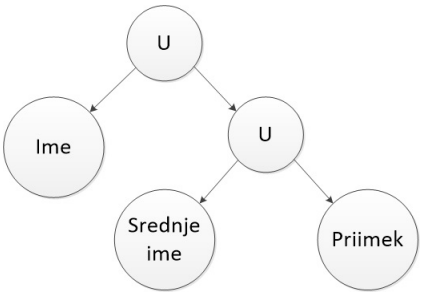
Poglejmo še preslikave številnosti 1:*N*. Za primer vzemimo elemente *Ime* in *Priimek*, ki sta del ene sheme, ter element *ImeInPriimek*, ki pripada drugi shemi. V tem primeru imamo preslikavo številnosti 1:*N*. Regularni izraz za opis vsebine elementa *ImeInPriimek* bi v splošnem predstavljal niz dveh besed, ki sta ločeni s presledkom. Metoda zato v tem primeru odpove ravno zaradi vmesnega presledka. Če bi primerjali instanco elementa *Ime*, ne bi dobili ujemanja ob predpostavki, da je ime sestavljeno le iz ene besede, podobno pa velja tudi za *Priimek*. Na podlagi predstavljenega primera lahko torej sklepamo, da bo predlagana metoda na ta način redko uspešna. Metodo bi bilo potrebno razširiti z algoritmom, ki bi znal poiskati kombinacijo elementov in jo tudi ustrezno predstaviti z enim, skupnim regularnim izrazom.

Integriranje shem z razdvojevanjem podatkov

Avtorji članka v [3] za iskanje kompleksnih preslikav predlagajo metodo, ki je osnovana na evolucijskem algoritmu, deluje pa na osnovi iskanja podvojenih podatkov. S podvojenimi podatki mislimo na podatke, pojavljajoče v vseh sistemih, ki jih integriramo. To pomeni, da spada pod iskalnike, ki uporabljajo podatke podatkovnih instanc. Ideja evolucijskih tehnik izhaja iz procesa naravne selekcije, ki vpliva na vsa živa bitja. Predlagana metoda temelji na tehniki genetskega programiranja [14] (v nadaljevanju GP). GP si lahko predstavljamo kot prilagodljivo hevristiko, pri čemer gre za neposredno evolucijo algoritmov za nadzorovano učenje, ki jih uporabljamo za reševanje optimizacijskih problemov. Metoda GP se zelo dobro obnese pri iskanju rešitev v zelo velikem iskalnem prostoru. Vsi problemi so predstavljeni in interpretirani kot programi, vključno s podatki.

Metoda GP v procesu razvija populacijo podatkovnih struktur, ki jim pravimo posamezniki (*individuals*). Vsak posameznik predstavlja eno rešitev problema. Če želimo aplicirati metodo GP na problem iskanja kompleksnih preslikav, moramo rešitev modelirati z neko podatkovno strukturo. Glede na to, da se ukvarjamo s shemami, je najbolj naraven način predstavitev z drevesno podatkovno strukturo, pri čemer mora ta omogočati manipulacijo strukture s strani procesa GP. Drevo modelira soodvisnosti in kombinacije elementov sheme. Elementi so prisotni v listih drevesa, v središču so vozlišča, ki določajo operacijo med elementi. Kot primer, predstavljen na sliki 2.3, navedimo unijo dveh elementov (npr. *ime* in *priimek*). Za metodo GP je potrebno definirati tudi terminale in funkcije. Terminali so vhodi, konstante ali vozlišča brez argumentov, ki prekinejo vejo v drevesu - določajo konec veje drevesa (listi). Množica funkcij predstavlja zbirko operacij in izrazov, ki so namenjeni procesu GP za manipulacijo terminalov. Ti lahko zasedajo notranja vozlišča drevesa.

V procesu evolucije se nad posamezniki izvajajo genetske operacije. To so operacije reprodukcije (*reproduction*), križanja (*crossover*) in mutacije (*mutation*). Evolucija se izvaja iterativno, kar pomeni, da v vsaki iteraciji z neko



Slika 2.3: Model za prikaz kombinacij in relacij med elementi sheme, kot ga uporablja metoda GP.

verjetnostjo generira boljšo rešitev problema (generira se boljši posameznik). Operacija reprodukcije zagotovi, da evolucijski proces skozi izvajanje ohranja dobre rešitve, ki jih je evolucijski proces našel v zgodnji fazi. Proces križanja je namenjen generiranju novih posameznikov. V predlagani metodi se križanje izvaja tako, da se izbereta dve drevesi (posameznika), v katerih se naključno izbirajo veje, ki se zamenjujejo. Operacija mutiranja skrbi za minimalno razliko med posamezniki v populaciji. To se izvede tako, da se v drevesu naključno izbere vozlišče in zamenja z naključno generiranim poddrevesom. Vsaka rešitev, generirana v fazi križanja, ima enako verjetnost, da se nad njo izvede mutacija.

Evolucijski algoritem predstavljen v [3] je sestavljen iz naslednjih korakov:

1. **Inicializacija populacije**: Populacija je lahko generirana naključno ali pa jo določi uporabnik. Pri generiranju populacije je potrebno biti pazljiv pri določanju njene velikosti.

2. **Vrednotenje posameznikov**: Vrednotenje vseh posameznikov v populaciji. Vsak posameznik dobi oceno, ki pove, kako dobro rešitev predstavlja.

3. **Preveri pogoj za ustavitev**: Če je izpolnjen pogoj za ustavitev, skoči na korak 7, sicer nadaljuj.

4. **Reprodukcija kandidatov**: Reproduciranje najboljših n rešitev v populacijo naslednje generacije.

5. **Izbor najboljših posameznikov**: Izbor m posameznikov, ki bodo sestavljali naslednjo generacijo z najboljšimi starši.

6. **Izvedba križanja in mutacije**: Apliciranje genetskih operacij križanja in mutacije nad izbranimi posamezniki. Njuni potomci predstavljajo naslednjo generacijo populacije. Algoritem se za tem vrne na korak 2.

7. **Rezultati**: Vrni najboljše rezultate iz populacije.

Naloga drugega koraka je, da oceni, kako dobro predlagan posameznik rešuje problem ujemanja med shemami. Na podlagi te ocene se algoritem odloči, kateri posamezniki so najprimernejši za naslednjo generacijo. Iz vidika iskanja preslikav med shemami je to korak, ki nas najbolj zanima. Naloga GP je torej ta, da s pomočjo evolucije pride do drevesa, ki predstavlja strukturo in odvisnosti med elementi izvorne sheme, ki se najbolje ujemajo s strukturo ciljne sheme.

Omenili smo, da predlagana metoda spada v razred pristopov, ki delujejo na osnovi podatkovnih instanc. Avtorji metode predlagajo dva pristopa za ocenitev kakovosti preslikave, pri čemer obe temeljita na osnovi podatkovnih instanc. Prva metoda temelji na entitetno orientirani strategiji za razdvojevanje podatkov, ki so jo predlagali isti avtorji v [4]. Druga metoda temelji na vrednostno orientirani strategiji, ki je osnovana na tehniki pridobivanja informacij (*information retrieval technique*).

Entitetno orientirana strategija temelji na naslednji ideji: če obstajajo skupne entitete v instancah dveh shem, med katerima iščemo preslikavo, potem bomo v primeru pravilne preslikave taki instanci prepoznali. Naj I_A predstavlja instanco sheme A in I_B instanco sheme B. Hkrati naj obstajata neprazni množici entitet $\{e_1,..., e_n\}$ iz I_A in $\{f_1,.., f_n\}$ iz I_B, kjer velja $e_j = f_j (1 \leq \mathrm{j} \leq \mathrm{n})$. Potem mora obstajati vsaj ena preslikava, za katero velja, da je podobnost med shemama zelo visoka. Torej, če imamo dve instanci, kjer ena hrani podatke v elementih *Ime* in *Priimek*, druga pa v elementu

ImeInPriimek, in ker predpostavljamo, da so podatki isti, potem bi morala obstajati entiteta, npr. "Janez Novak", v obeh shemah. Ko bo evolucijski algoritem našel pravo kombinacijo in razmerje med elementoma *Ime* in *Priimek*, bo algoritem ocenil zelo dobro ujemanje, kar pomeni, da je našel preslikavo med shemama. Avtorji izpostavijo, da je slabost te metode njena dovzetna do tega, da moramo imeti v obeh podatkovnih virih zelo podobne podatke, poleg tega je zelo odvisna od tehnik za kombiniranje zapisov in razdvojevanja podatkov.

Vrednostno orientirana strategija deluje nekoliko drugače. Deluje tako, da primerja množico vrednosti elementov iz obeh podatkovnih instanc. Naj bo d_A element sheme A in d_B element sheme B, oba istega podatkovnega tipa. Imejmo hkrati dve podatkovni instanci, eno za shemo A (I_A) in drugo za shemo B (I_B). Naj bo $\mathcal{V}(d_X) = \{\mathrm{v}(d_X, \mathrm{e}) \mid \mathrm{e} \in I_X\}$ množica vrednosti elementa d_X iz vseh entitet instance I_X, kjer $\mathrm{v}(d_X, \mathrm{e})$ predstavlja vrednost elementa d_X v entiteti e in $X \in \{A, B\}$. Če instanci I_A in I_B veljata za reprezentativna vzorca, potem z veliko verjetnostjo obstaja vsaj ena preslikava, za katero velja, da sta $\mathcal{V}(d_A)$ in $\mathcal{V}(d_B)$ podobna.

Kot primer predpostavimo, da ima instanca sheme A naslednje entitete za element *Ime* = ("Rose", "Katheri", "Mary") in za element *Priimek* = ("Leslie", "Hand", "White"), shema B pa naslednje elemente *ImeInPriimek* = ("Leslei Rose", "Katherine Hand", "Mary Wite"). Recimo, da algoritem z evolucijo privede do tega, da se elementa *Ime* in *Priimek* iz sheme A združita in ustvari element d_A. Tedaj se generirata naslednji množici:

- $\mathcal{V}(d_A) = \{$"Rose Leslie", "Katheri Hand", "Mary White"$\}$
- $\mathcal{V}(d_B) = \{$"Leslei Rose", "Katherine Hand", "Mary Wite"$\}$.

Iz primera je jasno razvidno, da podatki niso čisto podobni. Kljub temu lahko z ustrezno metriko ocenimo dobro ujemanje. Vidimo, da je v primerjavi z entitetno orientirano strategijo, ki potrebuje v obeh instancah isti zapis, ta pristop boljši. Za izračun podobnosti med dvema množicama so zato

primerne metrike, kot Jaccardova razdalja (*Jaccard distance* [22]) ali TF-IDF (*term frequency-inverse document frequency* [15]).

Če povzamemo rezultate metode na primerih, lahko rečemo, da se ta zelo dobro izkaže v primerih, ko so podatki v obeh instancah praktično isti. Pri delni podobnosti podatkov se natančnost najdenih preslikav zmanjša. Avtorji navajajo, da metoda najde 75% kompleksnih preslikav in v povprečju 60% enostavnih preslikav. Pri testiranju na disjunktnih podatkih so rezultati natančnosti še nekoliko slabši, kar je natančneje razloženo v [3].

Predlagana metoda na osnovi genetskega programiranja je ena redkih metod, ki se ukvarja tudi z iskanjem kompleksnih preslikav (številnosti 1:N in N:1) med podatkovnimi shemami. Metoda se izkaže pri iskanju preslikav med dvema instancama z zelo podobnimi podatki.

2.2.4 Iskanje ujemanj s ponovno uporabo shem in informacij o preslikavah

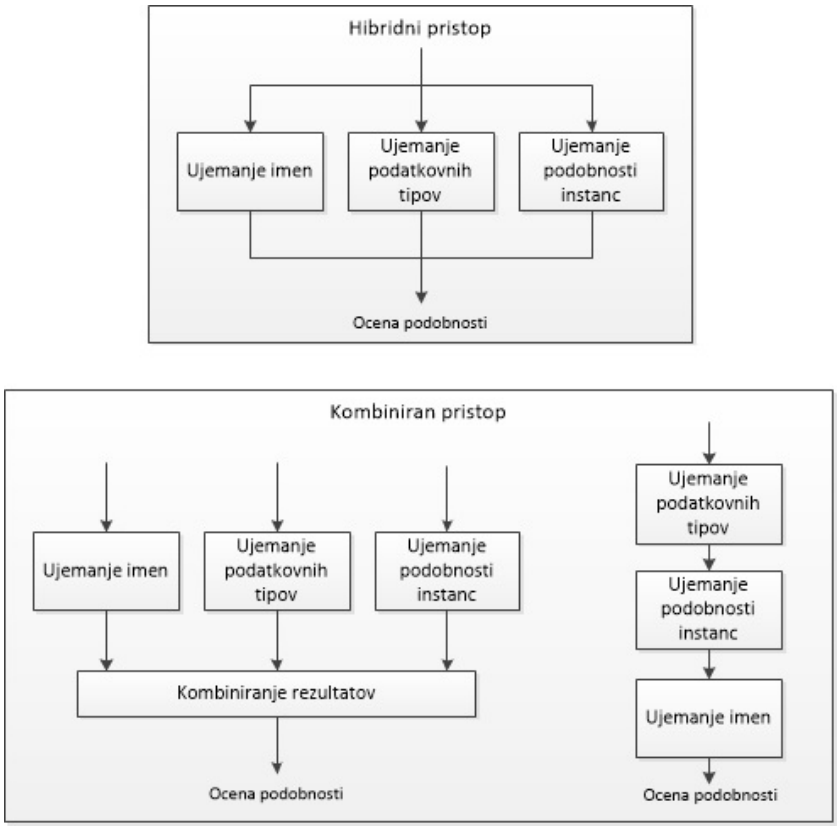
Omenili smo že uporabo zunanjih virov informacij, kot so slovarji in druge informacije, ki jih zagotovi uporabnik. Učinkovitost algoritmov za iskanje ujemanj lahko izboljšamo tudi s ponovno uporabo skupnih komponent shem in preslikav, ki so bile identificirane v preteklosti. Pristopi s ponovno uporabo informacij o preslikavah so smiselni, saj vemo, da so si v splošnem sheme med sabo zelo podobne. Dober primer so aplikacije elektronskega poslovanja, kjer se sheme pogosto ponavljajo (npr. naslovi, imena itd.). Primer pristopa s ponovno uporabo predstavlja že uporaba imenskih prostorov (*namespace*) ali uporaba specifičnih slovarjev. Bolj splošen pristop predstavljajo metode, ki delujejo na ponovni uporabi preslikav za celotno shemo ali del sheme, skupaj s podatkovnimi tipi, ključi in omejitvami. Uporabnost tega pristopa se izkaže pri uporabi pogostih shem kot *Naslov*, *Zaposleni*, *Naročilo*, *Stranka*, *Faktura*, ipd. Verjetno se strinjamo, da je težko pričakovati, da se bo celoten svet poenotil z neko shemo, je pa to realno na nivoju organizacije, podjetja in njegovih partnerjev ali podjetij v isti panogi [8].

2.2.5 Kombinirani pristopi za iskanje ujemanj

Oba pristopa, tako na osnovi podatkov shem kot na osnovi podatkovnih instanc, imata pomanjkljivosti, ki lahko privedejo do tega, da metoda ni uporabna. V primeru, da shema ni na voljo, ne moremo uporabiti metode, ki preslikave išče na osnovi teh podatkov. Ravno tako se lahko zgodi, da metoda na osnovi instanc ni uporabna, če sistema, ki ju integriramo, ne hranita nobenih podatkov. Uporaba le enega pristopa za integriranje podatkovnih shem zato v večini primerov ne zadostuje. Opazili smo še eno zanimivo lastnost, in sicer se pogosto zgodi, da je pri odpovedi enega pristopa lahko drugi pristop učinkovit, kar pomeni, da se pristopi zelo dobro dopolnjujejo. To je zelo dobro izhodišče za uporabo kombiniranih pristopov. Kombinirane pristope lahko uporabimo na dva načina. Prvi način predstavljajo hibridni pristopi, kjer integriramo kriterije, na podlagi katerih iščemo ujemanja, drugi način predstavljajo pristopi, kjer kombiniramo rezultate neodvisnih iskalnikov, ki se izvedejo vzporedno.

Hibridni iskalniki ujemanj neposredno kombinirajo več pristopov in ujemanja iščejo na podlagi več kriterijev in več virov informacij (npr. podobnost imen, slovarjev in podatkovnih tipov elementov). Na ta način lahko odkrijejo več preslikav in dosežejo večjo natančnost. Boljšo učinkovitost dosežemo ravno z uporabo več kriterijev, ki poskrbijo, da slaba ujemanja hitro izpadejo iz seznama potencialnih kandidatov za preslikavo. Pristop z iskanjem preslikav na osnovi podobnosti struktur lahko kombiniramo s pristopom iskanja na podlagi podobnosti imen elementov, kjer prvi pristop uporabimo, da identificiramo okvirne preslikave, drugega pa za identificiranje končnih preslikav.

Uporaba kombiniranih pristopov omogoča združevanje rezultatov posameznih iskalnikov ujemanj. V primerjavi s hibridnimi pristopi je ta veliko bolj fleksibilen, pri čemer so hibridni pristopi veliko bolj prepleteni oziroma tesno povezani, saj so združeni v en skupni iskalnik preslikav. Prednost kombiniranih pristopov je, da lahko izbiramo kombinacijo iskalnikov glede na posamezen primer, pri čemer upoštevamo dejstvo, da se nekateri pristopi



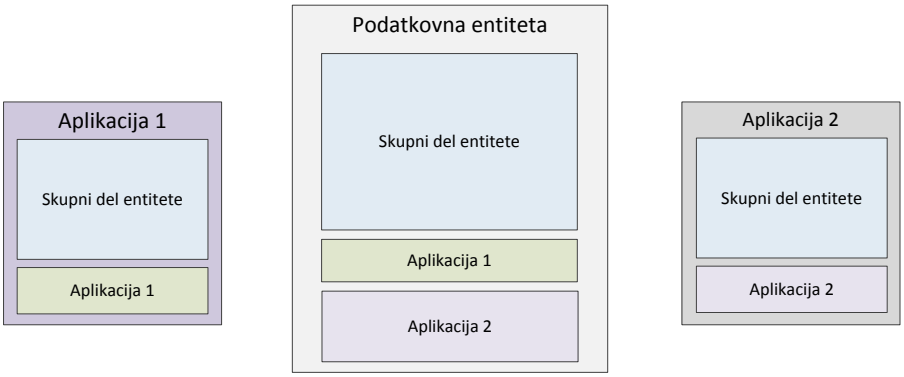
Slika 2.4: Strukturna razlika med hibridnimi in kombiniranimi pristopi iskalnikov ujemanj med podatkovnimi shemami.

v določenih primerih izkažejo za učinkovitejše. Za iskalnike ujemanj s kombiniranimi pristopi je zato primerno, da uporabniku omogočijo možnost, kako se naj posamezni iskalniki izvajajo: bodisi vzporedno bodisi zaporedno, in kateri iskalniki se naj uporabijo, pri čemer uporabniku ponudi nabor iskalnikov ujemanj. V primeru zaporednega izvajanja se rezultati enega iskalnika uporabijo kot vhod v drugega, kot to določa vnaprej določeno zaporedje, npr. rezultati iskalnika na podlagi podobnosti imen se uporabijo za vhod iskalnika, ki deluje na osnovi podatkovnih instanc. Strukturne razlike med pristopoma so prikazane na sliki 2.4.

