Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Peter Remec

Integracija problema izomorfnega podgrafa v sistem ALGator

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Jurij Mihelič
Somentor: doc. dr. Tomaž Dobravec

Ljubljana, 2015

Rezultati diplomskega dela so intelektualna lastnina avtorja. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Integracija problema izomorfnega podgrafa v sistem ALGator

Tematika naloge:

Eksperimentalno testiranje in primerjava algoritmov ima velik praktičen pomen v računalništvu. Sistem ALGator omogoča avtomatizacijo testiranja algoritmov za različne probleme in analizo pridobljenih podatkov.

V diplomski nalogi se osredotočite na problem iskanja izomorfnega podgrafa, ki je eden izmed najpomembnejših problemov s področja iskanja vzorcev v grafih. Zanj obstaja več različnih algoritmov, poleg tega nastajajo tudi novi. V povezavi s problemom opišite osnovne pojme, predstavite in implementirajte več algoritmov za njegovo reševanje. Problem izomorfnega podgrafa integrirajte v sistem ALGator, pri čemer uporabite eno izmed standardnih baz testnih primerov. Poleg tega v sistem integrirajte implementirane algoritme in ga uporabite za njihovo eksperimentalno primerjavo.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Peter Remec, z vpisno številko **63040143**, sem avtor diplomskega dela z naslovom:

Integracija problema izomorfnega podgrafa v sistem ALGator

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Jurija Miheliča in somentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 25. januarja 2015 Podpis avtorja:

Za strokovno pomoč in nasvete pri izdelavi diplomskega dela se iskreno zahvaljujem mentorju doc. dr. Juriju Miheliču in somentorju doc. dr. Tomažu Dobravcu.

Posebna zahvala velja tudi staršem in partnerki Sanji, ki so mi stali ob strani in me spodbujali vse do zaključka študija.

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Opis področja 1

1.2 Motivacija 3

1.3 Pregled dela 3

2 Osnovne definicije **5**

2.1 Graf 5

2.2 Podgraf 7

2.3 Morfizmi 8

2.4 Izomorfizem 9

2.5 Problem podgrafnega izomorfizma 11

2.6 Časovna zahtevnost problema 11

3 Algoritmi **13**

3.1 Ullmannov algoritem 13

3.2 Izboljšani Ullmannov algoritem 20

3.3 Algoritem RI 26

4 Integracija v sistem ALGator **33**

4.1 ALGator 33

4.2 Delovanje sistema 33

4.3 Opredelitev problema 34

4.4 Implementacija algoritmov 35
4.5 Integracija problema izomorfnega podgrafa 36

5 Eksperimentalna primerjava algoritmov 43
5.1 Testne množice 43
5.2 Testi 43
5.3 Rezultati eksperimenta 45

6 Sklepne ugotovitve 57

A Implementacija javanskih razredov v ALGator-ju 59

Seznam uporabljenih kratic

kratica	angleško	slovensko
CSV	comma-separated values	vrednosti, ločene z vejico
JSON	JavaScript object notation	JavaScript standard opisovanja objektov
JAR	Java archive	arhivska datoteka z Java razredi
CAD	Computer aided design	računalniško podprto načrtovanje
BFS	breadth-first search	strategija preiskovanja drevesa v širino

Povzetek

V diplomskem delu obravnavamo problem podgrafnega izomorfizma in njegovo integracijo v sistem ALGator. Iskanje izomorfnih podgrafov je dandanes prisotno na večih znanstvenih področjih, zato se vseskozi pojavljajo novi algoritmi za reševanje problema. Podrobneje smo opisali Ullmannov algoritem, izboljšani Ullmannov algoritem in algoritem RI, ki je najnovejši izmed naštetih. Njihove implementacije, katere so primerne za iskanje izomorfizmov tako na usmerjenih kot neusmerjenih neoznačenih grafih, smo zaradi integracije v sistem ALGator napisali v programskem jeziku Java. Sistem je namenjen predvsem raziskovalcem na področju razvoja algoritmov, saj omogoča enostavno testiranje njihove učinkovitosti in analizo pridobljenih rezultatov. Funkcionalnosti sistema smo v praksi uporabili na izbranemu problemu. Poleg definicije problema smo v projekt v ALGator-ju vključili vse implementirane algoritme (izboljšani Ullmannov algoritem je podan v dveh različicah) in opazovali njihovo učinkovitost. Eksperiment smo izvedli na več kot 50000 različnih parih grafov. Z analizo rezultatov v ALGator-ju smo pokazali, da je algoritem RI v primerjavi z ostalimi tremi algoritmi najhitrejši. Na drugi strani se je najslabše izkazal Ullmannov algoritem, njegovi izboljšani različici pa sta se v določenih scenarijih celo približala algoritmu RI. Z integracijo problema v sistem ALGator smo predstavili njegove zmogljivosti in navedli predloge za izboljšave.

Ključne besede: graf, podgraf, algoritem, izomorfizem, ALGator.