Poglavje 3

Arhetipska analiza za povzemanje podatkovnih množic

Spoznali smo različne pristope in metode za integriranje podatkovnih shem ter njihovo uporabnost v različnih primerih. V praksi se pogostokrat srečujemo z integracijo različnih aplikacij in sistemov, ki hranijo določen delež istih podatkov in jih v večini primerov strukturirajo drugače, zato potrebujemo metode za avtomatizirano integriranje shem, na podlagi katerih lahko ustrezno transformiramo podatke za prenos iz ene aplikacije v drugo. Integriranje podatkovnih shem ni potrebno le pri prenosu iz ene aplikacije v drugo, pač pa se velikokrat srečujemo tudi s problemom, kako združiti podatke neke entitete iz različnih virov in prikazati 360° pogled nad vsemi podatki. V tem primeru moramo ugotoviti, kateri deli podatkov se prekrivajo in kateri deli predstavljajo razširitve skupnega dela. To seveda lahko ugotovimo z iskanjem preslikav, kot to prikazuje primer na sliki 3.1. Metode, ki smo jih predstavili v prejšnjem poglavju, lahko uporabljajo podatke sheme, podatke instanc, ali celo kombinacijo obeh. Zadnja v večini primerov predstavlja najboljši pristop, saj se metode med sabo zelo dobro dopolnjujejo. Res je, da so metode, ki uporabljajo le podatke sheme, zelo omejene s količino podatkov,



Slika 3.1: Prikaz dveh aplikacij in skupne entitete z ilustracijo delov entitete, ki se prekrivajo in delov, ki so specifični za aplikacijo. Naloga preslikovalnika pri gradnji skupnega podatkovnega modela je, da na podlagi preslikav odkrije skupni del entitete aplikacij 1 in 2. Tisti deli, ki nimajo preslikav, predstavljajo razširitve skupne entitete.

ki jih imajo na razpolago.

V določenih primerih podatkov shem ne moremo uporabiti za iskanje preslikav, ker sheme preprosto niso na voljo, npr. zaradi varnostnih razlogov, ali so podatki le delno strukturirani. Za te primere predlagamo metodo za iskanje preslikav, ki temelji na podatkovnih instancah. Predlagana metoda temelji na povzetkih podatkov posameznih elementov sheme. Za povzemanje podatkov elementa uvedemo metodo arhetipske analize, s katero bomo izračunali približek konveksne ovojnice podatkov. Metodo podrobneje predstavimo v podpoglavju 3.1. Poudarimo, da s predlagano metodo želimo odkriti tudi kompleksne preslikave številnosti 1:N in N:1. Naš cilj je doseči visoko natančnost odkritih preslikav samo na osnovi podatkovnih instanc. Podatke sheme bomo uporabili le, če se bi to izkazalo za nujno potrebno.

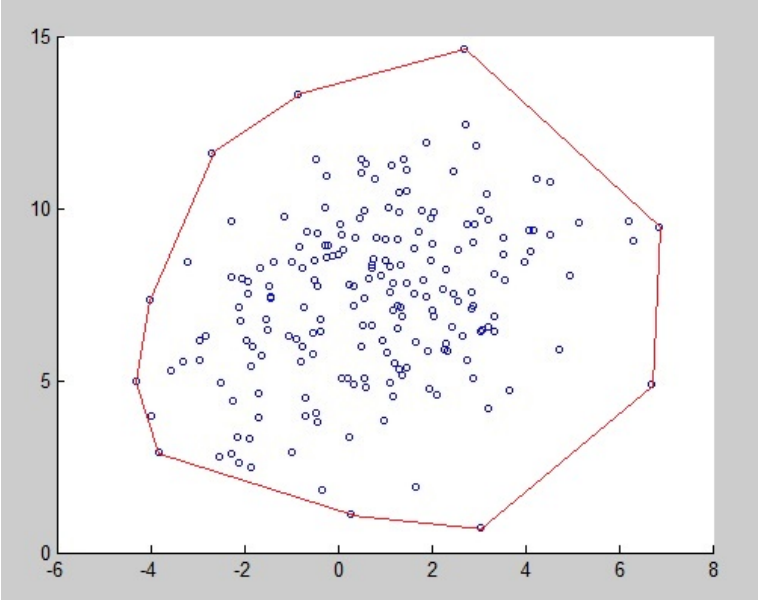
Pri raziskovanju obstoječih metod smo odkrili, da ima večina metod probleme z odkrivanje preslikav med elementi z istim podatkovnim tipom. Metode tako npr. težko razlikujejo med dvema elementoma tipa niz ali pa decimalno število, kjer npr. predstavljena metoda z regularnimi izrazi v teh

primerih hitro odpove. Naša metoda sicer temelji na predpostavki, da iščemo preslikave med sistemi z zelo podobnimi podatki in bi morala biti sposobna razlikovati tudi te primere. Kljub temu bomo še dodatno poskusili zagotoviti, da bo metoda sposobna najti preslikave med instancama, ki imata nekoliko manj podobne podatke, kar bomo poskušali doseči preko značilk, ki jih bomo pridobili iz podatkov.

Metoda arhetipske analize je na področju rudarjenja podatkov dokaj neuveljavljena. V uvodu smo omenili, da je bila uporabljena za iskanje povzetkov tekstovnih dokumentov, kjer se je izkazala za eno najboljših na področju. Gre za to, da se podatki predstavijo z množico arhetipov. Pomen besede arhetip si lahko razlagamo kot originalen vzorec ali model (prototip), na katerem so osnovane vse stvari istega tipa. Arhetip izvira iz platonske filozofije, navezujoče se na čisto obliko, ki izraža temeljne karakteristike neke stvari. V primeru povzemanja tekstovnih dokumentov gre za iskanje stavkov in povedi, ki v največji meri povzemajo celoten dokument. V našem primeru bodo arhetipi podatki elementa, ki najbolje opišejo celotno podatkovno množico elementa. V nadaljevanju razlagamo delovanje algoritma za iskanje povzetkov in predstavimo metodo za integriranje podatkovnih shem.

3.1 Arhetipska analiza

Metodo arhetipske analize (v nadaljevanju AA) sta prvotno predstavila avtorja Cutler in Breiman [9] kot novo metodo za reduciranje dimenzij multivariantnih (*multivariate*) podatkov. Osnovna ideja metode je, da vsako točko znotraj konveksne ovojnice predstavi kot približek s konveksno kombinacijo arhetipov. Če si konveksno ovojnici predstavljamo v dvodimenzionalnem prostoru, kjer točke predstavimo z žebljički, je konveksna ovojnica elastika, s katero ovijemo vse točke. Metoda zahteva, da so arhetipi množica posameznih podatkovnih točk, ki jih je enostavno interpretirati, za razliko od metod kot so PCA (Principal component analysis) [12], ki za rezultat vrnejo elemente, kateri nimajo fizičnega pomena. V primerjavi z metodo, ki



Slika 3.2: Podatkovna množica predstavljena z 200 točkami in njena konveksna ovojnica. Točke, ki ležijo na konveksni ovojnici imenujemo arhetipi. Z ustrezno linearno kombinacijo arhetipov lahko predstavimo katerokoli točko znotraj konveksne ovojnice.

temelji na nenegativni faktorizaciji matrik (NMF - *Non-negative matrix factorization* [13]), s katero pridobimo karakteristične informacije o podatkovni množici, AA vrne reprezentativne vogale podatkovne množice. Na sliki 3.2 je prikazana podatkovna množica 200 točk in njena konveksna ovojnica.

Metode iz področja podatkovnega rudarjenja, med katere spadajo razcep na singularne vrednosti (SVD - *singular value decomposition*), PCA, analiza neodvisnih komponent (ICA - *independent component analysis*), NMF, metode za mehko gručenje kot k-means z mehko logiko ali pa EM (*expecation-maximization*) algoritmi, pa tudi metode za gručenje s trdim določanjem kot k-means, imajo skupno lastnost, in sicer, da podatke predstavijo kot linearno kombinacijo ali s faktorji in varianco, kot pri faktorski analizi [19] z različnimi omejitvami. Kljub temu da so našteti pristopi v osnovi zelo podobni, se njihova jedrna predstavitev podatkov precej razlikuje in s tem tudi

njihova narava analize. Pri SVD in PCA metodah značilke predstavljajo smer največje variance, pri NMF značilke predstavljajo sestavne dele podatkov, pri metodah gručenja pa najbolj reprezentativne prototipne objekte.

Prednost metod za gručenje je ta, da so značilke precej podobne podatkom, kar jih naredi enostavne za interpretacijo. Njihova slabost je pomanjkanje fleksibilnosti zaradi binarnega (diskretnega) določanja pripadnosti. Na drugi strani imamo metode na osnovi aproksimacije, kot so SVD/PCA/NMF, ki so zelo fleksibilne in učinkovitejše, vendar je običajno zaradi kompleksnosti težko interpretirati podatke. Arhetipska analiza združuje enostavno interpretacijo rezultatov, ki je značilna za metode gručenja in fleksibilnost metod z razcepom matrik.

Cutler in Breiman za identifikacijo arhetipov predlagata metodo z minimizacijo napake pri predstavitvi podatkovne množice s kombinacijo arhetipov. V [9] sta dokazala, da minimalno napako dobimo takrat, ko arhetipe predstavljajo ekstremne točke, ki ležijo na konveksni ovojnici podatkovne množice. Predlagata iterativni optimizacijski algoritem, ki izmenično rešuje konveksna problema najmanjših kvadratov, dokler ne doseže dovolj majhne napake ali preseže število vnaprej določenih iteracij.

Predstavitev podatkov s konveksno kombinacijo arhetipov nudi zanimive možnosti na področju razpoznavanja vzorcev. Avtorji v [10] predstavijo, kako metodo učinkovito prilagoditi za uporabo na področju rudarjenja podatkov. Iskanje konveksne ovojnice je namreč kompleksen problem, zato potrebujemo učinkovit algoritem, ki je zmožen procesirati velike količine podatkov. Problem iskanja konveksne ovojnice je rešljiv v $\mathcal{O}(N)$, pri čemer velikost konveksne množice drastično narašča z dimenzijo podatkov. Pričakovana velikost konveksne množice za N točk v M dimenzionalnem prostoru raste eksponentno z dimenzijo in je enaka $\mathcal{O}(log(N)^{M-1})$.

Metoda arhetipske analize je že bila uporabljena na področju računalniškega vida za iskanje podmnožice slik obrazov, s katero lahko opišemo vse ostale obraze v celotni množici (z ustrezno konveksno kombinacijo), uporabljena je bila tudi na področju kemije, rudarjenja tekstovnih podatkovnih množic, za

priporočilne sisteme na osnovi filtriranja s sodelovanjem (*collaborative filtering*) itd. V tem delu jo bomo vpeljali še na področju integriranja podatkovnih shem.

3.1.1 Formalna predstavitev metode

Imejmo množico podatkov $X = \{x_1, x_2, ..., x_m\}$ kjer $x_i \in \mathbb{R}^n$, kjer m predstavlja število podatkov in n dimenzijo prostora. Metoda arhetipske analize se ukvarja z iskanjem l arhetipov $\{z_1, z_2, ..., z_l\}$, pri čemer velja $l \ll m$. Arhetipi so določeni z linearno kombinacijo podatkovnih točk

$$z_j = \sum_{i=1}^{n} x_i c_{ij}, \tag{3.1}$$

kjer velja $c_{ij} \geq 0$, kar pomeni, da arhetipi kar najbolje pokrivajo podatke, hkrati mora veljati tudi $\sum_i c_{ij} = 1$, kar pomeni, da množica predstavlja konveksno kombinacijo podatkov. Arhetipska analiza za dano množico arhetipov minimizira izraz

$$\|x_i - \sum_{j=1}^{l} z_j s_{ji}\|^2 \tag{3.2}$$

tako, da poišče take koeficiente s_{ji}, ki v največji meri pokrivajo podatke x_i z arhetipi. Tudi v tem primeru mora veljati $s_{ij} \geq 0$, tako da je vsaka podatkovna točka predstavljena s smiselno kombinacijo arhetipov, in $\sum_j s_{ji} = 1$, da so podatkovne točke predstavljene kot kombinacija arhetipov. Ustrezna množica arhetipov tako minimizira rezidualno vsoto kvadratov:

$$RSS(l) = \sum_{i=1}^{n} \|x_i - \sum_{j=1}^{l} z_j s_{ji}\|^2 = \sum_{i=1}^{n} \|x_i - \sum_{j=1}^{l} \sum_{k=1}^{n} x_k c_{kj} s_{ji}\|^2. \tag{3.3}$$

Če enačbo predstavimo v matrični obliki, kjer so podatki $X \in \mathbb{R}^{m \times n}$ in arhetipi $Z \in \mathbb{R}^{m \times l}$, lahko enačbo (3.3) zapišemo kot:

$$RSS(l) = \|X - ZS\|^2 = \|X - XCS\|^2, \tag{3.4}$$

kjer sta $S \in \mathbb{R}^{l \times n}$ in $C \in \mathbb{R}^{n \times l}$ stolpično stohastični matriki.

Avtorji članka [11] na osnovi članka [9] predlagajo algoritem za iskanje arhetipov s postopkom projiciranega gradienta (*projected gradient procedure*). Iz članka Cutler in Breiman [9] je razvidno, da problem iskanja arhetipov ni rešljiv z enačbo zaprte oblike. Zato moramo za reševanje problema uporabiti pristop z optimizacijskim algoritmom. Izhajamo iz trditve 3.0.1, ki pove, da iskanje ustrezne matrike S pri fiksni matriki C in obratno tvori dva konveksna optimizacijska problema z merjenjem napake po metodi najmanjših kvadratov, določene s Frobeniousovo normo [25]: $\|X - XCS\|_F^2$.

Trditev 3.0.1 *Za optimizacijo z metodo najmanjših kvadratov alternirajoča optimizacija C pri fiksiranem S in S pri fiksiranem C tvori konveksna optimizacijska problema.*

Dokaz. Vsak alternirajoči podproblem lahko trivialno zapišemo kot kvadratni optimizacijski problem z linearnimi nenegativnimi omejitvami, ki tako tvorijo konveksni problem. Pri optimizaciji matrike S je Hessova matrika (matrika drugih parcialnih odvodov) določena kot $H_S = \mathrm{I} \otimes [\ (XC)^T\ \mathrm{XC}\ ]$, za optimizacijo matrike C je določena kot $H_C = \mathrm{diag}(\mathrm{S}S^T) \otimes (X^T\ \mathrm{X})$, kjer je operacija $\otimes$ definirana kot $[A \otimes B]_{ij} = max([A]_{i1} + [B]_{1j}, ..., [A]_{ip} + [B]_{pj})$. V tem primeru sta obe Hessovi matriki pozitivno semidefinitni, kar velja natanko tedaj, ko je $xMx \geq 0$ za vsak $x \in \mathbb{R}^n$. □

V članku [11] avtorji za iskanje arhetipov predlagajo podobno metodo, kot se uporablja pri metodi NMF, in sicer projekcijo gradienta z uporabo pristopa invariantne normalizacije. Problem se pretvori v l_1-normirane invariantne spremenljivke:

$$\tilde{s}_{d,m} = \frac{s_{d,m}}{\sum_{d'} s_{d',m}} \tag{3.5}$$

in

$$\tilde{c}_{m,d} = \frac{c_{m,d}}{\sum_{m'} c_{m',d}}, \tag{3.6}$$

tako da je eksplicitno izpolnjen pogoj enakosti. Pri odvajanju enačb (3.5) in (3.6) velja

$$\frac{\partial \tilde{s}_{d',m}}{\partial s_{d,m}} = \frac{\partial_{d',d}}{\sum_d s_{d,m}} - \frac{s_{d',m}}{(\sum_d s_{d,m})^2} \tag{3.7}$$

in

$$\frac{\partial \tilde{c}_{m',d}}{\partial c_{m,d}} = \frac{\partial_{m',m}}{\sum_m c_{m,d}} - \frac{c_{m',d}}{(\sum_m c_{m,d})^2}. \tag{3.8}$$

Z odvajanjem po delih pridemo do naslednjih posodobitvenih izrazov za spremenljivke pri iterativnem postopku AA:

$$s_{d,m} \leftarrow max\{\tilde{s}_{d,m} - \mu_{\tilde{s}}(g^{\tilde{s}}_{d,m} - \sum_{d'} g^{\tilde{s}}_{d',m}\tilde{s}_{d',m}), 0\}, \tag{3.9}$$

$$\tilde{s}_{d,m} = \frac{s_{d,m}}{\sum_d s_{d,m}}, G^{\tilde{s}} = \tilde{C}^T X^T X \tilde{C}\tilde{S} - \tilde{C}^T X^T X, \tag{3.10}$$

$$c_{m,d} \leftarrow max\{\tilde{c}_{m,d} - \mu_{\tilde{c}}(g^{\tilde{c}}_{m,d} - \sum_{m'} g^{\tilde{c}}_{m',d}\tilde{c}_{m',d}), 0\} \tag{3.11}$$

in

$$\tilde{c}_{m,d} = \frac{c_{m,d}}{\sum_m c_{m,d}}, G^{\tilde{c}} = X^T X \tilde{C}\tilde{S}\tilde{S}^T - X^T X \tilde{S}^T. \tag{3.12}$$

Vsaka alternirajoča posodobitev se v enem koraku izvede nad vsemi elementi z uporabo matrične operacije. Parametra $\mu_{\tilde{c}}$ in $\mu_{\tilde{s}}$ predstavljata velikost koraka, ki ju lahko prilagajamo z metodo linearnega iskanja [24]. V [11] je predlagana implementacija z uporabo 10 korakov linearnega iskanja pri vsaki alternirajoči posodobitvi S in C, hkrati navedejo, katere dele enačb je mogoče izračunati vnaprej, kar privede k izboljšanju časovne kompleksnosti, ki je sedaj enaka algoritmu za NMF.

Pri posodobitvi vrednosti S lahko vnaprej izračunamo zvezi matričnih produktov $\tilde{C}^T X^T X$ in $\tilde{C}^T X^T X \tilde{C}$, pri čemer je časovna zahtevnost za izračun teh izrazov enaka $\mathcal{O}(dnm)$, kjer d predstavlja število arhetipov, n dimenzijo

prostora in m število vseh podatkov. Napako, katero želimo minimizirati, izračunamo z naslednjo enačbo:

$$\|X - XCS\|_F = \sum\sum X \circ X - 2\langle \tilde{C}^T X^T X, \tilde{S}\rangle + \langle \tilde{C}^T X^T X \tilde{C}, \tilde{S}\tilde{S}^T\rangle, \tag{3.13}$$

kjer operacija $\circ$ predstavlja Hadamardovo matrično množenje; velja tudi $\langle A, B\rangle = \sum_{ij} a_{ij} b_{ij}$. Izračun ima časovno zahtevnost $\mathcal{O}(d^2 m)$, s čimer je enaka časovni kompleksnosti za izračun gradientov.

Podobno lahko v procesu posodobitve vrednosti C vnaprej izračunamo $X^T C \tilde{S}^T$ s časovno kompleksnostjo $\mathcal{O}(dnm)$ in $\tilde{S}\tilde{S}^T$ s kompleksnostjo $\mathcal{O}(d^2 m)$. Napako v tem primeru izračunamo z enačbo:

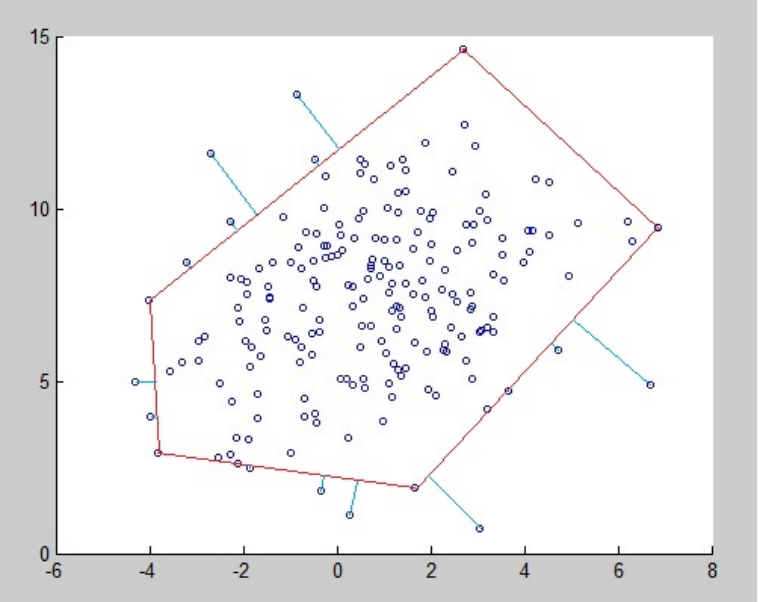
$$\|X - XCS\|_F = \sum\sum X \circ X - 2\langle X^T X \tilde{S}^T, \tilde{C}\rangle + \langle \tilde{C}^T X^T X \tilde{C}, \tilde{S}\tilde{S}^T\rangle, \tag{3.14}$$

s časovno zahtevnostjo $\mathcal{O}(dnm)$. Ti izrazi so zelo pomembni za učinkovito implementacijo algoritma za iskanje arhetipov.

Lastnosti pri obravnavi arhetipske analize

Izpostaviti je potrebno dve pomembni lastnosti AA. Prva lastnost se nanaša na inicializacijo arhetipov. V [9] lahko zasledimo, da začetni izbor arhetipov vpliva na hitrost konvergence in na verjetnost, da metoda odkrije nepomembne arhetipe. Da povečamo verjetnost za optimalno rešitev, avtorji v [11] za inicializacijo predlagajo metodo *FurthestSum*, ki poskrbi za to, da so izbrani arhetipi čim bolj oddaljeni eden od drugega. Metoda deluje tako, da se prvi arhetip iz množice podatkov izbere naključno. Naj bo $\mathcal{A}$ množica izbranih arhetipov. Naslednji element, ki ga vstavimo v $\mathcal{A}$, je izbran tako, da se za vse točke zunaj množice izračuna vsota razdalj do vseh točk v množici $\mathcal{A}$, pri čemer metoda izbere tistega, pri katerem je vsota razdalj največja. Na ta način zagotovimo, da so točke znotraj množice arhetipov pozicionirane čim bolj narazen.

Druga pomembna lastnost je izbira števila arhetipov, torej, s kolikšnim številom arhetipov želimo opisati oz. povzeti podatkovno množico. Ta neposredno določa, kako dobro bomo opisali množico. Če za število arhetipov izberemo število točk v konveksni ovojnici, je to optimalna rešitev, vendar se težava pojavi pri tem, da števila točk na konveksni ovojnici ne poznamo vnaprej. Na sliki 3.3 je prikazan primer rešitve, ko za število arhetipov določimo manjše število, kot je točk v konveksni ovojnici. Slika 3.3 sicer prikazuje arhetipe, kjer so arhetipi dejanske točke in ne linearne kombinacije ostalih točk, kot to velja za metodo uporabljeno v tej magistrski nalogi. Iz pogoja enačbe (3.1), ki smo ga upoštevali pri definiciji arhetipske analize, arhetipi niso nujno dejanske točke, ampak so točke, ki jih dobimo z linearno kombinacijo ostalih točk. Algoritem arhetipske analize deluje tako, da za vnaprej določeno število arhetipov za arhetipe določi tiste točke, za katere velja, da je rezidualna vsota (napaka) najmanjša. Katero koli točko znotraj konveksne ovojnice lahko predstavimo kot linearno kombinacijo arhetipov, točke zunaj konveksne ovojnice lahko predstavimo s približkom - projekcijo na rob ovojnice. Modre črte na sliki 3.3 predstavljajo napako. Naloga algoritma je, da izbere tiste točke, pri katerih je vsota napak najmanjša.



Slika 3.3: Na sliki vidimo približek konveksne ovojnice, kjer smo določili pet arhetipov. Iz slike 3.2 lahko razberemo, da se v optimalni konveksni ovojnici nahaja 10 arhetipov.

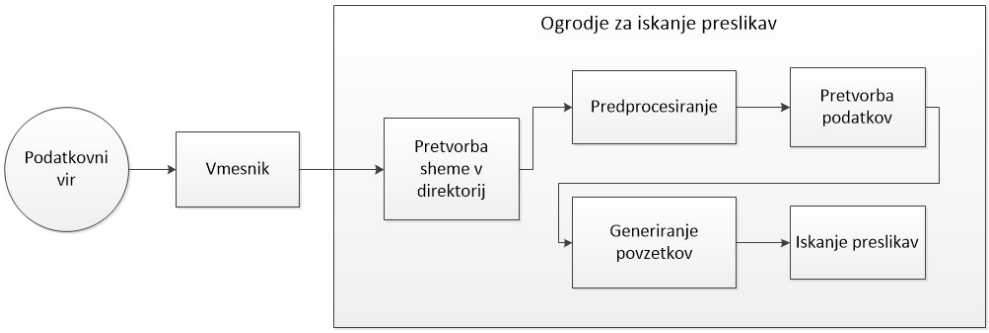
Poglavje 4

Arhitektura ogrodja za iskanje preslikav na osnovi povzetkov podatkovnih instanc z AA

Za avtomatizirano integriranje podatkovnih shem na osnovi podatkovnih instanc predlagamo ogrodje, ki bo primerno za integriranje poljubnih tipov podatkovnih shem, npr. XML, sheme različnih podatkovnih baz itd. Zgrajeno orodje kot rezultat vrne formalni zapis, ki predstavi preslikave med elementi vhodnih shem, določene s strani uporabnika. Neodvisnost od vira podatkov in jezika, s katerim je predstavljena shema, dosežemo z uporabo vmesnikov, ki poskrbijo za interpretacijo podatkov in sheme vira podatkov. Naloga vmesnika je, da shemo in podatke pretvori v obliko, ki jo določa in razume predlagano ogrodje. Konceptualni pogled na ogrodje je prikazan na sliki 4.1.

4.1 Vmesnik

Vmesnik je komponenta, preko katere ogrodje pridobi potrebne informacije za integriranje shem iz vseh podatkovnih virov. Vmesnik omogoči dostop do podatkovne sheme in podatkov. Hkrati mora poskrbeti za pretvorbo podat-

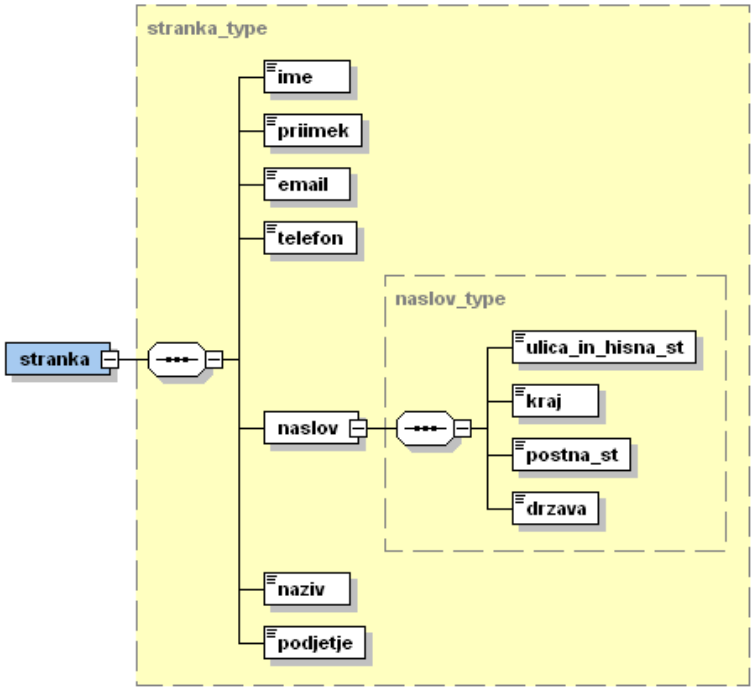


Slika 4.1: Visokonivojska predstavitev komponent ogrodja za avtomatiziramo integriranje shem.

kovne sheme v format, kot ga zahteva ogrodje za iskanje preslikav. Odločili smo se, da ogrodje za iskanje preslikav sheme interpretira kot sheme XSD (XML Schema Definition). To pomeni, da mora vmesnik dostaviti shemo v obliki XSD. Poleg sheme mora vmesnik zagotoviti tudi podatke. Ker je algoritem za izračun povzetka precej kompleksen, je zelo pomembno, kako izberemo vzorec podatkov. Zaradi hitrosti izvajanja želimo, da je vzorec čim manjši. Po drugi strani moramo vzorec izbrati tako, da čim bolj celovito pokrije podatke. Vmesnik lahko uporabi isto metodo, kot jo uporablja algoritem za generiranje povzetkov, v našem primeru metoda največje vsote kvadratov razdalj med točkami ($FurthestSum$). Vmesnik podatke ogrodju posreduje v XML obliki.

4.2 Pretvorba podatkovne sheme

Prvi korak, ki ga izvede ogrodje za preslikovanje, je, da znotraj delovnega imenika z mapami ustvari hierarhično strukturo podatkovne sheme, kot je prikazano na sliki 4.2. Vsaka shema ima svoj mapo z več podmapami, ki predstavljajo bodisi element sheme bodisi podshemo. Na sliki 4.2 podshemo predstavlja $Naslov$. Mape, ki predstavljajo elemente, hranijo datoteke s podatki. Za ta način smo se odločili predvsem zaradi shranjevanja vmesnih



Slika 4.2: Prikaz podatkovne sheme Stranka kot hierarhična struktura.

rezultatov, ki jih generirajo komponente ogrodja. Shranjevanje vmesnih rezultatov je smiselno tudi zaradi dolgotrajnih procesov za izračun povzetkov, saj se tako izognemo ponovnemu generiranju povzetkov ob vsakem zagonu. Naloga te komponente je tudi pretvorba podatkov iz XML oblike v tekstovno datoteko, kjer vsaka vrstica predstavlja en zapis. Posamezni zapis predstavlja en niz brez presledkov, ki ga imenujemo žeton.

4.3 Predobdelava

Korak za predobdelavo skrbi za filtriranje podatkov, poenotenje formatov in odstranjevanje duplikatov. Naloga filtra je, da odstrani prazne in enoznakovne zapise oz. vrstice, hkrati pa odstrani vse podvojene zapise. Proces poenotenja formatov poskrbi za transformacijo zapisov v enoten format. Kot primer vzemimo datum, ki je lahko zapisan z različnimi formati: $dd\backslash mm\backslash llll$,

$mm - dd - llll$, $mm : dd : llll$ itd. Pretvorba v enoten format je pomembna zaradi boljše medsebojne primerljivosti podatkov. Poenotenje formatov ni nujno vedno potrebno, recimo v primeru, da za predstavitev podatkov v vektorskem prostoru uporabimo metodo, ki na format ni občutljiva. Predobdelava poskrbi tudi za pretvorbo tekstovnih podatkov v malo ali veliko pisavo, če tako zahteva predstavitev podatkov. Rezultat tega koraka je nova datoteka, ki vsebuje le enolične zapise.

4.4 Pretvorba podatkov

Proces pretvorbe podatkov je eden najpomembnejših procesov pri iskanju preslikav z metodo AA. Metoda arhetipske analize za iskanje povzetkov deluje nad točkami v prostoru. To pomeni, da moramo vse podatke, ne glede na podatkovni tip, pretvoriti v ustrezno obliko. V našem primeru je glede na naravo problema najustreznejša predstavitev s točkami v večdimenzionalnem vektorskem prostoru. Pretvorba podatkov mora ohraniti značilnosti podatkov v izvorni obliki. Če sta si dva zapisa v izvorni obliki med sabo zelo podobna, morata biti blizu skupaj tudi v vektorskem prostoru (t.j. razdalja med njima mora biti minimalna). Izhod komponente za pretvorbo podatkov je datoteka, kjer so podatki predstavljeni z vektorji, njihova dimenzija pa je odvisna od posamezne predstavitve podatkov.

4.5 Generiranje povzetkov

V fazi generiranja povzetkov za vsak element sheme izračunamo povzetek. Prejšnji korak je poskrbel za pretvorbo podatkov v vektorje, v tem koraku pa vse vektorje preberemo in jih zapišemo v matriko, ki služi kot vhod v algoritem za izračun povzetkov. Rezultat algoritma za izračun povzetkov je prav tako matrika, ki hrani arhetipe podatkov elementa. Izhodna matrika se ponovno zapiše v datoteko in je namenjena zadnji fazi procesa za iskanja preslikav, kjer se povzetek uporabi za izračun podobnosti z drugimi elementi.

Za izračun povzetkov uporabljamo algoritem arhetipske analize.

4.6 Iskanje preslikav

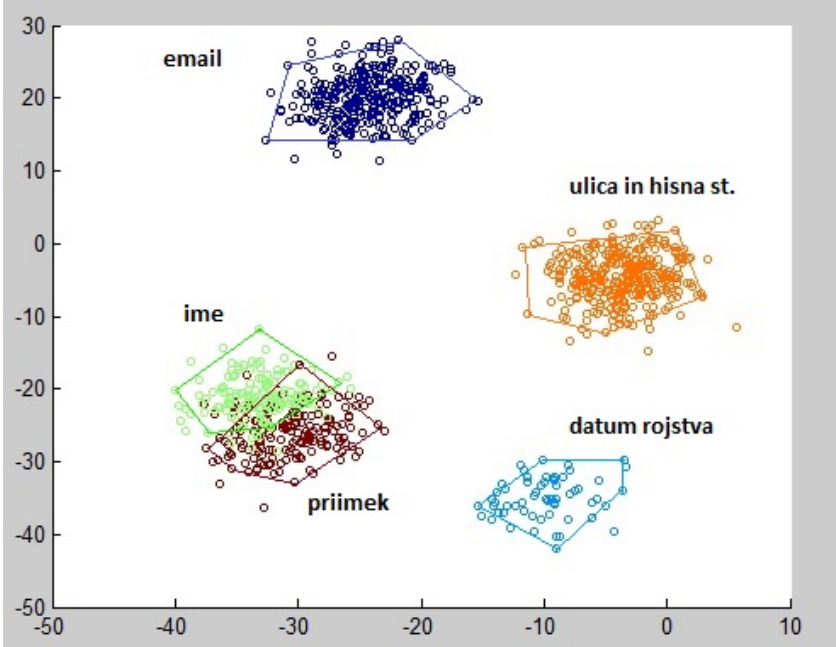
Ko za obe vhodni shemi pridobimo podatkovne povzetke za vse elemente, se lahko lotimo integriranja podatkovnih shem. Algoritem za iskanje preslikav deluje v dveh fazah. V prvi fazi izračuna podobnosti med vsemi elementi iz ene in druge sheme. Rezultat prve faze je seznam elementov druge sheme, ki predstavljajo potencialne kandidate za preslikavo za vsak element prve sheme. Naloga prve faze je, da odkrije enostavne preslikave številnosti 1:1. Druga faza poskrbi za iskanje kompleksnih preslikav, ki za vhod vzame rezultate prve faze. Za vse elemente, tudi za tiste, ki že imajo odkrito preslikavo, preveri, ali s kombinacijo z drugim elementom iste sheme obstaja preslikava z visokim ujemanjem. Ker je kombinacij za iskanje kompleksnih preslikav zelo veliko, bomo uporabili enostavno metodo, ki ni najbolj natančna, je pa zelo hitra. S tem bomo precej zmanjšali nabor elementov in število kombinacij, ki jih moramo preveriti. Uporabljene metode za izračun podobnosti povzetkov bomo podrobneje opisali v naslednjem poglavju.

Poglavje 5

Integracija shem na osnovi arhetipske analize

Ideja za našo metodo temelji na predpostavki, da pri integraciji v večini primerov aplikacije in sistemi hranijo podobne podatke, ki jih drugače strukturirajo. Za vsak element sheme želimo generirati povzetek podatkov, ki jih hrani. V našem primeru povzetek predstavlja približek konveksne ovojnice. Ideja temelji na tem, da podatki posameznega elementa v prostoru predstavljajo eno samo gručo. Pri metodah gručenja lahko gručo predstavimo bodisi s centroidom (točka, ki predstavlja težišče podatkov) bodisi s točko v gruči, za katero velja, da minimizira vsoto kvadratnih razdalj do vseh ostalih točk v gruči (*clustroid*). V našem primeru pa gručo (podatke elementa) predstavimo z množico arhetipov.