Abstract

This thesis deals with a subgraph isomorphism problem and its integration into ALGator system. Detection of isomorphic subgraph is present in many scientific fields nowadays, therefore new problem solving algorithms constantly appear. We focus on a detailed description of Ullmann algorithm, improved Ullmann algorithm and RI algorithm, the most recent among listed. Those algorithms, which are suitable for isomorphism detection in directed or undirected unlabeled graphs, were implemented in Java programming language for the purpose of integration into ALGator. The system is intended for the use of algorithm development researchers, providing simplified testing of algorithm's efficiency and analysis of testing results. We have applied system functionalities on the selected problem. In addition to a problem definition we have included all the implemented algorithms into ALGator project (improved Ullmann algorithm is given in two versions) and observed their efficiency. We have executed the experiment on more than 50000 different pairs of graphs. The analysis of testing results with ALGator shows, that RI algorithm is the fastest one among all four algorithms. On the other hand Ullmann algorithm performed the worst, while the performance of its improved versions were even comparable with RI algorithm in certain scenarios. By integrating the problem into ALGator we have presented its capabilities and proposed some improvements.

Keywords: graph, subgraph, algorithm, isomorphism, ALGator.

Poglavje 1

Uvod

1.1 Opis področja

1.1.1 Testiranje algoritmov

Pomembnejša opravila tekom razvoja novega algoritma so meritve njegove učinkovitosti, analiza pridobljenih podatkov ter primerjava z že obstoječimi algoritmi. Pomanjkanje razvijalskih orodij, ki bi te naloge celovito pokrila in jih čimbolj poenostavila, je privedlo do realizacije sistema *ALGator*.

Podobna orodja, namenjena testiranju programske kode, so se sicer pojavila že prej. Za primer vzemimo Java knjižnico *Caliper* [19], ki je uporabna za testiranje manjših izsekov kode (t.i. microbenchmarking), ponuja parametrizirano izvajanje in grafični prikaz rezultatov. Podobne lastnosti ima tudi knjižnica *Perf4J* [20], ne ena ne druga pa ne izpolnjuje naših želja glede zmogljivosti sistema.

Poglavitne lastnosti ALGator-ja so:

- primeren za testiranje javanske kode,
- merjenje časa izvajanja algoritmov (najmanjši, največji, povprečen),
- izvajanje algoritmov z različnimi vhodnimi podatki (parametriziran vhod),
- trajno shranjevanje rezultatov izvajanja testov v datoteko,
- grafični prikaz rezultatov izvajanja testov.

Pravkar naštete funkcionalnosti so v samem orodju že implementirane, hkrati pa so prilagodljive za specifike posameznega problema. Za primer vzemimo shranjevanje rezultatov izvajanja testov. Sistem ima samo realizacijo shranjevanja že implementirano, uporabnik mu posreduje le vrsto podatkov (preko javanskih razredov ali konfiguracijskih datotek), ki jih želi shraniti za kasnejšo analizo. Prav tako uporabniku ni potrebno skrbeti za inicializacijo časovnika, ki beleži čas izvajanja algoritma. Naloga uporabnika se nanaša le na specifične nastavitve kot so največji dopustni čas izvajanja testnega primera in želena vrsta meritve časa (npr. največji, najmanjši, povprečni čas).

1.1.2 Problem podgrafnega izomorfizma

S problemom izomorfnega podgrafa se dandanes srečujemo na več različnih področjih. V kemijski informatiki na ta način iščejo podobnosti med kemijskimi spojinami na podlagi njihovih kemijskih formul. Bioinformatiki algoritme za iskanje izomorfnih podgrafov s pridom izkoriščajo pri preučevanju PPI omrežij (Protein-protein interaction networks) [12]. Ostali aktualni primeri uporabe so še razpoznavanje napak oziroma vzorcev v slikah [28], prepoznavanje (delnih) prstnih odtisov [29], analiza socialnih omrežij [21], mapiranje navideznega omrežja v fizično na področju virtualizacije omrežij [22] in analiza elektronskih vezij, ki igra pomembno vlogo v CAD oblikovanju.

Za reševanje problema izomorfnega podgrafa imamo na voljo nemalo različnih algoritmov. Najstarejši izmed njih je Ullmannov algoritem [2], ki temelji na iskanju rešitve s sestopanjem v primeru neobetavnosti delne rešitve. Izboljšano različico tega algoritma so leta 2012 predstavili v [9, 10]. Naslednji izmed standardnih algoritmov je VF oziroma izboljšana različica VF2 [24, 25], ki postopoma v delno rešitev dodaja nove pare vzorčnih in ciljnih vozlišč. Izmed novejših algoritmov velja omeniti Subsea [27] in RI [12], katera se osredotočata na čimprejšnjo detekcijo neobetavnih delnih rešitev z vpeljavo različnih omejitev. Eksperimentalne primerjave različnih algoritmov si lahko bralec med drugim ogleda tudi v [14, 23, 25, 26].

V okviru diplomskega dela smo se odločili za implementacijo Ullmannovega algoritma, njegove izboljšane verzije in algoritma RI. Izboljšana različica Ullmannovega algoritma nastopa v dveh oblikah, kateri se razlikujeta po načinu določanja vrstnega reda obiskovanja vozlišč pri iskanju rešitve, zato ju pri testiranju in ka-

snejši analizi obravnavamo ločeno. Implementacije vseh naštetih algoritmov smo vnesli v sistem ALGator, jih testirali na veliki bazi najrazličnejših grafov ter analizirali pridobljene rezultate.

1.2 Motivacija

Poglavitni cilj te diplomske naloge je integrirati problem izomorfnega podgrafa s pripadajočimi algoritmi, ki ta problem rešujejo, v sistem ALGator.