Trdimo lahko, da imajo elementi, ki hranijo podobne podatke, posledično tudi podobne povzetke. S primerjanjem povzetkov tako odkrijemo preslikave med elementi sheme. Slika 5.1 kot primer prikazuje elemente podatkovne sheme z gručami v dvodimenzionalnem prostoru. Vse sheme v prostoru predstavimo na isti način, tako da pride do prekrivanja med povzetki elementov, ki hranijo podobne podatke. Večje kot bo prekrivanje med povzetki, večja je verjetnost, da elementa hranita iste podatke. Pri tem nas čakajo izzivi, ki jih moramo rešiti. Odkriti moramo namreč način za transformacijo vseh ti-



Slika 5.1: Simbolični prikaz instančnih podatkov sheme za entiteto *Oseba* v dvodimenzionalnem prostoru, kjer vsaka gruča predstavlja podatke enega elementa. Za vsak element je prikazan povzetek s petimi arhetipi.

pov podatkov v vektorski prostor, saj prav na predstavitvi podatkov temelji uspešnost metode za odkrivanje preslikav na osnovi arhetipske analize. Prav tako pa je zelo pomemben izračun podobnosti med povzetki elementov. Z ustrezno predstavitvijo podatkov in primerno metriko za izračun podobnosti povzetkov smo se lotili implementacije algoritma za iskanje preslikav, ki ga podajamo v nadaljevanju. Začnemo z enostavnimi preslikavami številnosti 1:1, nato pa poskusimo z iskanjem kompleksnejših preslikav številnosti 1:N in N:1.

5.1 Psevdokoda algoritma za iskanje povzetkov z AA in implementacija

Pri implementaciji izhajamo iz postopka, predstavljenega v [11]. Omenili smo že, da gre v osnovi za optimizacijski algoritem z alternirajočim posodabljanjem matrik S in C, pri čemer iščemo rešitev, za katero velja, da je ocenjena napaka čim manjša. Algoritem 1 predstavlja glavno zanko algoritma, ki se izvaja, dokler je napaka večja od konvergenčnega kriterija, ki smo ga določili ročno, oziroma dokler ne doseže maksimalnega števila iteracij. Znotraj zanke se izvaja alternirajoča optimizacija matrik S (algoritem 2) in C (algoritem 3). Algoritma za posodobitev matrik S in C izvajata linearno iskanje za optimizacijo parametra μ_s oz. μ_c.

Algoritem 1: AA main loop

```
begin
    C ⟵ FurthestSum(X)
    S ⟵ Random()
    initialize XC, CtXtX, CtXtXC, SSt
    initialize const
    updateS(S, CtXtX, CtXtXC, μ_S, const, error, SSt, 25); initialize error
    repeat
        XSt = X * St
        updateC(C, XSt, SSt, X, XC, CtXtXC, μ_C, error, const, 10);
        CtXtX = (XC)t * X;
        updateS(S, CtXtX, CtXtXC, μ_S, const, error, SSt, 10);
    until iteration < maxIteration
    index = sort( ∑ S, descend)
    S = S (index, :)
    C = C(:, index)
    XC = XC(:, index)
```

5.1.1 Implementacija

Za implementacijo prototipnega ogrodja za integriranje shem smo uporabili razvojno okolje matlab. Matlab je okolje, ki omogoča enostavno in hitro implementacijo prototipov, ki temeljijo na matričnem računu. Glavni razlog

Algoritem 2: updateS

```
begin
    initialize gS
    for i < nIter do
        errorOld = error
        gS = (CtXtXC * S - CtXtX) / (const / c)
        gS = gS - e * ∑ (gS ∘ S)
        sOld = S
        while 1==1 do
            S = sOld - gS * μ_S
            S(S < 0) = 0
            SSt = S * St
            error = ∑∑ (X ∘ X) - 2 ⟨ CtXtX, S⟩ + ⟨ CtXtXC, SSt ⟩
            if error ≤ errorOld * (1 + 10^−9) then
                μ_S = μ_S * 1.2
                break
            else
                μ_S = μ_S / 2
```

Algoritem 3: updateC

```
begin
    initialize gC
    for i < nIter do
        errorOld = error
        T = XC * SSt; gC = (Xt * T - CtXSt) / const
        gC = gC - e * ∑ (gC ∘ C)
        cOld = C
        while 1==1 do
            C = cOld - μ_C * gC
            C(C < 0) = 0
            nC = ∑ C + eps
            C = C * sparce(1:c, 1:c, 1 ⊘ nC)
            XC = X * C
            CtXtXC = (XC)t * XC
            error = ∑∑ (X ∘ X) - 2 ⟨ XC, XSt⟩ + ⟨ CtXtXC, SSt ⟩
            if error ≤ errorOld * (1 + 10^−9) then
                μ_C = μ_C * 1.2
                break
            else
                μ_C = μ_C / 2
```

```
   488 |   2.4428e+05 |   3.0942e-05 |   4.1092e+01 |   2.8396e-01 |       0.1248
   489 |   2.4427e+05 |   2.0900e-05 |   2.4655e+01 |   2.1978e-01 |       0.1872
   490 |   2.4426e+05 |   2.5305e-05 |   2.9586e+01 |   6.8040e-01 |       0.1092
   491 |   2.4426e+05 |   2.3762e-05 |   3.5504e+01 |   5.2660e-01 |       0.1092
   492 |   2.4425e+05 |   3.2156e-05 |   4.2604e+01 |   4.0757e-01 |       0.2028
   493 |   2.4425e+05 |   6.1168e-06 |   5.1125e+01 |   3.1545e-01 |       0.1248
   494 |   2.4424e+05 |   5.6155e-05 |   6.1350e+01 |   4.8830e-01 |       0.1248
   495 |   2.4423e+05 |   3.1955e-05 |   1.8405e+01 |   1.8896e-01 |       0.1092
   496 |   2.4422e+05 |   2.1460e-05 |   2.2086e+01 |   2.9250e-01 |       0.1092
   497 |   2.4422e+05 |   2.0979e-05 |   2.6503e+01 |   4.5277e-01 |       0.1404
   498 |   2.4421e+05 |   2.5590e-05 |   3.1804e+01 |   1.7522e-01 |       0.1716
   499 |   2.4420e+05 |   2.5377e-05 |   3.8165e+01 |   1.0849e+00 |       0.1560
   500 |   2.4420e+05 |   3.1385e-05 |   4.5798e+01 |   4.1983e-01 |       0.1248
Total time:      58.2820
```

Slika 5.2: Izpis parametrov zadnjih 10 iteracij algoritma arhetipske analize pri iskanju 30 arhetipov na podatkovni matriki velikosti 1000x15, ki je bil implementiran v matlab. Zadnji stolpec predstavlja čas ene iteracije glavne zanke.

za izbiro tega okolja so matrične operacije, na podlagi katerih je zasnovana metoda za izračun podatkovnih povzetkov. V prvi različici razvoja prototipa smo celotno kodo implementirali v matlabu. Implementiran algoritem smo testirali na testnih podatkih, ki so obsegali 1000 zapisov imen in pri tem analizirali časovno zmogljivost. Podatke smo v točke transformirali tako, da smo nize predstavili z ASCII vrednostnimi znakov, dolžino pa za vse zapise poenotili na maksimalno dolžino zapisa v vzorcu, v konkretnem primeru na 15 dimenzij. Iz vzorca podatkov smo nato generirali 30 arhetipov, kjer smo algoritem omejili na 500 iteracij. Algoritem je za generiranje povzetkov potreboval 58 s. Slika 5.2 prikazuje zadnjih 10 iteracij izvajanja. Eden izmed naših izzivov je zato bil, ali lahko izboljšamo časovno učinkovitost iskanja arhetipov.

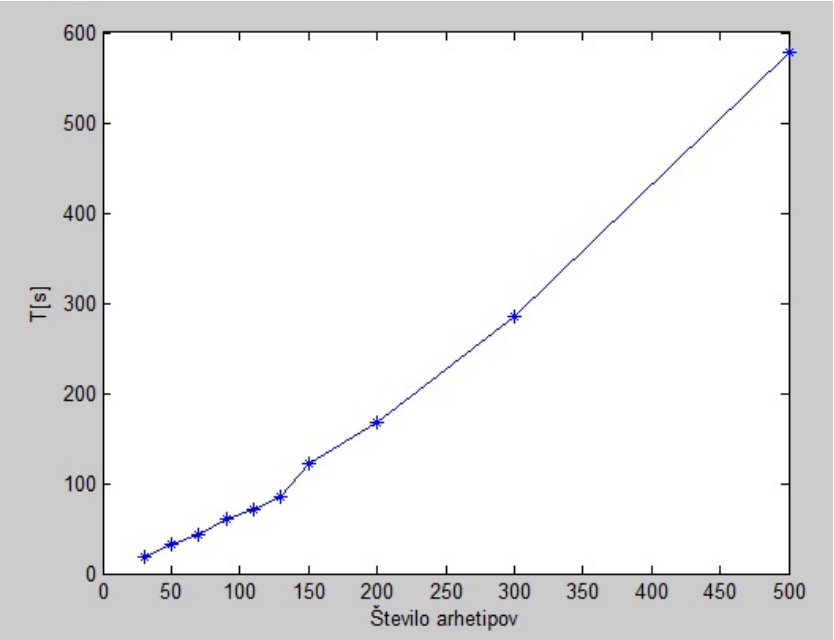
Optimizacijski algoritem za iskanje arhetipov je implementiran na način, predlagan v [11], kjer avtorji predlagajo najprimernejše pristope k optimalnemu reševanju obravnavane problematike. V nadaljnjo optimizacijo izvajalnega časa algoritma se v tej magistrski nalogi ne bomo spuščali, saj to ni predmet naše raziskave. Kljub temu pa obstaja še ena možnost, da zmanjšamo čas iskanja arhetipov. Iz preteklih izkušenj pri uporabi okolja matlab vemo,

da najbolj časovno potraten del kode predstavljajo zanke, saj matlab sodi med interpretirne programske jezike. Znano je, da je matlab optimiziran za izvajanje matričnih in vektorskih operacij, ki med drugim omogoča tudi vključevanje programskih jezikov, kot je *C* ali *C++*. Kadar želimo doseči visoke performance, je priporočljivo težke dele programske kode implementirati v nižjenivojskih programskih jezikih. Odločili smo se za *C++* in knjižnico *armadillo*, ki predstavlja *C++* knjižnico za linearno algebro [16]. Knjižnica *armadillo* podpira zelo podobno sintakso za matrične operacije kot matlab. Glavni razlog, da smo se odločili za to knjižnico, je možnost uporabe visoko performančnih matematičnih knjižnic za optimalno izrabo procesnih virov, ki jih uporablja tudi sam matlab. Uporabili smo knjižnico *Intel MKL* (Math Kernel Library), ki jo je enostavno integrirati s knjižnico *armadillo*. Algoritem za iskanje arhetipov smo tako implementirali v *C++* in ga integrirali v ogrodje, razvitem v okolju matlab z uporabo API vmesnika MEX, ki omogoča klice funkcij drugih programskih jezikov, v našem primeru *C++*.

Vnovično testiranje je pokazalo 50 do 60% pohitritev v primerjavi z neposredno implementacijo v matlabu. Performančno izboljšanje je natančneje razdelano na sliki 5.3, pri čemer smo testiranje izvedli na prenosniku s procesorjem Intel Core i7, 8 jeder, 1,6GHz in 8GB pomnilnika. Dosegli smo precejšnjo pohitritev, pri čemer moramo poudariti, da ta rezultat ne odraža celotne slike, saj hitrost izračuna povzetkov pride do izraza šele pri iskanju preslikav med podatkovnimi shemami. Kljub temu, da se v tem delu posvečamo natančnosti in ne hitrosti pri iskanju preslikav, smo se za tak korak odločili predvsem zaradi hitre in enostavne implementacije, predvsem lahko ob hitrejšem izvajanju zato več časa namenimo samemu testiranju predlagane metode. Testirali smo tudi časovno odvisnost algoritma glede na število arhetipov v povzetku. Izmerjena časovna odvisnost je prikazana na grafu 5.4.

488	2.0158e+05	1.7474e+01	8.6683e-05	3.9015e+02	1.4198e-01	0.0390
489	2.0157e+05	1.5298e+01	7.5893e-05	6.0393e+02	2.1978e-01	0.0380
490	2.0155e+05	1.8633e+01	9.2447e-05	4.6742e+02	3.4020e-01	0.0390
491	2.0153e+05	2.0666e+01	1.0255e-04	3.6177e+02	5.2660e-01	0.0390
492	2.0151e+05	1.8314e+01	9.0884e-05	2.8000e+02	2.0379e-01	0.0380
493	2.0149e+05	1.8647e+01	9.2544e-05	2.1671e+02	1.5772e-01	0.0390
494	2.0147e+05	1.8783e+01	9.3230e-05	3.3545e+02	1.2207e-01	0.0380
495	2.0145e+05	2.0226e+01	1.0040e-04	5.1926e+02	1.8896e-01	0.0360
496	2.0143e+05	2.0549e+01	1.0202e-04	4.0189e+02	2.9250e-01	0.0390
497	2.0141e+05	2.0053e+01	9.9560e-05	3.1105e+02	1.1319e-01	0.0410
498	2.0139e+05	2.2212e+01	1.1029e-04	4.8148e+02	1.7522e-01	0.0380
499	2.0137e+05	1.9931e+01	9.8977e-05	3.7265e+02	1.3561e-01	0.0420
500	2.0135e+05	2.2105e+01	1.0979e-04	2.8842e+02	2.0992e-01	0.0460
Total time:	20.1700					

Slika 5.3: Zadnjih 10 iteracij algoritma, implementiranega v C++. Test je bil izveden nad istimi podatki kot implementacija v matlabu. Desni stolpec prikazuje čas ene iteracije glavne zanke.



Slika 5.4: Čas iskanja povzetka v odvisnosti od števila arhetipov v povzetku pri 977 zapisih v vektorjih z dimenzijo 15.

5.2 Metode za predstavitev podatkov v vektorskem prostoru

Metoda arhetipske analize je namenjena iskanju približka konveksne ovojnice točk v prostoru. Če želimo to metodo uporabiti za iskanje povzetkov posameznih elementov sheme, moramo podatke pretvoriti v vektorsko obliko. Pri pretvorbi moramo paziti, da ohranjamo podobnost med posameznimi zapisi, npr. zapisa "Janez" in "Jan" ali "22-8-2013" in "21-9-2013". Hkrati mora predstavitev podatkov v vektorskem prostoru omogočati primerjavo podatkov z različnimi podatkovnimi tipi. Razlog za to je obstoj podatkovnih tipov, ki lahko hranijo tudi podatke, za katere v osnovi niso namenjeni. Kot primer vzemimo podatkovni tip *String*, ki lahko hrani datum, celo število, decimalno število itd. Zaradi tega vse podatke interpretiramo kot tekstovne. Hkrati želimo podatke predstaviti čim bolj učinkovito. Ker govorimo o točkah v večdimenzionalnem vektorskem prostoru, to pomeni uporabo čim manjših dimenzij, saj bomo s tem časovno optimizirali tudi iskanje konveksnih ovojnic. Paziti pa moramo, da s tem ne bi zmanjšali natančnosti pri iskanju preslikav med podatkovnimi shemami.

5.2.1 Pretvorba na podlagi ASCII vrednosti

Prva ideja za pretvorbo podatkov v večdimenzionalni vektorski prostor je precej enostavna. Ker vsak zapis interpretiramo kot tekstovni podatek, nam to omogoča, da ga pretvorimo v vektorje, kjer so znaki predstavljeni številčno z ASCII vrednostjo. Primer predstavitve zapisa, vrednost **"Janez"**, se pretvori v vektor **[74, 97, 110, 101, 122]**. Ker imajo zapisi različne dolžine, moramo dimenzije vseh vektorjev poenotiti. To storimo tako, da za dimenzijo določimo maksimalno dolžino zapisa v množici podatkov elementa sheme. Zapise, pri katerih se dolžina ne ujema z maksimalno, pretvorimo v vektorsko obliko, tako da jim na koncu dodamo ničle. Slika 5.5 prikazuje izsek datoteke s podatki v vektorski obliki.

Dobra lastnost te predstavitve podatkov je, da ohranja podobnost podat-

```
370  106,97,110,105,99,101,0,0,0,0,0,0,0,0,0,0
371  106,97,110,111,115,0,0,0,0,0,0,0,0,0,0,0
372  106,97,114,105,114,0,0,0,0,0,0,0,0,0,0,0
373  106,97,115,111,110,0,0,0,0,0,0,0,0,0,0,0
374  106,97,115,112,97,108,0,0,0,0,0,0,0,0,0,0
375  106,97,115,119,105,110,100,101,114,0,0,0,0,0,0
376  106,97,118,105,101,114,0,0,0,0,0,0,0,0,0,0
377  106,97,121,0,0,0,0,0,0,0,0,0,0,0,0,0
378  106,97,121,97,110,116,0,0,0,0,0,0,0,0,0,0
379  106,97,121,97,118,101,108,0,0,0,0,0,0,0,0,0
380  106,101,97,110,45,99,108,97,117,100,101,0,0,0,0
381  106,101,97,110,45,109,97,114,105,101,0,0,0,0,0
382  106,101,97,110,45,112,105,101,114,114,101,0,0,0,0
383  106,101,102,102,0,0,0,0,0,0,0,0,0,0,0,0
384  106,101,102,102,114,101,121,0,0,0,0,0,0,0,0,0
385  106,101,104,97,110,45,102,114,97,110,99,111,105,115,0
386  106,101,110,110,105,102,101,114,0,0,0,0,0,0,0
387  106,101,110,113,45,110,101,110,103,0,0,0,0,0,0
```

Slika 5.5: Izsek podatkovnih zapisov v vektorsi obliki, ki so bili pretvorjeni na osnovi ASCII vrednosti znakov.

kov in omogoča primerjav med vsemi podatkovnimi tipi. Po drugi strani, ima slabo lastnost, ki se kaže pri generiranju povzetkov elementov sheme, kjer moramo zadostiti dvema pogojema: število arhetipov v povzetku mora biti za vse elemente enako, dimenzije vektorjev morajo biti enake za vse elemente obeh shem. Število arhetipov lahko enostavno poenotimo, za poenotenje dimenzij moramo pregledati podatke obeh shem in najti najdaljši zapis. V primeru velike količine podatkov to ni optimalen način, saj moramo skozi podatke iterirati dvakrat, najprej za iskanje najdaljšega zapisa, nato še za pretvorbo podatkov v vektorje. Metodo smo zato uporabili za testiranje algoritma za iskanje arhetipov, ker je enostavna za implementacijo in interpretacijo rezultatov. Večkratni iteraciji skozi podatke si želimo izogniti, zato poskusimo najti boljšo predstavitev.