To razvijalsko orodje je razmeroma novo in se vseskozi nadgrajuje. Tudi v času nastajanja tega prispevka se je pokazala potreba po nekaterih dodatnih funkcionalnostih. Pričakujemo, da se bodo tudi v prihodnje pojavljale želje uporabnikov po nadgradnji sistema. Problem, ki ga v tem diplomskem delu obravnavamo, je sicer eden izmed prvih resnih projektov, vključenih v ALGator, zato ima tudi funkcijo podajanja povratnih informacij o uporabi sistema in morebitnih nepravilnostih v njegovem delovanju. Z integracijo novih problemov in pripadajočih algoritmov bi lahko ALGator postal nekakšna zbirka, v sklopu katere bo lahko raziskovalec na področju algoritmov le-te na enostaven način primerjal med seboj in analiziral njihovo učinkovitost.

1.3 Pregled dela

To diplomsko delo smo vključno z uvodom razdelili na šest vsebinsko zaokroženih poglavij.

Po uvodu smo v drugem poglavju navedli definicije pojmov, ki se uporabljajo v nadaljnjih sklopih. Natančneje smo opisali grafe oziroma podgrafe ter z njimi povezanime pojme. Predstavili smo formalne zapise grafov ter relacij med njimi. V skladu s tematiko diplomskega dela smo definirali različne morfizme s poudarkom na izomorfizmu, predvsem podgrafnemu. Poglavje zaključimo z umestitvijo problema izomorfnega podgrafa v razred NP-polnih problemov in navedbo časovne zahtevnosti.

V tretjem poglavju se osredotočamo na natančnejšo predstavitev izbranih algoritmov, ki rešujejo problem podgrafnega izomorfizma. Prvi izmed njih je Ullmannov algoritem, ki je hkrati tudi najstarejši algoritem za reševanje tovrstnih proble-

mov. Sledi izboljšani Ullmannov algoritem, prispevek slovenskih avtorjev, kateri v primerjavi z osnovno različico vsebuje nekaj popravkov z namenom zmanjšanja tako časovne kot prostorske zahtevnosti. Zadnji opisani algoritem se imenuje RI in je trenutno eden izmed najobetavnejših na tem področju.

V četrtem poglavju predstavimo osrednjo nalogo tega diplomskega dela, integracijo izbranih algoritmov v sistem ALGator. Začnemo s splošnimi informacijami o sistemu, nadaljujemo pa z natančnejšim opisom delovanja. Razložili smo splošne naloge, ki jih mora administrator projekta v ALGatorju opraviti za pravilno umestitev problema in algoritmov znotraj sistema. Končno smo opisali samo integracijo problema podgrafnega izomorfizma in specifike, ki se nanj nanašajo.

Sledi poglavje z opisanim potekom testiranja algoritmov in primerjave med njimi. Predstavili smo baze grafov, ki smo jih pri testiranju uporabili. Natančneje smo opisali tudi strukturo testnih množic ter grupiranje grafov v posamezno množico. Sledi potek priprave in izvedbe testov v sistemu ALGator. Poglavje zaključimo s predstavitvijo rezultatov eksperimenta in njihovo interpretacijo.

Zadnje poglavje smo namenili sklepni misli glede opravljenega dela in končnega rezultata. V njem ocenimo tudi vrednost našega prispevka in navedemo predloge za nadaljnje raziskovanje področja izomorfnih podgrafov kot tudi razvoj sistema ALGator.

Poglavje 2

Osnovne definicije

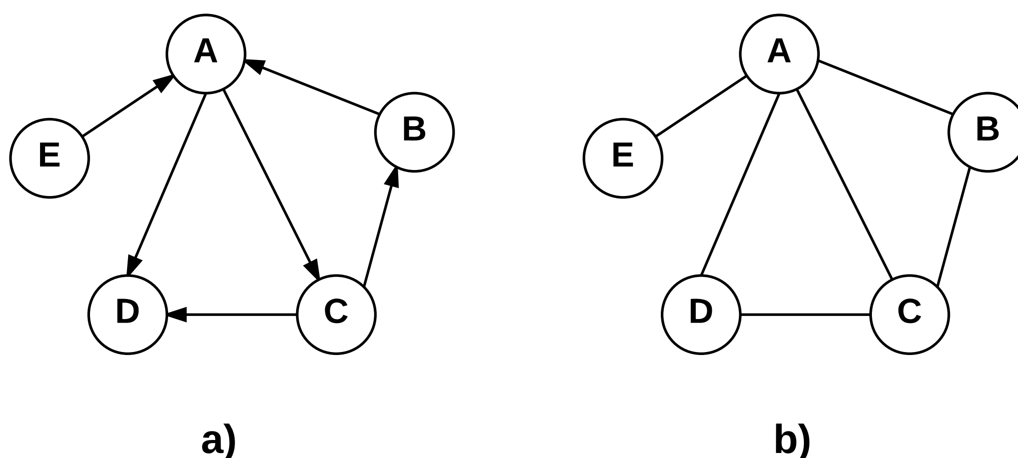
2.1 Graf

V teoriji grafov je pojem *graf* definiran kot urejen par $G = (V, E)$ z množico *vozlišč* V (angl. vertices) in množico *povezav* E (angl. edges), kjer vsaka izmed povezav sestoji iz para elementov iz množice V (formalno $E \subseteq V \times V$) [6]. Za vozlišči, ki tvorita posamezno povezavo, pravimo, da sta *sosednji.* Če za vozlišči $u, v \in V$ velja relacija *sosednosti* (angl. adjacency), uporabimo zapis $u \sim v$. V primeru, da je razmerje $|E|$ / $|V|$ veliko, pravimo, da je graf *góst*, v nasprotnem primeru je *redek.* Vozlišče, katerega največja razdalja do poljubnega vozlišča v grafu je najmanjša, imenujemo središče grafa (angl. graph center).

Graf je lahko *usmerjen* (angl. directed graph oz. digraph) ali *neusmerjen* (angl. undirected graph). V usmerjenem grafu (primer na sliki 2.1 a) povezava $e = (u, v)$ predstavlja urejen par vozlišč $u, v \in V$. Prvo vozlišče se imenuje *začetek povezave* (angl. head), drugo pa *konec povezave* (angl. tail). Z zapisom $e = (u, v)$ torej povemo, da povezava e poteka od vozlišča u proti vozlišču v. Številu povezav, ki imajo vozlišče $v \in V$ za začetek povezave, pravimo izstopna stopnja (angl. outdegree) vozlišča v in jo označimo z $deg^+(v)$. Število povezav, ki imajo vozlišče $v \in V$ za konec povezave, imenujemo *vstopna stopnja* (angl. indegree) vozlišča v in jo označimo z $deg^-(v)$. Če velja $deg^-(v) = 0$, vozlišču v pravimo *izvor* (angl. source), če pa velja $deg^+(v) = 0$, pa ga imenujemo *ponor* (angl. sink). Graf z vozliščem v, za katerega velja $deg^-(v) = deg^+(v) = 0$, je *nepovezan*, vozlišče v

pa je *osamljeno*.

Povezava $e = \{u, v\}$ v neusmerjenem grafu (primer na sliki 2.1 b) je z razliko od povezave $e = (u, v)$ v usmerjenem grafu neurejen par vozlišč oz. podmnožica množice V z natanko dvema elementoma. To pomeni, da povezava nima usmeritve in se potemtakem zapisa $\{u, v\}$ in $\{v, u\}$ ne razlikujeta. Stopnja vozlišča $v \in V$ v neusmerjenem grafu je število povezav, ki vsebujejo to vozlišče. Označimo jo z $deg(v)$.



Slika 2.1: Primer *a*) usmerjenega in *b*) neusmerjenega grafa.

Za vsak neusmerjeni graf velja *lema o rokovanju* [7]:

$$\sum_{v \in V} deg(v) = 2|E|,$$

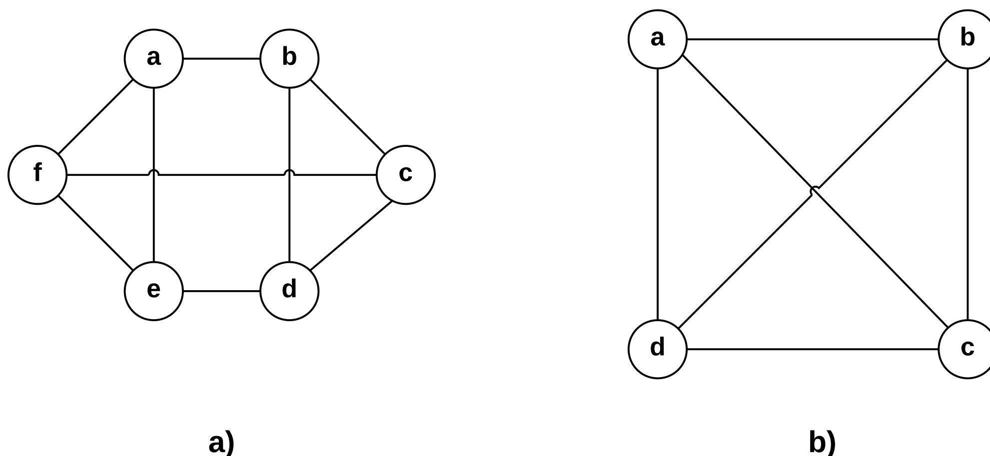
o vsoti stopenj usmerjenega grafa pa govori digrafska verzija, imenovana tudi *dilema o rokovanju*:

$$\sum_{v \in V} deg^+(v) = \sum_{v \in V} deg^-(v) = |E|.$$

Za neusmerjen graf G je *(odprta) soseščina* (angl. open neighborhood) vozlišča $v \in V$ množica $N_G(v) = \{u \in V \mid \{u, v\} \in E\}$, *zaprta soseščina* (angl. closed neighborhood) vozlišča $v \in V$ pa množica $N_G[v] = N_G(v) \cup \{v\}$. Soseščina množice $A \subseteq V$ je definirana kot unija soseščin vozlišč v množici A, tj. $N_G(A) = \bigcup_{v \in A} N_G(v)$.

Utežen graf (angl. weighted graph) je posebna oblika grafa, katerega povezave imajo določene *uteži* (angl. weights). Te uteži lahko predstavljajo ceno, razdaljo ali katerokoli drugo količino, odvisno od problema. Primera problema, pri katerem uporabljamo utežen graf, sta iskanje najcenejše poti in največjega pretoka. Uteženemu grafu enostavno rečemo tudi *omrežje* (angl. network).

Polni graf (angl. complete graph) je neusmerjen graf, v katerem so vsa vozlišča neposredno povezana z vsemi ostalimi. *Regularni graf* (angl. regular graph) je graf z vozlišči enake stopnje. Če imajo vsa vozlišča v regularnem grafu stopnjo k, pravimo, da je graf *k-regularen.* Slika 2.2 prikazuje 3-regularen in poln graf.