5.2.2 Pretvorba podatkov na osnovi lokacij znakov in uporabe razpršilnih funkcij

Ideja te predstavitve je, da vektor definiramo kot abecedo, kjer vsak indeks vektorja predstavlja en element abecede. Ponovno se ukvarjamo s tekstovnimi podatki, zato v osnovi za abecedo lahko vzamemo kar kodno tabelo ASCII, vendar smo se odločili, da bo v prvi fazi dovolj, če pokrijemo 40 znakov: male črke angleške abecede, številke in znake {'@' ',' '.' ':'}. S tem dosežemo, da so vsi podatki predstavljeni z vektorji enake dimenzije. V tem primeru podatke v vektorsko obliko pretvorimo tako, da za vsak znak shranimo lokacije, na katerih se pojavi v zapisu. Kot primer vzemimo telefonsko številko: *01-33-67-333*, abeceda naj obsega številke in znak '-'. To pomeni, da ima prostor enajst dimenzij (na indeksih 0 - 9 imejmo znake števil, na indeksu 10 znak '-'). Zapis se tedaj pretvori v [**0:[0], 1:[1], 2:[], 3:[3,4,9,10,11], 4:[],5:[], 6:[6], 7:[7], 8:[], 9:[], -:[2,5,8]**]. Zapis v taki obliki ni ustrezen, saj s tem dobimo dvodimenzionalno tabelo, zato tabele posameznih znakov, ki hranijo indekse, na katerih se pojavijo znaki v zapisu, pretvorimo še v številčne vrednosti. Enega izmed načinov za pretvorbo v številčne vrednosti predstavljajo razpršilne funkcije (*hash*). V našem primeru smo za pretvorbo uporabili funkcijo $h(z) = \mathrm{mod}(\mathrm{h}(z-1) \cdot 65599 + z, 2^{32} - 1)$. Končna oblika zgornjega zapisa je tako enaka [**177621, 177622, 0, 3.2318e+09, 0, 0, 177627, 177628, 0, 0, 195088781**].

Ta način zagotavlja enotnost dimenzij pri vseh zapisih, težava se pojavi pri ohranjanje podobnosti. Če na začetek telefonske številke iz prejšnjega primera dodamo 0 (*001-33-67-333*), dobimo vektor [**193430770, 177623, 0, 3.5013e+09, 0, 0, 177628, 177629, 0, 0, 196275792**]. Izvirna zapisa sta si zelo podobna, razlikujeta se namreč le v enem znaku. Pretvorba v vektorsko obliko to podobnost pokvari, saj če primerjamo vektorja, takoj opazimo, da sta si precej različna. Težava te predstavitve je v razpršilni funkciji, ki podatke preveč razprši. Zato taka predstavitev v našem kontekstu ni ustrezna.

...	1	2	3	...
4	0	0	0	0
5	0	0	1	0
6	0	0	0	0
...	0	0	0	...

Slika 5.6: Izsek iz matrike sosednosti za zapis 53, vse ostale celice v matriki imajo vrednost 0.

5.2.3 Pretvorba podatkov na podlagi sosednosti

Tretji način za pretvorbo podatkov je osnovan na sosednosti znakov v zapisu. Temelji na ideji, da zgradimo kvadratno matriko, kjer celice predstavljajo frekvence ponovitev zaporedja dveh znakov. Znak na indeksu i (stolpec) sledi znaku na indeksu j (vrstica). Vektorsko obliko dobimo tako, da matriko preprosto vektoriziramo (v matlab to storimo s funkcijo *reshape*). V tej predstavitvi smo uporabili isto abecedo s 40-imi znaki, kot v podpoglavju 5.2.2. Prednost te predstavitve v primerjavi s prejšnjo je ta, da ohranja podobnost in sosednost znakov. Če se v obeh zapisih pojavi zaporedje "ab", bosta vektorja v isti dimenziji zelo podobna, kar je odvisno od frekvence pojavitev. Predstavitev izpolnjuje tudi pogoj o poenoteni dimenziji za vse podatke.

Predstavitev smo testirali na majhnem vzorcu podatkov, in sicer na številskih vrednostih: **[11, 17, 53, 02, 82, 97, 31, 29, 48, 63]**. Če pogledamo odsek matrike sosednosti za zapisa 53, dobimo matriko, ki je prikazana na sliki 5.6. Nad temi podatki smo izračunali povzetek s petimi arhetipi. Če podatke iz vektorske oblike pretvorimo nazaj v številčne vrednosti, v konveksni ovojnici dobimo naslednje vrednosti: **[82, 25, 21, 61, 13]**. To so vrednosti, ki v naši predstavitvi predstavljajo točke, ki resnično najbolje povzemajo zgornjo množico.

Testiranje na večji količini podatkov je pokazalo, da tudi ta predstavitev ni najboljša. Prva težava je število dimenzij, ki je enako 1600 v primeru uporabe abecede s 40-imi znaki, zato je iskanje arhetipov zaradi zasnove al-

goritma zelo počasno. Druga težava je gostota zapisov in vrednosti. Matrika, ki jo pridobimo s to predstavitvijo, je redka (*sparse matrix*), poleg tega so vrednosti, ki so različne od 0, zelo majhne. Pri velikih dimenzijah in tako majhnih vrednostih je lahko iskanje arhetipov zelo neučinkovito.

5.2.4 Pretvorba podatkov z večdimenzionalnim skaliranjem

Naslednji način predstavitve podatkov temelji na razdaljah med podatkovnimi zapisi. Ideja je, da med vsemi podatkovnimi zapisi ene sheme izračunamo vse medsebojne razdalje med posameznimi zapisi in sestavimo kvadratno matriko razdalj. Za izračun razdalj med zapisi smo uporabili Levenshteinovo razdaljo [26], ki predstavlja standard računanja razdalj med nizi. Matriko razdalj smo sestavili tako, da stolpec in vrstica predstavljata posamezni zapis. Matrika ima dimenzijo enako številu vseh podatkov. Drugi korak pretvorbe zajema postopek večdimenzionalnega skaliranja (*Multidimensional scaling* - MDS [18]), ki na podlagi razdalj med posameznimi zapisi generira vektorje. Le-ti predstavljajo pozicije posameznih zapisov v n-dimenzionalnem vektorskem prostoru, kjer je dimenzija enaka številu vseh zapisov. Algoritem MDS bo poskrbel za to, da bo vsak objekt (zapis) postavil v n-dimenzionalni prostor tako, da v največji meri ohrani podobnost (razdaljo) med objekti. Rezultat algoritma je matrika razdalj, kjer stolpci predstavljajo posamezni zapis. Da iz matrike pridobimo podatke enega elementa, si ob gradnji matrike razdalj zapomnimo intervale indeksov, kjer se nahajajo podatki posameznega elementa sheme. Na koncu za vse elemente izračunamo povzetke.

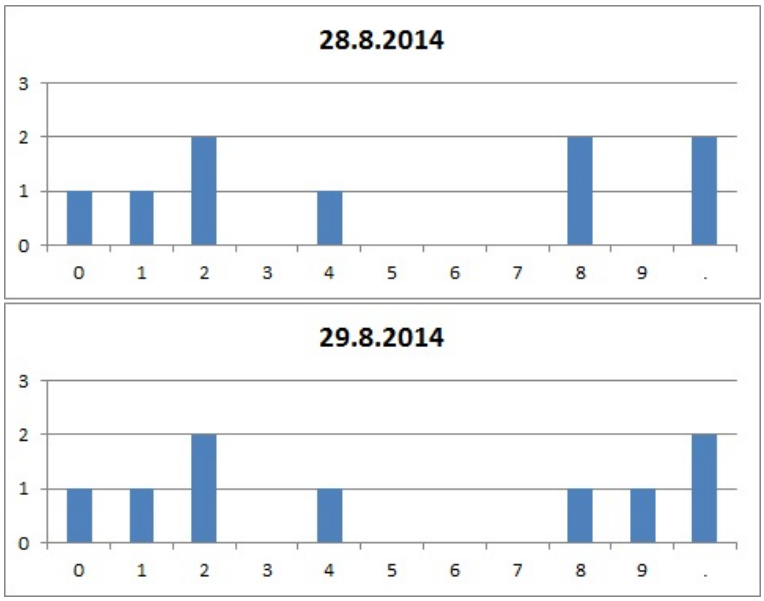
Osnovna ideja predstavitve je zelo dobra, saj metoda MDS zagotovi, da se podobnost podatkov ohranja. Tudi za to metodo se je izkazalo, da je v takšni obliki ne moremo uporabiti za namen iskanja preslikav. Predlagan postopek izvedemo na obeh shemah, za kateri iščemo preslikave. Problem se pojavi, ker lahko metodo uporabimo le v primeru, če je zaporedje podatkov v matrikah razdalj obeh shem enako. To pomeni, da se morajo podatki elementov, ki predstavljajo iste podatke, nahajati v obeh matrikah na istem

mestu. Poleg tega se mora ujemati tudi število podatkov v elementih shem, ki hranijo iste podatke in s tem isto število podatkov v obeh shemah. Pri taki predstavitvi je namreč dimenzija prostora določena s številom podatkov. Poleg tega je izračun podobnosti med zapisi časovno potraten proces, vsak zapis moramo primerjati z vsemi drugimi. Če imamo shemo z 10 elementi, kjer ima vsak element 1000 zapisov, moramo za izračun razdalj med zapisi izvesti 10000^2 operacij. Predstavitve podatkov v taki obliki zato ne moremo uporabiti za želen namen.

5.2.5 Pretvorba podatkov v histograme

Vzemimo abecedo, ki vsebuje 40 znakov, kot smo jo predstavili v podpoglavju 5.2.2, in za vsak zapis generirajmo histogram, ki ga pretvorimo v vektor. Predstavitev podatkov na ta način ima v primerjavi z ostalimi relativno malo število dimenzij (40) in ohranja podobnost med podatki. Slaba lastnost te predstavitve je neupoštevanje zaporedja znakov, zato lahko za dve popolnoma različni besedi, ki vsebujeta iste znaka, dobimo enak histogram. Kot primer vzemimo besedno zvezo "Tom Marvolo Riddle" in anagram "I am Lord Voldemort", ki bosta na ta način predstavljena z istim histogramom. Iz vidika hostogramov sta to enaka zapisa, vendar nas to ne moti, saj je verjetnost, da bo nek zapis anagram drugega, v praksi zelo majhna. Pomembno je, da predstavitev ohranja podobnost tudi v primerih, ko imamo majhne spremembe v zapisih kot npr. pri datumih "28.8.2014" in "29.8.2014", katerih histograma sta prikazana na sliki 5.7.

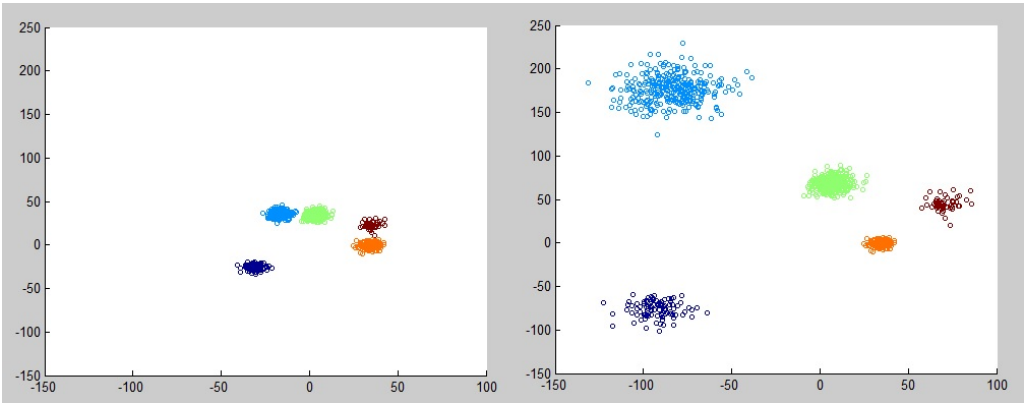
Predstavitev podatkov s histogrami izpolnjuje vse zahtevane pogoje, dimenzije so poenotene za vse podatke in pri pretvorbi se ohranja podobnost. Izračun povzetkov podatkovnih množic je v primerjavi s prejšnjima dvema metodama precej hitrejši. Pri prejšnji metodi predstavitev zaradi omenjenih omejitev nismo mogli uporabiti za iskanje preslikav med dvema shemama, v tem primeru teh omejitev ni. Testiranje metode na majhni količini podatkov je pokazalo, da metoda zelo dobro loči podatke različnih podatkovnih tipov, nekoliko slabše se obnese pri podatkih istega tipa, saj privede do nekoliko



Slika 5.7: Primera histogramov za zapisa “28.8.2014” in “29.8.2014”, podobnost histogramov je očitna.

večjega prekrivanja med povzetki. Kljub temu to ne predstavlja kritične slabosti, saj je prekrivanje v veliki meri odvisno tudi od samih podatkov, vseeno skušamo to zaobiti. Vprašali smo se, ali obstaja značilka, ki jo lahko pridobimo iz vseh podatkov, in ali je za vsak element sheme v veliki verjetnosti različna. Če takšne značilke obstajajo, jih lahko uporabimo tako, da povzetke podatkov, preden jih primerjamo med sabo, čim bolj medsebojno ločimo. S tem načinom bomo za elemente sheme pridobili gruče z manjšim medsebojnim prekrivanjem. Ena značilka, ki ustreza opisu, je povprečna dolžina podatkov elementa. Pred primerjavo povzetkov tako vektorje pomnožimo s povprečno dolžino. Skalarni produkt matrike in skalarja povzroči večjo razpršenost točk (velikost gruč), kot je razvidno na sliki 5.8. Če smo pozorni na skalo grafa, vidmo, da so razmiki med gručami večji kot v originalu.

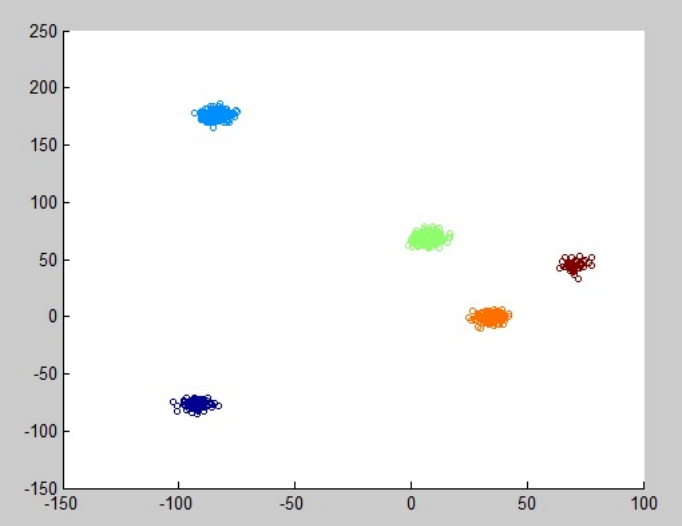
Izkaže se, da je ta pristop učinkovit, vendar ima eno slabost. Če vse vektorje pomnožimo s skalarjem, se transformira velikost gruče, kar lahko opa-



Slika 5.8: Izvorna razporeditev gruč na levi strani in modificirana predstavitev na desni strani s skaliranjem vektorjev.

zimo na sliki 5.8 pri svetlomodri gruči. Ta transformacija vpliva na same povzetke podatkov, zato ni najboljša. Predlagamo naslednjo spremembo. Namesto da pomnožimo celotno matriko, bomo za vsako gručo poiskali težiščno točko, jo pomnožili s povprečno dolžino podatkov in nazadnje razliko med originalno težiščno točko in novo točko prišteli vsem točkam v gruči. Tako bomo gruči v celoti le zamaknili (translacija), velikost bo ostala enaka. Za gruče smo uporabili naslednje povprečne dolžine elementov: [3, 5, 2, 1, 2]. Rezultat tega načina je predstavljen na sliki 5.9. V primerjavi z originalno predstavitvijo na sliki 5.8 (levo) je jasno razvidno, da so gruče med sabo bolj ločene.

Takšna predstavitev je na testnih podatkih vrnila najboljše rezultate v primerjavi z vsemi naštetimi metodami za predstavitev podatkov. V nadaljevanju si bomo pogledali, kako se metoda obnese pri iskanju preslikav med shemami. Še pred tem bomo predstavili pristope, ki jih uporabljamo za izračun podobnosti med podatkovnimi povzetki - približki konveksnih ovojnic. Izračun metrike podobnosti mora biti hiter in natančen, saj je od nje odvisno, kako dobro bo deloval algoritem za integriranje shem.



Slika 5.9: Predstavitev podatkov z upoštevanjem povprečne dolžine podatkov posameznega elementa.

5.3 Metrike za izračun podobnosti povzetkov

Pri iskanju preslikav med podatkovnimi shemami iščemo korespondence med elementi. Naša metoda za vsak element obeh shem izračuna povzetke podatkov, ki so predstavljeni s približki konveksnih ovojnic. Elementa dveh shem hranita podobne podatke, če med dvema konveksnima ovojnicama obstaja prekrivanje. Bolj kot se povzetka prekrivata, večja je podobnost. Zato uvedemo metriko, ki bo povedala, koliko sta si opazovana povzetka podobna. Podobnost med povzetki ocenjujemo na intervalu $\mathbb{R} = [0, 1]$, kjer 0 predstavlja nično podobnost in 1 popolno podobnost med povzetkoma. Želimo, da bo izračun podobnosti dveh povzetkov kar se da hiter in natančen. Iz vidika natančnosti je najprimernejša metrika za podobnost prekrivanja med dvema konveksnima ovojnicama. V dvodimenzionalnem prostoru lahko to zelo enostavno vizualno predstavimo. Če imamo dva konveksna lika, za katera moramo izračunati delež prekrivanja, to enostavno opišemo kot geometrijski problem. Za izračun prekrivanja dveh likov v dvodimenzionalnem prostoru lahko tedaj uporabimo algoritem Sutherland-Hodgman [20].

V dvodimenzionalnem prostoru lahko podobnost med povzetkoma izračunamo tudi z razmerjem med ploščino v preseku in skupno ploščino dveh povzetkov. Drugi način izračuna podobnosti je lahko osnovan na osnovi razdalj, npr. z evklidsko razdaljo [23] med centralnima točkama povzetkov. Evklidska razdalja sama po sebi ne predstavlja podobnosti, lahko jo kombiniramo z varianco obeh povzetkov, npr. če je razdalja večja od vsote obeh varianc, je podobnost 0, če je ta manjša, lahko izračunamo razmerje med razdaljo in vsoto varianc ter ga nato odštejemo od vrednosti 1, s čimer dobimo podobnost. Boljši primer na osnovi razdalje za našo situacijo predstavlja kosinusna podobnost, ki jo opišemo v podpoglavju 5.3.1.

Izračun prekrivanja konveksnih ovojnic z višjimi dimenzijami, npr. 40, si težko vizualno predstavljamo, dodatno težavo predstavlja kompleksnost natančnega izračuna prekrivanja, zato poskusimo najti metodo, ki je hitra, vendar na račun manjše natančnosti. V nadaljevanju predstavimo tri metode, s katerimi lahko izračunamo podobnost konveksnih ovojnic in jih uporabimo pri algoritmu za iskanje preslikav.

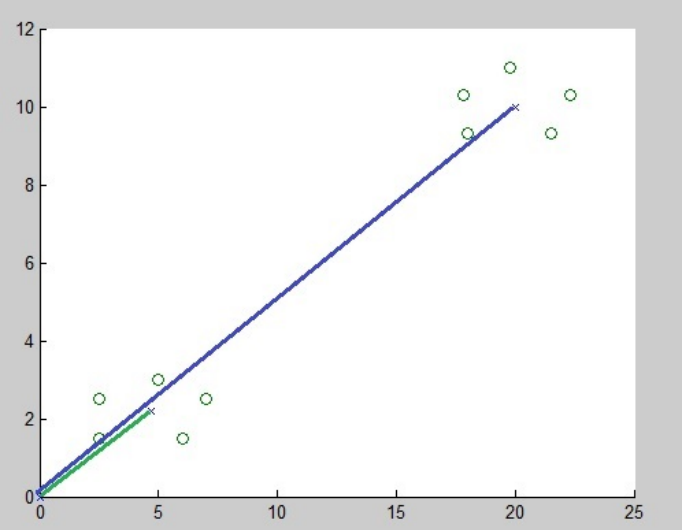
5.3.1 Kosinusna podobnost

Eden najhitrejših načinov za izračun podobnosti med konveksnimi ovojnicami je kosinusna podobnost. V našem primeru smo kosinusno podobnost računali tako, da smo vse konveksne ovojnice predstavili s povprečno sredinsko točko (centroid), s čimer smo pridobili vektorje, med katerimi enostavno izračunamo kosinusno podobnost po enačbi:

$$cos(\theta) = \frac{a \cdot b}{\|a\|\|b\|}. \tag{5.1}$$

Prednost tega pristopa je hitrost in enostavnost. Njena slabost je, da ne upošteva oblike samega povzetka podatkov. Metoda ima težave tudi v primerih, ko je kot med povzetkoma majhen, razdalja pa velika, kot prikazuje primer na sliki 5.10.

Kljub temu je metodo smiselno uporabili pri algoritmu za integriranje shem zaradi njene hitrosti. Čeprav ni nujno, da takoj najdemo pravo rešitev,



Slika 5.10: Problem pri računanju podobnosti s kosinusno metodo za naš primer. Ker metoda deluje na osnovi kotov med vektorji, v podobnih primerih, kot je prikazan na sliki, dobimo visoko podobnost, čeprav prekrivanja ni.

jo lahko uporabimo za zmanjšanje nabora potrebnih primerjav oz. za izračun začetnega približka pri iskanju preslikav za nek element sheme.

5.3.2 Izračun podobnosti povzetkov na osnovi Jaccardove mere podobnosti

Da bi pridobili natančnejšo metriko podobnosti, smo poskusili z iskanjem prekrivanj med konveksnimi ovojnicami na podlagi skupnih točk. Jaccardova mera podobnosti se uporablja za izračun podobnosti med podatkovnimi množicami. Njen izračun je preprost, gre za kvocient števila elementov v preseku z unijo dveh množic:

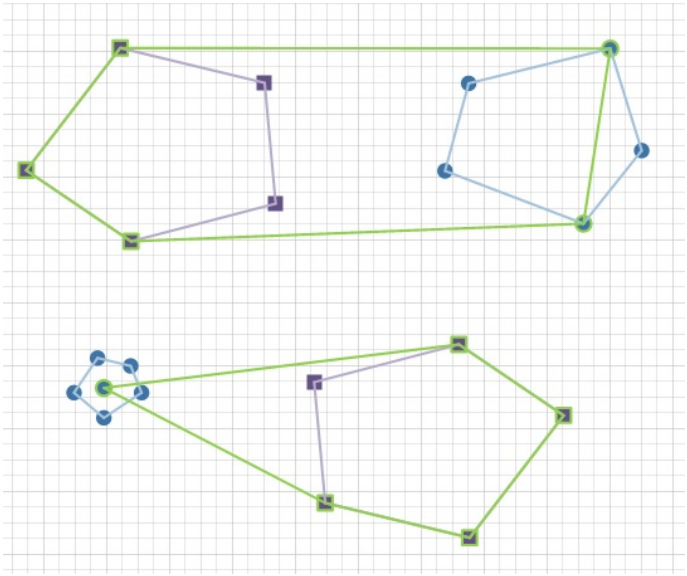
$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}. \tag{5.2}$$

Za izračun števila točk v preseku predlagamo naslednjo metodo, ki temelji na Jaccardovi razdalji. Imejmo dve konveksni ovojnici A in B, ki se

delno prekrivata. Vzemimo točke konveksne ovojnice A. V naslednjem koraku združimo točke konveksnih ovojnic A in B v eno množico in za novo množico določimo konveksno ovojnico z arhetipsko analizo, pri čemer dobljen povzetek označimo z AB. Če sta si konveksni ovojnici podobni in obstajajo točke v preseku, potem bo razlika med povzetkoma A in AB majhna, saj v tem primeru večina točk v konveksni ovojnici ostane enaka. V primeru, da se konveksna ovojnica B v celoti nahaja v A, potem je povzetek $A = AB$. Če sta povzetka med sabo različna in se ne prekrivata, potem bo skupni povzetek zajemal točke obeh množic, ki bo vseboval približno polovico točk iz A in polovico točk iz povzetka B.

Opisane lastnosti lahko uporabimo za izračun podobnosti z Jaccardovo enačbo. Število skupnih točk izračunamo tako, da primerjamo povzetek A in skupni povzetek AB dveh konveksnih ovojnic. Tiste točke, ki se iz povzetka A ne pojavijo v skupnem povzetku AB, so v preseku konveksnih ovojnic. Ker vemo, koliko je točk v obeh povzetkih in v preseku, lahko izračunamo število točk v uniji, prav tako pa lahko izračunamo podobnost s pomočjo enačbe (5.2). Tak pristop za vsak par elementov sheme, ki jih preverjamo, zahteva iskanje konveksne ovojnice in zato v primerjavi s kosinusno podobnostjo zahteva več časa za izračun. Spomnimo se že omenjene lastnosti algoritma arhetipske analize, ki pravi, da so arhetipi linearna kombinacija ostalih točk. Ko računamo skupni povzetek, ni nujno, da za arhetipe dobimo iste točke, saj algoritem arhetipe predstavi z linearno kombinacijo točk v množici. To oteži identificiranje točk, ki se ne spremenijo ob združitvi. Problem smo rešili z izračunom evklidske razdalje med novimi in starimi arhetipi. Če je razdalja dovolj majhna, arhetip identificiramo kot stari arhetip.

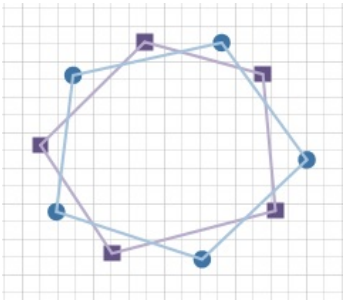
Problem predlagane metrike za izračun predstavljajo primeri, ko se dve konveksni ovojnici ne prekrivata. V teh primerih bo izračunana podobnost okoli 0,5, seveda v primeru, da je velikost konveksnih ovojnic primerljiva. Če je ena konveksna ovojnica znatno manjša od druge, se lahko zgodi, da bo v skupnem povzetku predstavljena z majhnim številom točk. Za primer vzemimo zelo majhno in zelo veliko konveksno ovojnico. Algoritem bo tedaj



Slika 5.11: Problem pristopa z združevanjem povzetkov.

manjšo konveksno ovojnico predstavil z manjšim številom arhetipov, zato ker so razdalje med točkami v tem primeru manjše in predstavljajo manjšo napako. V nasprotnem primeru so, pri predstavitvi večje konveksne ovojnice z manjšim številom arhetipov, razlike med točkami večje in je s tem posledično večja tudi napaka. Rezultat bo velika podobnost, čeprav temu v resnici ni tako, kar je razvidno iz slike 5.11. Zgornji primer na sliki prikazuje problem, ko ni prekrivanja. Zelena obroba označuje točke v povzetku združenih množic. V tem primeru izračun podobnosti vrne okoli 0,5. Spodnji primer prikazuje problem razlike v velikosti konveksnih ovojnic, ki kot rezultat vrne visoko podobnost, čeprav podobnosti med ovojnicama sploh ni. Če povzamemo ta pristop, lahko rečemo, da je učinkovit le v primeru, ko želimo ugotoviti ali je neka konveksna ovojnica znotraj druge, kar nam pri izračunu podobnosti ne pomaga.

Za pravilno delovanje metrike smo pristop za iskanje števila točk v preseku nekoliko modificirali. Namesto, da iščemo skupni povzetek, raje preprosto preverjamo, ali se točka druge konveksne ovojnice nahaja znotraj prve. Če



Slika 5.12: Primer, ko predlagana metrika, na osnovi Jaccardove podobnosti, ni primerna za izračun podobnosti.

se, potem je točka v preseku. Ali se točka nahaja znotraj konveksne ovojnice, ugotovimo tako, da v prvo množico dodamo točko in izračunamo nov povzetek. Če se povzetek ne spremeni, je točka znotraj konveksne ovojnice. Postopek moramo izvesti za testiranje vsebnosti točk prve ovojnice v drugi in obratno. Ta pristop za vsako točko zahteva iskanje konveksne ovojnice z arhetipsko analizo, zato z vidika časovne kompleksnosti ni najboljša. Z vidika izračuna podobnosti nam ravno to ustreza, razen v redkih primerih, npr. na tipičnem primeru, prikazanem na sliki 5.12. Kljub temu da se lika praktično prekrivata, ocena ujemanja ne bo pravilna. To poskušamo izničiti tako, da za točko, ki jo testiramo, preverimo, če je njena razdalja do najbližje točke v konveksni ovojnici manjša od razdalje, določene z vnaprej določenim deležem δ minimalne razdalje izmed vseh točk na konveksni ovojnici do središča. Če to drži, točko pridružimo množici v preseku. V naših poskusih se je vrednost $\delta = 0,5$ izkazala za najboljšo.

5.4 Algoritem za integriranje podatkovnih shem

Jedro naše raziskave predstavlja iskanje preslikav med elementi podatkovnih shem. Definirali smo način za predstavitev podatkov v vektorskem prostoru, tako da iz njih lahko izračunamo povzetke z arhetipsko analizo. Arhetipska analiza podatkov nam vrne reducirano število točk, ki predstavljajo povzetek

podatkov. Definirali smo tudi metrike, s katerimi medsebojno primerjamo povzetke posameznih elementov. Naslednji korak problema predstavlja iskanje elementov, ki hranijo isto podatkovno vsebino.

V tem poglavju se najprej lotimo enostavnih preslikav številnosti 1:1. Algoritem mora znati poiskati preslikave tudi med elementi na različnih nivojih vhodnih shem, ne glede na številnost preslikav. Predlagamo tudi postopek za iskanje kompleksnih preslikav.

5.4.1 Postopek iskanja enostavnih preslikav (1:1)

Za iskanje enostavnih preslikav uporabimo enostaven pristop, ki smo ga že omenili, in sicer postopek za reševanje operacij združevanja. Postopek za iskanje enostavnih preslikav vsak element sheme primerja z elementi druge sheme in oceni njuno podobnost. Element z največjo oceno podobnosti, ki je dovolj velika, tj. presega mejno vrednost, predstavlja preslikavo. Pragovno vrednost ocene podobnosti določimo eksperimentalno na podlagi testiranj. Metrika na osnovi Jaccardove podobnosti, ki smo jo definirali v prejšnjem poglavju za izračun podobnosti, je časovno potratna, zato v prvem koraku iskanja preslikav uporabimo kosinusno podobnost ali evklidsko razdaljo. Ti dve metriki predstavljata enostaven in hiter izračun potencialnih kandidatov za preslikavo. Evklidska razdalja v tem primeru zadostuje kot metrika, na podlagi katere izberemo najboljše kandidate. Kandidate pridobimo tako, da vse elemente druge sheme uredimo po velikosti glede na njihovo evklidsko razdaljo, do izbranega elementa, in vzamemo najbližje tri kandidate. V drugem koraku izračunamo podobnost med kandidatom in izbranim elementom z metriko na osnovi Jaccardove podobnosti. Kandidat z najvišjo oceno podobnosti predstavlja preslikavo. Če ocena podobnosti ne dosega mejne vrednosti, preslikava za element ne obstaja.

Postopek preveri preslikave med shemama v obe smeri, S1 $\rightarrow$ S2 in S2 $\rightarrow$ S1. Identificirane preslikave v obeh smereh se ob koncu primerjajo med sabo. Tiste preslikave, ki se pojavijo v obeh seznamih, vzamemo za pravilne in te predstavljajo tudi končni rezultat.

Določanje praga, na podlagi katerega se odločamo, ali element predstavlja preslikavo ali ne, zahteva testiranje in analizo rezultatov. Na podlagi predpostavke, da so si podatki med sabo podobni, bi morala biti zahtevana meja dokaj visoka (blizu 1). Predstavitev podatkov v vektorskem prostoru, ki jo uporabljamo, dovoljuje prekrivanja, četudi podatki niso isti. Ugotovili smo, da predstavitev podatkov s histogrami ne loči najbolje med podatki istega tipa, zato smo pri postopku iskanja enostavnih preslikav uvedli dodatno značilko, s katero poskušamo podatke dodatno ločiti. Če tega ne storimo, lahko pride do dokaj majhnega ujemanja. S testiranjem smo ocenili, da je primerna mejna vrednost 0,3. Če je vrednost manjša, zaradi dvoumnosti preslikavi ne zaupamo in je zato ne predlagamo.

5.4.2 Postopek iskanja kompleksnih preslikav (1:N in N:1)

Največji problem pri iskanju kompleksnih preslikav predstavlja število kombinacij, ki jih je potrebno preveriti v procesu iskanja. Gre za iskanje kombinacije dveh ali več elementov, ki skupaj tvorijo isto podatkovno množico kot element, v katerega se slikajo. Vzemimo dve shemi, kjer imata obe po 100 elementov. Naj shema S1 vsebuje element, ki se slika v kombinacijo dveh elementov sheme S2. Ker ima shema S2 100 elementov, to pomeni, da obstaja $\binom{100}{2} = 4950$ potencialnih kandidatov. Število kandidatov še naprej skokovito naraste s številom elementov v eni kombinaciji, npr. $\binom{100}{3} = 161700$, medtem, ko ima naraščanje števila elementov v shemi manjši vpliv, npr. $\binom{110}{2} = 5995$. Iz statističnega vidika lahko trdimo, da večje kot je število elementov v eni kombinaciji, manjša je verjetnost, da taka preslikava obstaja. To pomeni, da je manj verjetno, da se pojavi preslikava 1:3, kot 1:2. Glede na naše poskuse predlagamo največ 4 elemente v kombinaciji.

Elementi znotraj kombinacije so lahko v različnih relacijah; govorimo o tem, kako jih združiti, da se skupaj preslikajo v nov element. V večini primerov gre za združevanje (konkatenacijo) elementov, pri čemer se v našem primeru to preslika v unijo ovojnic. Enostaven primer je preslikava $\{S1.ImePriimek$

$\leftrightarrow$ *S2.Ime* $\cup$ *S2.Priimek*}, lahko gre tudi za vsoto števil, npr. {*S1.cenaDDV* $\leftrightarrow$ *S2.cena* + *S2.znesekDDV*}. Prototip, ki ga razvijamo v tej magistrski nalogi, podpira združevanje elementov (unijo).

Omejitev števila elementov v kombinaciji ni dovolj stroga, saj lahko še vedno obstaja veliko število kombinacij. V našem primeru hočemo število kombinacij zmanjšati, predvsem zaradi pohitritve izračuna podobnosti, hkrati skušamo v največji meri ohraniti učinkovitost. Zato predlagamo trifazni algoritem za iskanje kompleksnih preslikav.

1. faza: Algoritem sprejeme povzetek ciljnega elementa, za katerega želimo najti preslikavo v ciljni shemi. V prvi fazi se generirajo vse možne kombinacije elementov ciljne sheme vseh dolžin od 1 do m, kjer m predstavlja maksimalno število elementov v kombinaciji. Od dobljenih kombinacij ohranimo tiste, za katere velja, da je povprečna dolžina kombiniranih elementov zelo podobna dolžini ciljnega elementa, za katerega iščemo preslikavo. To določimo tako, da povprečna dolžina kombiniranih elementov pade znotraj pričakovanega intervala dolžine ciljnega elementa s toleranco npr. $\epsilon = \pm 20\%$, ki se je izkazala kot empirično najboljša. Če kombinirani elementi hranijo zapise, ki imajo isto dolžino kot ciljni element, potem kombinacija predstavlja kandidata za preslikavo.

2. faza: V drugi fazi za vse kandidate izračunamo evklidsko razdaljo do povprečne sredinske vrednosti povzetka ciljnega elementa. Pri vsakemu kandidatu za vsak element v kombinaciji izračunamo sredinsko vrednost povzetka in ga zamaknemo z upoštevanjem značilke (podpoglavje 5.2.5). Na koncu za kombinacijo izračunamo še povprečni vektor, tako da seštejemo povprečne vrednosti posameznih elementov in jih delimo. Če kandidat predstavlja pravo preslikavo, potem bo evklidska razdalja do povzetka ciljnega elementa majhna oz. najbližja glede na druge kandidate. Najverjetnejše kandidate dobimo tako, da jih sortiramo padajoče glede na evklidsko razdaljo. Iz pridobljenega seznama vzamemo najboljših N in tako še dodatno reduciramo število

kandidatov, npr. vzamemo najboljših 5, oziroma, v primeru, da imajo isto vrednost, tudi več.