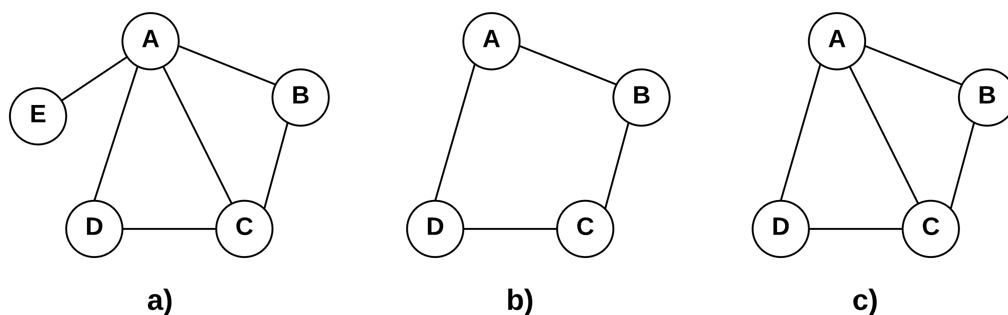


Slika 2.2: Primer *a*) 3-regularnega in *b*) polnega grafa.

2.2 Podgraf

Ker je poglavitna tema tega diplomskega dela podgrafni izomorfizem, je na mestu navesti tudi definicijo pojma *podgraf* (angl. subgraph). Graf $G' = (V', E')$ je podgraf danega grafa $G = (V, E)$, če velja $V' \subset V \wedge E' \subset E$. Iz tega sledi, da podgraf G' ne sme vsebovati vozlišč ali povezav, ki niso vsebovani tudi v grafu G.

Graf $G' = (V', E')$ je *inducirani podgraf* (angl. induced subgraph) grafa $G = (V, E)$, če $V' \subseteq V$ in za vsak par $u, v \in V'$ velja $(u, v) \in E' \Leftrightarrow (u, v) \in E$. Graf $G' = (V', E')$ je *delni podgraf* (angl. partial subgraph) grafa $G = (V, E)$, če $V' \subseteq V$ in za vsak par $u, v \in V'$ velja $(u, v) \in E' \Rightarrow (u, v) \in E$. Slika 2.3 predstavlja primer delnega in induciranega podgrafa.



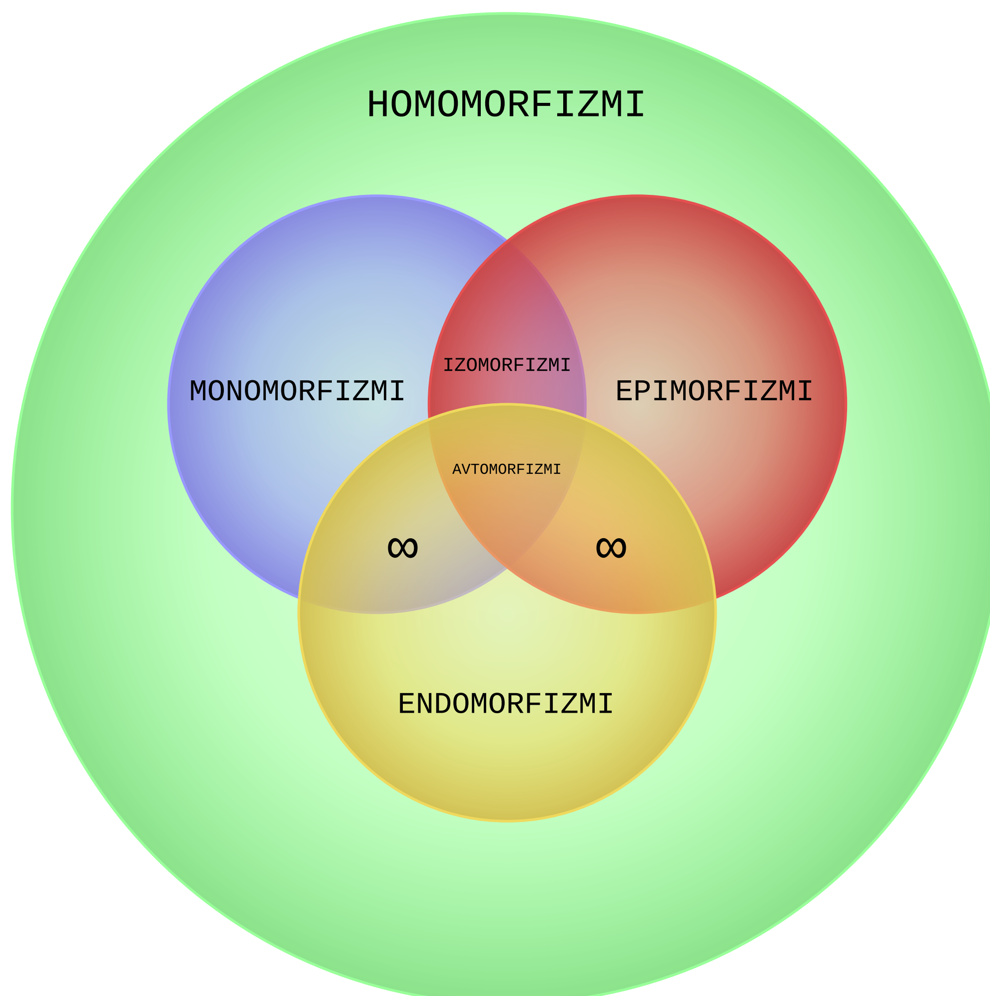
Slika 2.3: Graf b) je delni, graf c) pa inducirani podgraf grafa a).