3. faza: Zadnja faza algoritma za iskanje kompleksnih preslikav za najboljše kandidate preveri, ali kateri izmed njih dejansko predstavlja ustrezno preslikavo. Najprej za kandidata izračunamo skupni povzetek vseh elementov v kombinaciji z arhetipsko analizo. To moramo izvesti zaradi izračuna podobnosti po izbrani metriki, saj ta zahteva, da oba povzetka vsebujeta enako število arhetipov. Skupni povzetek kombinacije tudi ustrezno zamaknemo z upoštevanjem značilke (povprečne dolžine zapisov). Tako za vse kandidate izračunamo oceno podobnosti s ciljnim elementom in jih ponovno sortiramo v padajočem vrstnem redu. Če prvi kandidat v sortiranem seznamu dosega dovolj visoko oceno (prag), potem smo našli preslikavo, v nasprotnem primeru element preslikave nima.

Predlagan postopek podpira iskanje enostavnih in kompleksnih preslikav, torej številnosti 1:1, 1:N in N:1. Med testiranjem smo odkrili še eno prednost, in sicer lahko pri tem postopku za izračun podobnosti uporabimo tudi kosinusno podobnost, ki je zelo hitra. Razlog za to sta prvi dve fazi, ki uspešno poskrbita za filtriranje neustreznih kandidatov. Da bi našli vse možne preslikave, moramo tudi ta postopek izvesti za obe smeri, S1 $\rightarrow$ S2 (številnosti 1:N) in S1 $\leftarrow$ S2 (številnosti N:1).

Algoritem, ki ga predlagamo, je neodvisen od predstavitve podatkov, saj bi lahko eventualno uporabili katerokoli izmed naštetih predstavitev v podpoglavju 5.2, če bi poskrbeli za ustrezne transformacije. Ravno tako je algoritem neodvisen od metrike za izračun podobnosti, zato lahko enostavno vpeljemo novo.

5.5 Testiranje in potrditev koncepta

Za testiranje koncepta smo pripravili dve podatkovni shemi in za njiju generirali podatke. Generirani podatki sta za obe shemi zelo podobni, struktura

shema S1		**shema S2**	
Account		**Account**	
BirthDate	580	**Address**	
City	653	**CountryCity**	940
Company	189	**GPSLocation**	1400
Country	287	**PostalCode**	609
Email	600	**StreetAddress**	887
Name	464	**DateOfBirth**	560
Phone	100	**Company**	219
PostalCode	656	**Email**	700
StreetAddress	902	**NameSurname**	919
Surname	455	**Phone**	100

Tabela 5.1: Shemi uporabljeni za testiranje iskanja enostavnih preslikav.

shem pa je drugačna. Za generiranje podatkov smo uporabili internetno storitev *GenerateData* [21], kjer smo določili shemi in obliko podatkov za izvoz, v našem primeru v obliki XML. Strukturi obeh shem sta prikazani v tabeli 5.1 skupaj s številom podatkovnih entitet vseh elementov. Številke predstavljajo število enoličnih žetonov zapisov, pri čemer en žeton predstavlja niz, ki ima začetek in konec določen s presledkom. Vzemimo za primer zapis telefonskih številk, ki imajo obliko 'nn nn nn nn', vseh različnih dvomestnih kombinacij je v tem primeru 100.

Tabela 5.2 prikazuje pravilne preslikave med shemama iz tabele 5.1. Preslikave enostavno prepoznamo že na podlagi imen elementov. Prednost naše metode je, da je neodvisna od imen elementov. V okviru evalvacije našo metodo primerjamo tudi z iskalnikom preslikav COMA CE [17], ki ga razvija skupina za podatkovne baze na oddelku računalniške znanosti na univerzi Leipzig.

Za vse elemente smo generirali 700 podatkovnih zapisov, ki so lahko sestavljeni iz več besed. Faza predprocesiranja v našem ogrodju za integriranje shem poskrbi za razbitje zapisov v žetone in za filtriranje pridobljenih

S1.Account		S2.Account	
BirthDate	—	**DateOfBirth**	
City, Country	—	**CountryCity**	kompleksna preslikava
Company	—	**Company**	
Email	—	**Email**	
Name, Surname	—	**NameSurname**	kompleksna preslikava
Phone	—	**Phone**	
PostalCode	—	**PostalCode**	
StreetAddress	—	**StreetAddress**	

Tabela 5.2: Preslikave med shemama v tabeli 5.1. Vsi elementi imajo preslikave razen **Location** iz sheme S2.

žetonov, tako da v vsakem elementu ostanejo le enolični žetoni. S tem pojasnimo število zapisov v tabeli 5.1. Obe shemi se potem pretvorita v datotečno strukturo, ki jo ogrodje uporablja za shranjevanje vmesnih podatkov. Med te spadajo: izvirna oblika podatkov, filtrirana oblika podatkov, podatki, pretvorjeni v vektorski prostor (matrika), povzetek podatkov in metapodatki (povprečna dolžina). Rezultat ogrodja je predlog preslikav za vhodne sheme.

5.5.1 Preslikave številnosti 1:1

Slika 5.13 prikazuje rezultate iskalnika preslikav v primeru, ko obe shemi hranita skoraj identične instančne podatke. Iskalnik je našel pet enostavnih preslikav od šestih in je glede na pričakovane rezultate iz tabele 5.2 tako zgrešil eno preslikavo: *Email* ↔ *Email*. Podrobna analiza izvajanja je pokazala, da iskalnik za preslikavo ni dosegel dovolj velike podobnosti. Razlog za to je neučinkovita metrika za izračun. V vseh ostalih primerih je podobnost visoka, kot pričakovano.

V naslednjem koraku smo metodo testirali na nekoliko drugačnih podatkih, pri čemer je namen testa bil, da ugotovimo učinkovitost metode v primeru, ko podatki shem nimajo velike podobnosti. Za generiranje ponovno

```
personS1/birth -> personS2/birth    0.960000
personS1/company -> personS2/company    1.000000
personS1/phone -> personS2/phone    0.920000
personS1/postalcode -> personS2/address/postalcode  0.950000
personS1/streetaddress -> personS2/address/streetaddress    0.960000
personS2/address/postalcode -> personS1/postalcode  0.950000
personS2/address/streetaddress -> personS1/streetaddress    0.950000
personS2/birth -> personS1/birth    0.960000
personS2/company -> personS1/company    0.990000
personS2/phone -> personS1/phone    0.920000
Matches found
personS2/birth - personS1/birth
personS2/company - personS1/company
personS2/phone - personS1/phone
personS2/address/postalcode - personS1/postalcode
personS2/address/streetaddress - personS1/streetaddress
>>
```

Slika 5.13: Rezultati enostavnih preslikav iskalnika na osnovi arhetipske analize. Zgornji del, (pred *Matches found*), prikazuje najdene preslikave obeh smeri, spodnji del slike pa končne rezultate.

uporabimo storitev *GenerateData*. Storitev *GenerateData* generira podatke iz obsežnega vira podatkov, pri čemer sami določimo velikost vzorca, ki je v našem primeru bil enak 700. Zaradi velike količine podatkov smo dobili vzorca, ki imata zelo različne podatke. Rezultati testa so prikazani na sliki 5.14. V primerjavi z rezultati, ki smo jih dobili na praktično identičnih podatkih, lahko rečemo, da so precej slabši, kljub temu da smo pri testiranju spustili pragovno mejo podobnosti na 0,2. Mejo smo spustili, ker smo pričakovali manjše podobnosti med povzetki. Kot lahko opazimo je iskalnik odkril le dve pravilni preslikavi, pri čemer je ena izmed njih kompleksna, vendar nepopolna. Govorimo o preslikavi *personNew2/namesurname* $\leftrightarrow$ *personNew1/name*, kjer na desni strani manjka še element *personNew1/surname*.

5.5.2 Preslikave številnosti 1:N in N:1

Algoritem za iskanje kompleksnih preslikav deluje nekoliko drugače, zato pričakujemo, da bomo v primerih, ko shemi ne hranita zelo podobnih po-

```
personNew1/company -> personNew2/company    0.940000
personNew1/name -> personNew2/namesurname   0.270000
personNew1/phone -> personNew2/phone    0.930000
personNew2/company -> personNew1/company    0.940000
personNew2/namesurname -> personNew1/name   0.270000
personNew2/phone -> personNew1/phone    0.930000
Matches found
personNew2/company - personNew1/company
personNew2/namesurname - personNew1/name
personNew2/phone - personNew1/phone
>>
```

Slika 5.14: Rezultati iskalnika v primeru, ko shemi, med katerima iščemo preslikave, ne hranita dovolj podobnih podatkov.

datkov, vseeno dobili nekoliko boljše rezultate, tudi za enostavne preslikave. Pri tem testiranju smo zaradi hitrosti za izračun podobnosti uporabili kosinusno podobnost in ne metriko, ki smo jo predlagali sami. Algoritem za iskanje kompleksnih preslikav rezultate predstavi tako, da izpiše rezultate preslikav obeh smeri. Za to smo se odločili, ker lahko za nek element dobimo več predlogov, med katerimi se težko odločimo za pravilnega. Če bi želeli popolnoma avtomatizirati proces preslikovanja, bi morali uvesti odločitvena pravila, vendar bomo to pustili za nadaljnje raziskovalno delo. Rezultati tako predstavljajo zgolj predloge, zato smo se odločili, da dvoumnih preslikav ne filtriramo in odločitev o tem, katera preslikava je pravila prepustimo uporabniku. Dvoumne preslikave uredimo glede na izračunano oceno podobnosti.

Slika 5.15 prikazuje rezultate iskalnika kompleksnih preslikav v primeru, ko podatkovni shemi hranita praktično identične podatke. Opazimo lahko, da so rezultati zelo dobri. Namreč v primerjavi z iskalnikov enostavnih preslikav je bila odkrita tudi preslikava *Email* $\leftrightarrow$ *Email* in tudi obe kompleksni preslikavi (če upoštevamo predloge z najvišjo oceno podobnosti).

Poglejmo še učinkovitost algoritma pri iskanju preslikav, ko shemi vsebujeta različna vzorca podatkov iz istega vira, ki predstavlja bolj realen problem. Rezultati so prikazani na sliki 5.16. Pri iskanju enostavnih preslikav je algoritem popolnoma pravilno odkril pet od šestih preslikav. Preslikavo

```
personS1/birth - personS2/birth 0.999451
personS1/city - personS2/address/countrycity    0.996342
personS1/company - personS2/company 1.000000
personS1/country - personS2/address/countrycity, personS2/namesurname   0.978438
personS1/email - personS2/email 0.999795
personS1/name - personS2/namesurname    0.986284
personS1/phone - personS2/phone 0.999001
personS1/postalcode - personS2/address/postalcode   0.999275
personS1/streetaddress - personS2/address/streetaddress 1.000000
personS1/surname - personS2/namesurname 0.982013
personS2/address/countrycity -> personS1/city, personS1/country 0.997369
personS2/address/postalcode -> personS1/postalcode  0.999275
personS2/address/streetaddress -> personS1/streetaddress    1.000000
personS2/birth -> personS1/birth    0.999451
personS2/company -> personS1/company    1.000000
personS2/email -> personS1/email    0.999795
personS2/namesurname -> personS1/name, personS1/surname 0.993822
personS2/phone -> personS1/phone    0.999001
```

Slika 5.15: Rezultati iskalnika kompleksnih preslikav v primeru podobnih podatkov. Če podrobneje analiziramo kompleksni preslikavi, vidimo, da je ocena podobnosti v obeh primerih najvišja pri kombiniranih elementih.

StreetAddress ↔ *StreetAddress* ne bomo šteli za uspešno, čeprav je algoritem v rešitvi predlagal tudi pravilno preslikavo, vendar ne z najvišjo oceno podobnosti. Pri iskanju kompleksnih preslikav je bil uspešen le pri eni od dveh, in sicer pri preslikavi *Country, City* ↔ *CountryCity*, pri *Name, Surname* ↔ *NameSurname* pa ne, saj ni odkril pravilne kombinacije. Razlog lahko enostavno razberemo iz rezultatov na sliki 5.16: v prvem delu (pri iskanju preslikav 1:*N*) je algoritem odkril, da se elementa *Name* in *Surname* slikata v element *NameSurname*. Ko ju združimo v kombinacijo, ima ta kandidat manjšo podobnost z *NameSurname* kot sam element *Name*. Na sliki 5.16 vidimo, da ima *Name* z *NameSurname* podobnost 0,98, *Surname* pa 0,96. V splošnem velja, da je podobnost dveh kombiniranih elementov nekje vmes, npr. 0,97. To pomeni, da kombinacija elementov predstavlja slabšo rešitev kot element *Name*.

Če povzamemo rezultate izvedenih testov, lahko zaključimo, da je algoritem za iskanje enostavnih preslikav učinkovit v primeru, ko imata shemi, med katerimi iščemo preslikave, zelo veliko podobnost med podatki. Algoritem,

```
personNew1/birth - personNew2/birth 0.993249
personNew1/company - personNew2/company 0.999325
personNew1/country - personNew2/address/countrycity, personNew2/namesurname 0.972987
personNew1/email - personNew2/email 0.998608
personNew1/name - personNew2/namesurname    0.987741
personNew1/phone - personNew2/phone 0.999001
personNew1/postalcode - personNew2/address/postalcode   0.988309
personNew1/streetaddress - personNew2/address/streetaddress 0.975462
personNew1/surname - personNew2/namesurname 0.969347
personNew2/address/countrycity -> personNew1/city, personNew1/country   0.990454
personNew2/address/postalcode -> personNew1/postalcode  0.988309
personNew2/address/streetaddress -> personNew1/company, personNew1/postalcode, personNew1/streetaddress 0.976385
personNew2/birth -> personNew1/birth    0.993249
personNew2/company -> personNew1/company    0.999325
personNew2/email -> personNew1/email    0.998608
personNew2/namesurname -> personNew1/name   0.987741
personNew2/phone -> personNew1/phone    0.999001
```

Slika 5.16: Odkrite preslikave v primeru, ko shemi vsebujeta različne vzorce podatkov iz istega vira.

ki smo ga predlagali za iskanje kompleksnih preslikav, je izkazal učinkovitost v obeh testih, tudi ko podobnost podatkov ni tako velika. Zraven uspešno odkritih preslikav moramo biti pozorni tudi na pravilno odkrivanje elementov brez preslikav. V tem testu sta obe metodi pravilno odkrili elemente, ki preslikav nimajo. V nadaljevanju metodo testiramo še z nekoliko kompleksnejšo shemo in rezultate neposredno primerjamo z uveljavljenim iskalnikom preslikav COMA CE.

5.6 Primerjava z ostalimi iskalniki preslikav

Realni pokazatelj učinkovitosti predlagane metode je primerjava z uveljavljenimi rešitvami. Ena izmed rešitev je iskalnik COMA CE (Community Edition). COMA CE je ogrodje, ki je namenjeno samodejnemu iskanju semantičnih korespondenc med podatkovnimi shemami, XML formati sporočil in ontologijami. Ogrodje podpira množico knjižnic z algoritmi za iskanje preslikav, ki lahko delujejo na različnih osnovah. V našem primeru bomo test izvedli z iskalnikom, ki deluje na osnovi podatkov shem, zato v nastavitvah COMA CE izberemo *AllContext*.

Pri pregledu literature, ki se navezuje na iskalnike preslikav med podatkovnimi shemami ugotovimo, da večina avtorjev uporablja svoj testni nabor

podatkov, kar velja tudi za ogrodje COMA CE. Avtorji COMA CA so tako na svoji spletni strani objavili referenčne podatke, s katerimi so testirali ogrodje. Za iskalnike, ki delujejo na osnovi instančnih podatkov, avtorji nudijo sheme s podatkovnimi zapisi. Težava je, da ti podatki niso primerni za naš iskalnik, saj za vsako shemo vsebujejo le eno instanco. Da bi bil čim bolj realen pokazatelj uspešnosti iskalnikov preslikav, smo sklenili, da generiramo neodvisne podatke, s pomočjo katerih izvedemo teste. Generirane podatke smo strukturirali, kot določata shemi XML 5.1 in 5.2.

Obe podatkovni shemi hranita podobne podatke, zato sta si tudi sami shemi zelo podobni, le da imata nekoliko drugačno strukturo in v določenih primerih drugačna imena za elemente. Vsi elementi, razen *service_used* in *id* iz sheme *AccountSchema1* ter *cc_type* iz sheme *AccountSchema2*, se pojavijo v obeh shemah. Našteti elementi nimajo preslikav.

Najprej si poglejmo delovanje iskalnika COMA CE, ko so imena elementov takšna, kot so definirana v shemah. Na sliki 5.17 vidimo odkrite preslikave. Opazimo, da v primeru nejasne podobnosti med imeni elementov, preslikave ne odkrije. Preslikavo med elementoma *ssn* ↔ *social_security_number* je npr. zamenjal z *ssn* ↔ *street_address*. Podobno velja tudi za element *s_number*, ki ga je preslikal v *social_security_number*, saj element *s_number* predstavlja hišno številko in ne številko socialnega varstva. Opazimo tudi, da iskalnik ni odkril kompleksnih preslikav *full_name* ↔ *name, surname* in *street_address* ↔ *s_name, s_number* oz. iskanje kompleksnih preslikav sploh ne podpira.