Klika (angl. clique) v (neusmerjenem) grafu $G = (V, E)$ je množica $V' \subseteq V$, za katero velja $\forall u, v \in V' \; \exists (u, v) \in E$ [16]. Kliko, kjer je $|V'| = k$, imenujemo k-klika. Primer 4-klike je prikazan na sliki 2.2 b).

2.3 Morfizmi

Pojem *morfizem* (angl. morphism) se uporablja na več matematičnih področjih in v splošnem pomeni preslikavo iz ene matematične strukture v drugo. Na področju grafov si lahko morfizem predstavljamo kot funkcijo, ki preslika *vzorčni graf* $G_p = (V_p, E_p)$ v *ciljni graf* $G_t = (V_t, E_t)$. Takšni funkciji, ki upošteva strukturo grafov G_p in G_t (vsak par sosednjih vozlišč iz G_p se preslika v par sosednjih vozlišč v G_t), imenujemo *grafni homomorfizem*. Formalno je definiran kot preslikava $f : V_p \rightarrow V_t$, da za vsak par $(u, v) \in E_p \Rightarrow (f(u), f(v)) \in E_t$. Pogosto ga zapišemo kar $f : G_p \rightarrow G_t$.

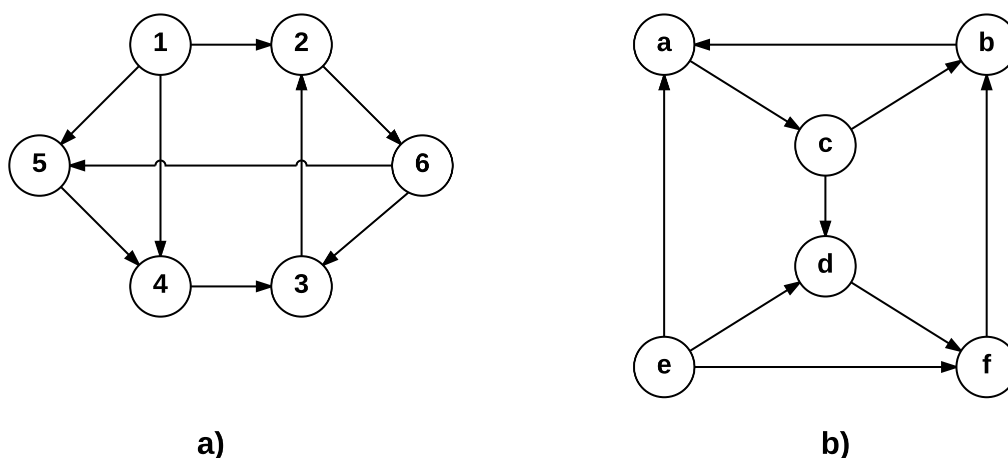
V nadaljevanju so navedeni specifični tipi homomorfizmov. Injektivnemu homomorfizmu pravimo *monomorfizem*, surjektivni homomorfizem je *epimorfizem*, bijektivni homomorfizem pa imenujemo *izomorfizem*. Slednji je torej monomorfizem in epimorfizem hkrati. Homomorfizem iz grafa G_p v samega vase se imenuje *endomorfizem*, če pa je endomorfizem hkrati tudi izomorfizem, pravimo tej preslikavi *avtomorfizem*. Razmerja med različnimi morfizmi prikazuje slika 2.4.



Slika 2.4: Predstavitev relacij med različnimi morfizmi z Vennovim diagramom. Vir: [5]

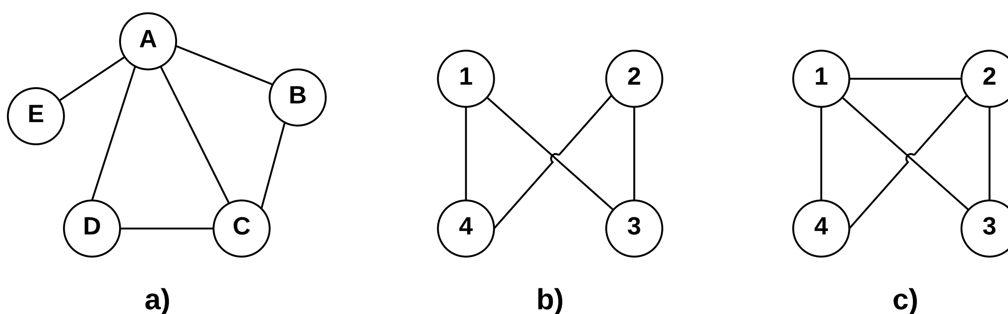
2.4 Izomorfizem

Izomorfizem iz grafa G_p v graf G_t je bijektivna preslikava $f : V_p \to V_t$, za katero velja $(u, v) \in E_p \Leftrightarrow (f(u), f(v)) \in E_t$ (slika 2.5) [17]. Če med grafoma G_p in G_t obstaja izomorfizem, velja zapis $G_p \simeq G_t$. Grafni izomorfizem torej predstavlja ekvivalenčno relacijo nad grafi.



Slika 2.5: Primer izomorfnih grafov s preslikavo $f = \{(1,e),(2,a),(3,b),(4,f),(5,d),(6,c)\}$.

Delni podgrafni izomorfizem med grafoma G_p in G_t je injektivna preslikava $f : V_p \rightarrow V_t$, za katero velja $(u,v) \in E_p \Rightarrow (f(u),f(v)) \in E_t$. *Inducirani podgrafni izomorfizem* med grafoma G_p in G_t je injektivna preslikava $f : V_p \rightarrow V_t$, za katero velja $(u,v) \in E_p \Leftrightarrow (f(u),f(v)) \in E_t$. Obe vrsti podgrafnega izomorfizma sta predstavljeni na sliki 2.6.