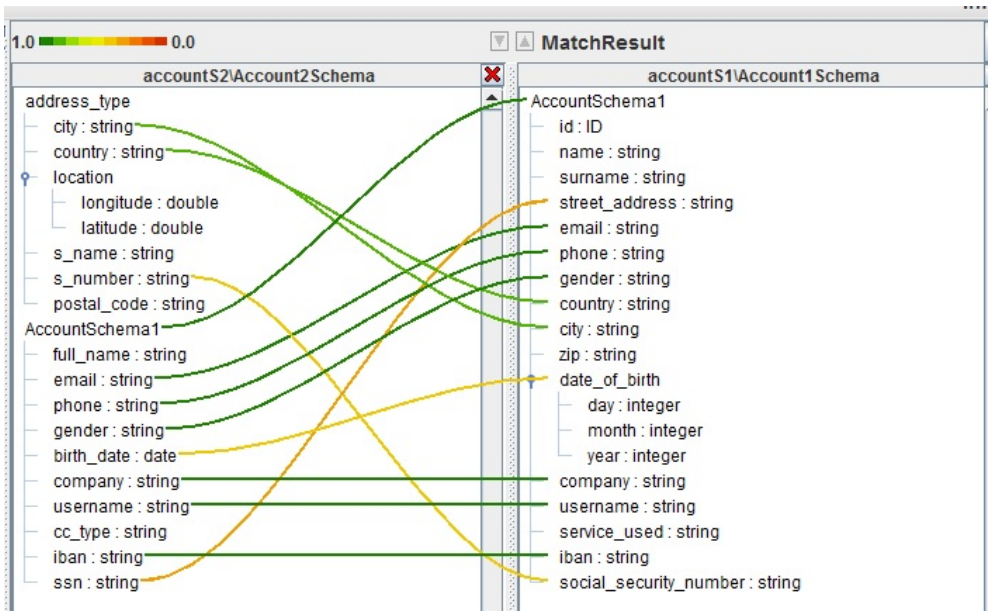
Poglejmo še rezultate našega iskalnika z metodo arhetipske analize, ki so prikazani na sliki 5.18. Analiza rezultatov razkrije, da iskalnik zelo dobro deluje za enostavne preslikave. Odkril je vse preslikave številnosti 1:1, razen za elemente *gender*, *service_used* in *cc_type*. Razlog za to je premajhna podatkovna množica, saj element *gender* vsebuje le dve vrednosti, element *services_used* vsebuje seznam besed, ki obsega zgolj 8 zapisov in *cc_type* z zgolj 16-imi zapisi. Vsi ostali elementi vsebujejo vsaj 290 zapisov. Če je podatkov premalo, ne moremo učinkovito izračunati povzetka podatkov in posledično poiskati ustreznih preslikav. Da je povzetek dovolj dober, potrebujemo v

Shema 5.1: AccountSchema1.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
    <xs:complexType name="date_of_birth_type">
        <xs:sequence>
            <xs:element name="day" type="xs:integer"/>
            <xs:element name="month" type="xs:integer"/>
            <xs:element name="year" type="xs:integer"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="AccountSchema1">
        <xs:sequence>
            <xs:element name="id" type="xs:ID"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="surname" type="xs:string"/>
            <xs:element name="street_address" type="xs:string"/>
            <xs:element name="email" type="xs:string"/>
            <xs:element name="phone" type="xs:string"/>
            <xs:element name="gender" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="zip" type="xs:string"/>
            <xs:element name="date_of_birth" type="date_of_birth_type"/>
            <xs:element name="company" type="xs:string"/>
            <xs:element name="username" type="xs:string"/>
            <xs:element name="service_used" type="xs:string"/>
            <xs:element name="iban" type="xs:string"/>
            <xs:element name="social_security_number" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Shema 5.2: AccountSchema2.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
    <xs:complexType name="geo_location_type">
        <xs:sequence>
            <xs:element name="longitude" type="xs:double"/>
            <xs:element name="latitude" type="xs:double"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="address_type">
        <xs:sequence>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
            <xs:element name="location" type="geo_location_type"/>
            <xs:element name="s_name" type="xs:string"/>
            <xs:element name="s_number" type="xs:string"/>
            <xs:element name="postal_code" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="AccountSchema2">
        <xs:sequence>
            <xs:element name="full_name" type="xs:string"/>
            <xs:element name="email" type="xs:string"/>
            <xs:element name="phone" type="xs:string"/>
            <xs:element name="gender" type="xs:string"/>
            <xs:element name="birth_date" type="xs:date"/>
            <xs:element name="company" type="xs:string"/>
            <xs:element name="username" type="xs:string"/>
            <xs:element name="cc_type" type="xs:string"/>
            <xs:element name="iban" type="xs:string"/>
            <xs:element name="ssn" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```



Slika 5.17: Rezultati iskalnika preslikav COMA CE v primeru, ko vhodni podatkovni shemi nimata jasne podobnosti med vsemi imeni elementov, ki hranijo iste podatke.

Preslikava	COMA CE	Iskalnik na osnovi AA
city	✓	✓
country	✓	✓
postal_code	✗	✓
email	✓	✓
phone	✓	✓
gender	✓	✗
company	✓	✓
username	✓	✓
iban	✓	✓
social_security_no	✗	✓
$\{street\_name, street\_number\}$	✗	✗
$\{name, surname\}$	✗	✓
$\{day, month, year\}$	✗	✓

Tabela 5.3: Primerjava iskalnika preslikav COMA CE in iskalnika na osnovi AA pri uspešnosti odkrivanja pravilnih preslikavah.

povzetku okoli 10% točk iz celotne množice. Iskalnik je uspešno odkril tudi dve od treh kompleksnih preslikav. Za pravilno odkrite kompleksne preslikave štejemo tiste, za katere velja, da imajo najvišjo oceno podobnosti, v primeru, da je za isti element iskalnik odkril več preslikav. Za tretjo kompleksno preslikavo lahko trdimo, da je bila le delno odkrita, saj je en element v kombinaciji napačno odkrit, preostala dva sta bila pravilno odkrita.

Če se osredotočimo izključno na pravilne preslikave, so rezultati obeh iskalnikov predstavljeni v tabeli 5.3. Iskalnik preslikav COMA CE je pravilno odkril **62%** preslikav ($\frac{8}{13}$), od tega **80%** enostavnih preslikav in **0%** kompleksnih preslikav. Medtem, ko je iskalnik, ki ga predlagamo v tej magistrski nalogi, odkril **85%** preslikav ($\frac{11}{13}$), od tega **90%** enostavnih in **66%** kompleksnih preslikav.

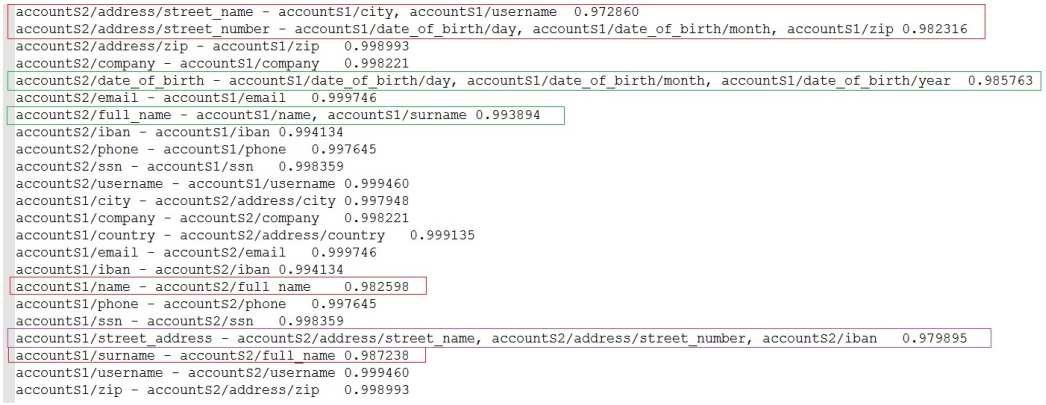
V tabeli 5.4 prikazujemo primerjavo uspešnosti odkrivanja preslikav med

Iskalnik	TP	TN	FP	FN	občutljivost $\frac{TP}{(TP+FN)}$	specifičnost $\frac{TN}{(FP+TN)}$	točnost $\frac{(TP+TN)}{VSI}$	natančnost $\frac{TP}{(TP+FP)}$
COMA CE	8	3	3	2	72,7%	50,0%	68,7%	72.7%
AA	11	3	1	1	91,6%	75,0%	87,5%	91,6%

Tabela 5.4: Primerjava uspešnosti odkrivanja preslikav med uveljavljeno metodo COMA CE in predlagano metodo na osnovi AA.

uveljavljeno metodo COMA CE in predlagano metodo na osnovi AA, kjer uspešnost ocenjujemo s štirimi metrikami, in sicer občutljivostjo, specifičnostjo, točnostjo in natančnostjo. Opazimo lahko, da je pri upoštevanju vseh metrikah naša metoda v povprečju boljša za 20%.

V primerjavi z iskalnikom preslikav COMA CE se je naša metoda obnesla veliko bolje pri kompleksnih preslikavah, nekoliko manj dobro pri enostavnih, če ne spreminjamo imen elementov. Slabost predlagane metode, ki smo jo odkrili, je v velikosti podatkovnih množic (preslikava *gender*). Če je podatkovna množica premajhna, arhetipske analize za iskanje povzetkov ne moremo uporabiti. Po drugi strani se je naša metoda veliko bolje obnesla v primeru kompleksnih preslikav, za razliko od iskalnika COMA CE, ki deluje samo na osnovi imen elementov sheme. To pomeni, da kompleksno preslikavo lahko odkrije le posredno, če ima sestavljeni element enako ime kot element, za katerega iščemo preslikavo. Iskalnik COMA CE se ne zaveda, da sestavljeni element predstavlja podshemo, in da gre za kompleksno preslikavo. V nasprotju je velika prednost naše metode ravno v tem, da se zaveda, kdaj gre za kompleksno preslikavo in poskuša najti elemente, ki jo sestavljajo.



Slika 5.18: Odkrite preslikave z iskalnikom na osnovi arhetipske analize. Zeleni okvir predstavlja pravilno odkrito kompleksno preslikava (2/3), vijolični okvir predstavlja delno nepravilno odkrito preslikavo, rdeči pa nepravilno preslikavo.

Poglavje 6

Sklepne ugotovitve

V okviru tega magistrskega dela smo najprej pregledali obstoječe rešitve na področju integriranja podatkovnih shem in identificirali njihove slabosti. Za tem smo predlagali svojo metodo za iskanje preslikav, ki temelji na povzetkih podatkovnih instanc posameznih elementov shem. Pri pregledu obstoječih metod smo identificirali pristope, ki se uporabljajo za iskanje preslikav. Ti se delijo na tri skupine, ki so določene glede na to, katere informacije se uporabljajo za iskanje preslikav. Prva skupina metod predstavljajo metode, ki za vir podatkov uporabljajo podatke shem - imena elementov, podatkovne tipe, zaloge vrednosti, omejitve in strukturo shem. Drugo skupino predstavljajo metode, ki ujemanja podatkovnih shem iščejo na osnovi podatkovnih instanc. Pri teh metodah se uporabljajo različne tehnike za rudarjenje podatkov in strojno učenje. Tretjo skupino predstavljajo hibridne metode, ki uporabljajo kombinacijo metod iz obeh skupin. Hibridne metode so v praksi najbolj uporabne, saj z uporabo le ene metode težko odkrijemo vse preslikave; poleg tega so metode iz prve in druge skupine komplementarne.

Predlagali smo novo metodo za integriranje podatkovnih shem, ki deluje na osnovi podatkovnih povzetkov posameznega elementa, ki jih pridobimo z arhetipsko analizo. Metoda arhetipske analize je namenjena iskanju približka konveksne ovojnice množice točk, ki deluje na osnovi optimizacijskega algoritma. Povzetek predstavlja reducirano množico podatkovnih točk, s ka-

terimi lahko predstavimo večino podatkov v množici. Metoda temelji na iskanju podobnosti med podatki, ki jih hranijo elementi. Če dva elementa iz različnih shem hranita iste podatke, bosta imela zelo podobne povzetke. Ta pristop je v praksi smiseln, saj vedno integriramo aplikacije v isti domeni, to pomeni, da si zagotovo delijo delež podatkov.

Da smo lahko uporabili metodo arhetipske analize za iskanje povzetkov, smo najprej morali odkriti, kako podatke predstaviti v večdimenzionalnem vektorskem prostoru, da iz njih lahko izračunamo približek konveksne ovojnice. Pri pretvorbi podatkov v pravo obliko smo morali biti pozorni na dve ključni stvari. Prva je ta, da pretvorba ohranja podobnost med podatkovnimi zapisi v izvorni obliki. Druga pomembna stvar pri predstavitvi, je omogočanje primerjave podatkov med različnimi podatkovnimi tipi. Preizkusili smo več metod za pretvorbo podatkov, pri čemer se je za najboljšo izkazala metoda s histogrami. Pretvorba podatkov s histogrami omogoča primerjavo podatkov med različnimi podatkovnimi tipi in hkrati ohranja podobnost pri pretvorbi. Drugi korak pri integriranju podatkovnih shem na osnovi arhetipske analize zahteva izračun med podatkovnimi povzetki. Uporabili smo tri različne pristope, in sicer kosinusno podobnost, evklidsko razdaljo in izračun podobnosti na osnovi Jaccardove podobnosti, ki smo jo zasnovali sami. Ideja predlagane metode je, da na podlagi ugotavljanja, ali se neka točka nahaja znotraj konveksne ovojnice ali ne, izračunamo število točk v preseku dveh ovojnic.

Predlagan postopek smo v celoti zapakirali v ogrodje za integriranje shem. Predstavili in opisali smo posamezne komponente, ki smo jih tudi implementirali. Najpomembnejša komponenta ogrodja je komponenta za iskanje preslikav. Implementirali smo dva algoritma, kjer je eden namenjen iskanju samo enostavnih preslikav številnosti 1:1, drugi omogoča tudi iskanje kompleksnih preslikav 1:N in N:1. Algoritem za iskanje kompleksnih preslikav deluje na osnovi iskanja kombinacij, ki vsebujejo elemente, ki skupaj predstavljajo iste podatke kot element, za katerega iščemo preslikavo. Predlagali smo hevristični algoritem, ki iz vseh možnih kombinacij za preslikavo ohrani le tiste,

ki predstavljajo prave kandidate. Za vse kandidate potem izračunamo podobnost in najboljšega predlagamo za rešitev.

Iskanje kompleksnih preslikav omogoča presenetljivo majhno število obstoječih metod. Predlagana metoda je ena izmed njih. Implementirano metodo smo testirali in jo primerjali z iskalnikom preslikav COMA CE, ki deluje na osnovi podatkov shem. Izkazalo se je, da je predlagana metoda kos uveljavljenim rešitvam na področju. Metoda ima še vedno precej prostora za nadgradnjo, pri čemer bi v prvi vrsti bilo potrebno rešiti problem premajhne množice podatkov elementa, ki ne omogoča izračun povzetka. Ena slabost metode je, da je v praksi dokaj počasna. Počasnost je posledica algoritma za izračun povzetka, saj je iskanje približka konveksne ovojnice v večdimenzionalnem prostoru časovno zelo zahtevno. Predstavili smo tudi nekaj metod za transformiranje podatkov v vektorski prostor. Tudi tu je možno doseči izboljšavo v smislu hitrejšega delovanja. Predlagane metrike so se že izkazale za učinkovite, predvsem kosinusna podobnost, ki je hitra in enostavna, manjka ji le natančnost. Velik potencial za natančnejše metrike predstavljajo tudi geometrijski pristopi.

V tem delu se nismo osredotočali na hitrost metode, pač pa na njeno učinkovitost. Na podlagi rezultatov in primerjave s COMA CE lahko trdimo, da je koncept metode uspešno potrjen in ima zelo velik potencial za nadaljnji razvoj. Učinkovitost naše metode smo ocenili na podlagi štirih metrik, in sicer občutljivostjo, specifičnostjo, točnostjo in natančnostjo. Testiranje je za posamezno metriko razkrilo naslednje rezultate: 91% za občutljivost, 75% za specifičnost, 87% za točnost in 91% za natančnost. V primerjavi z iskalnikom COMA CE opazimo, da je pri upoštevanju vseh metrik naša metoda v povprečju boljša za 20%.

Literatura

[1] E. Canhasi, "Multi-document summarization via Archetypal Analysis of the content-graph joint model", *Knowledge and Information Systems*, št. 1, zv. 37, str. 1–22, 2013.

[2] O. A. Mehdi, H. Ibrahim, L.S. Affendey "Instance based Matching using Regular Expression", v zborniku: Procedia Computer Science zv. 10, str. 688–695, 2012.

[3] M. G. de Carvalho, A.H.F. Laender, M.A. Goncalves, A.S. da Silva "An evolutionary approach to complex schema matching", *Information Systems*, št. 3, zv. 38, str. 302–316, 2013.

[4] M. G. de Carvalho, A.H.F. Laender, M.A. Goncalves, A.S. da Silva "A genetic programming approach to record deduplication", *Transactions on Knowledge and Data Engineering*, št. 3, zv. 24, str. 399–412, 2012.

[5] E. Rahm, P.A. Bernstein "A survey of approaches to automatic schema matching", The VLDB Journal 10, str. 334-250, november 2001.

[6] Do Hong Hai "Schema matching and mapping-based data integration", doktorsko delo, Univerza v Leipzig, avgust 2005.

[7] P. A. Bernstein, J. Madhavan, E. Rahm "Generic schema matching, ten years later", *PVLDB*, št. 11, zv. 4, str. 695–701, 2011.

[8] Do Hong Hai, E. Rahm "Matching large schemas: Approaches and evaluation", *Information Systems*, št. 6, zv. 32, str. 857–885, september 2007.

[9] A. Cutler, L. Breiman "Archetypal analaysis", *Technometrics*, št. 4, zv. 36, str. 338–347, 1994.

[10] C. Bauckhage, C. Thurau "Making archetypal analysis practical", *Pattern Recognition*, št. 5748, str. 272–281, 2009.

[11] M. Morup, L. K. Hansen "Archetypal analysis for machine learning and data mining", *Neurocomputing*, zv. 80, str. 54–63, 2012.

[12] I. Jolliffe "Principal Component Analysis", 2005.

[13] D. D. Lee, H. S. Seung "Algorithms for Non-negative Matrix Factorization", *NIPS*, str. 556–562, 2000.

[14] R. Poli, J. Koza "Chapter 6: Genetic Programming" *Search Methodologies*, str. 143–185, 2014.

[15] W. Zhang, T. Yoshida, X. Tang "A comparative study of TF*IDF, LSI and multi-words for text classification", *Expert Systems with Applications*, št. 3, zv. 38, str. 2758–2765, 2011.

[16] C++ programska knjižnica armadillo. Dostopno na: `http://arma.sourceforge.net/` (pridobljeno 8.9.2014)

[17] COMA 3.0 Comunity Edition. Dostopno na: `http://dbs.uni-leipzig.de/en/Research/coma_index.html` (pridobljeno 12.9.2014)

[18] J. B. Kruskal "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis", *Psychometrika*, št. 29, str. 1–27, 1964.

[19] I. M. Crawford, R. A. Lomas "Factory Analysis — a Tool for Data Reduction", *European Journal of Marketing*, št. 7, zv. 14, str. 414–421, 1980

[20] I. E. Sutherland, G. W. Hodgman "Reentrant polygon clipping", *Communications of the ACM*, št. 1, zv. 17, str. 32–42, 1974.

[21] Storitev GenerateData. Dostopno na: `http://www.generatedata.com/` (pridobljeno 16.9.2014)

[22] T. Pang-Ning, M. Steinbach, V. Kumar "Introduction to data mining", 2006.

[23] Evklidska razdalja, Wolfram. Dostopno na: `http://mathworld.wolfram.com/EuclideanMetric.html` (pridobljeno 8.9.2014)

[24] D. Knuth "Section 6.1: Sequential Searching", *The Art of Computer Programming 3*, str. 396–408, 1997.

[25] Frobenius norm, Wolfram. Dostopno na: `http://mathworld.wolfram.com/FrobeniusNorm.html` (pridobljeno 12.9.2014)

[26] L. Yujian, L. Bo "A normalized Levenshtein distance metric", *Pattern Analysis and Machine Intelligence*, št. 6, zv. 29, str. 1091–1095, 2005.