Slika 2.6: Graf b) je delni izomorfni podgraf, graf c) pa inducirani izomorfni podgraf grafa a). V obeh primerih je preslikovalna funkcija $f = (1,A),(2,C),(3,B),(4,D)$.

2.5 Problem podgrafnega izomorfizma

Poznamo več različic problema podgrafnega izomorfizma [4], katere v osnovi delimo glede na vrsto vhodnih podatkov (tabela 2.1) in želeno obliko rezultata (tabela 2.2).

Dana sta grafa G_p in G_t. Preverjamo, ali je G_p izomorfni podgraf grafa G_t.

Različica	**Vrsta vhodnih podatkov**
splošen problem	grafa G_p in G_t sta vhodna podatka
omejen problem	grafa G_p in G_t, ki sta vhodna podatka, pripadata določenemu razredu (npr. drevesa, ravninski grafi,...)
fiksen problem	graf G_p je vhodni podatek, graf G_t je fiksen, ali obratno

Tabela 2.1: Različice problema podgrafnega izomorfizma glede na vrsto vhodnih podatkov.

Različica	**Željena oblika rezultata**
odločitveni problem	graf G_t vsebuje / ne vsebuje podgraf, izomorfen grafu G_p
iskalni problem	prva najdena pojavitev grafa G_p kot izomorfni podgraf grafa G_t
preštevalni problem	število podgrafov grafa G_t, ki so izomorfni grafu G_p
naštevalni problem	vse pojavitve grafa G_p kot podgraf grafa G_t

Tabela 2.2: Različice problema podgrafnega izomorfizma glede na želeno obliko rezultata.

Problem podgrafnega izomorfizma v tem diplomskem delu je opredeljen kot splošen preštevalni problem, katerega rešujejo algoritmi, natančneje opisani v kasnejših poglavjih.

2.6 Časovna zahtevnost problema

Po časovni zahtevnosti odločitveni problem podgrafnega izomorfizma spada v razred NP-polnih problemov. Dokaz temelji na prevedbi *problema klike* (angl. clique problem), ki je znano NP-poln. Pri tem problemu vhod predstavlja graf G in število k, sprašujemo pa se, ali v G obstaja k-klika.

Problem klike prevedemo na problem izomorfnega podgrafa tako, da za graf G_p vzamemo kliko K_k, za G_t pa G. S tem postane odgovor na vprašanje obstoja podgrafnega izomorfizma med G_p in G_t enak odgovoru na problem klike za vhoda G in k. Ker je problem klike NP-poln in smo prevedbo izvedli v polinomskem času, lahko sklepamo, da v razred NP-polnih problemov spada tudi problem podgrafnega izomorfizma [18].

Alternativni dokaz NP-polnosti gre preko prevedbe *problema iskanja Hamiltonovega cikla (obhoda)*, še enega člana razreda NP-polnih problemov. V tem primeru je vhod graf G, za katerega preverjamo, ali vsebuje Hamiltonov cikel. Naj bo C_n cikel z istim številom vozlišč kot graf G (t.j. n). Če v grafu G najdemo podgraf, izomorfen grafu (ciklu) $C_{|V_G|}$, sledi, da G vsebuje Hamiltonov cikel. S tem smo problem Hamiltonovega cikla prevedli na problem podgrafnega izomorfizma in dokazali NP-polnost slednjega.

Pri izčrpnem iskanju rešitve problema izomorfnega podgrafa preverjamo vse možne preslikave vzorčnega grafa v ciljni graf. Med vzorčnim grafom G_p z n vozlišči in ciljnim grafom G_t z m vozlišči je v najslabšem primeru možnih kar $\binom{m}{n}n!$ preslikav. V algoritmih poskušamo iskalni prostor čim bolj in čim prej omejiti.

Poglavje 3

Algoritmi

3.1 Ullmannov algoritem

Najbolj znan algoritem za reševanje problema podgrafnega izomorfizma je *Ullmannov algoritem* [2], ki ga je leta 1976 predlagal Julian R. Ullmann. Algoritem predstavlja preiskovanje vzorčnega in ciljnega grafa v globino (angl. depth-first search - DFS), kjer vsaka pot preiskovanja predstavlja eno od možnih rešitev. Če na neki globini iskalnega drevesa ugotovimo, da trenutna pot ne vodi do rešitve problema, izvedemo *sestopanje* (angl. backtracking) do globine, na kateri je nadaljevanje iskanja spet smiselno. Ocena obetavnosti rešitve se izvede s pomočjo matrik sosednosti obeh grafov in drugih omejitev, na podlagi katerih odločimo, ali delno rešitev razvijamo naprej ali pa jo zavržemo. Ta algoritem je primeren tako za iskanje grafnih kot tudi (delnih ali induciranih) podgrafnih izomorfizmov [1], bodisi v usmerjenih ali neusmerjenih ter označenih ali neoznačenih grafih. V tem diplomskem delu se osredotočamo na iskanje induciranih podgrafnih izomorfizmov v usmerjenih neoznačenih grafih.

3.1.1 Opis algoritma

Ullmannov algoritem v našem primeru poišče vse pojavitve podgrafnega izomorfizma med vzorčnim grafom $G_p = (V_p, E_p)$ in ciljnim grafom $G_t = (V_t, E_t)$, saj je implementiran kot naštevalni algoritem. Vzorčni graf G_p vsebuje n vozlišč in p povezav, ciljni graf G_t pa m vozlišč in r povezav. Povezave med vozlišči teh dveh

grafov predstavimo v *matrikah sosednosti* (angl. adjacency matrix), kjer matrika $A = [a_{ij}]$ ustreza grafu G_p, matrika $B = [b_{ij}]$ pa grafu G_t. Če za primer vzamemo matriko A, predstavlja enica v i-ti vrstici in j-tem stolpcu povezavo med vozliščema i in j grafa G_p:

$$a_{ij} = \begin{cases} 1 & (i,j) \in E_p \\ 0 & \text{sicer.} \end{cases}$$

Podgrafni izomorfizem $f : V_p \to V_t$ je predstavljen v obliki dvojiške matrike M velikosti $n \times m$, imenovane *matrika združljivosti* (angl. compatibility matrix). Vrstice matrike M ustrezajo vozliščem $i \in V_p$, stolpci pa vozliščem $j \in V_t$. Element m_{ij} v vrstici i in stolpcu j matrike M je enak 1, če je vzorčno vozlišče i potencialni kandidat za preslikavo v ciljno vozlišče j:

$$m_{ij} = \begin{cases} 1 & f(i) = j \\ 0 & \text{sicer.} \end{cases}$$

Da matrika M predstavlja injektivno preslikavo, mora vsaka vrstica vsebovati natanko eno, vsak stolpec pa največ eno enico:

$$\forall i \in V_p : \sum_{j \in V_t} m_{ij} = 1 \quad in \quad \forall j \in V_t : \sum_{i \in V_p} m_{ij} \leq 1. \tag{3.1}$$

Matriko M, ki ustreza pogoju (3.1), lahko uporabimo za permutiranje vrstic in stolpcev matrike B, rezultat je matrika $C = [c_{ij}] = M(MB)^T$, kjer črka T predstavlja operacijo transponiranja. Če velja:

$$\forall i, j \in [1, n] : c_{ij} = a_{ij} \ [2], \tag{3.2}$$

potem matrika M predstavlja inducirani podgrafni izomorfizem iz G_p v G_t.

Očitno je, da vozlišča i ni možno preslikati v vozlišče j, če je stopnja vozlišča i večja od stopnje vozlišča j, zato začetno matriko $M^0 = [m^0_{ij}]$ napolnimo v skladu z naslednjim pravilom:

$$m^0_{ij} = \begin{cases} 1 & deg(i) \leq deg(j) \\ 0 & \text{sicer.} \end{cases} \tag{3.3}$$

Če imamo opravka z usmerjenimi grafi, mora pravilo (3.3) zadoščati tako v primeru vstopnih stopenj kot izstopnih stopenj vozlišč i in j.

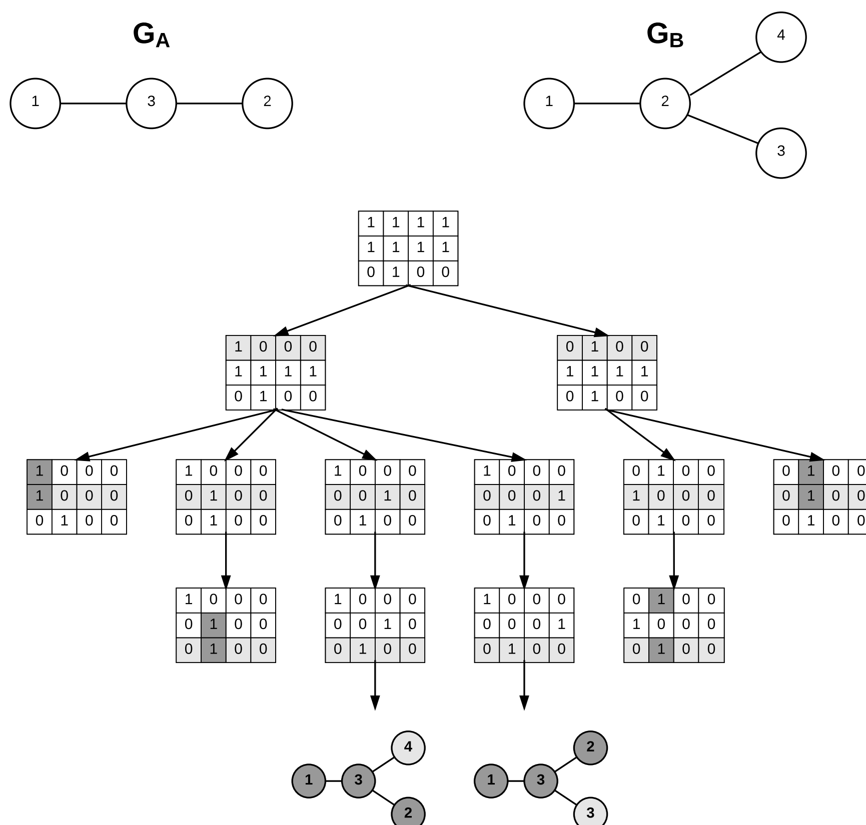
Algoritem deluje kot generator vseh možnih matrik M, ki zadoščajo pogoju, da za vsak element m_{ij} velja $(m_{ij} = 1) \Rightarrow (m^0_{ij} = 1)$, nato pa jih testira za izomorfizem z uporabo pogoja (3.2). Generiranje različnih matrik M poteka po korakih, ki sledijo inicializaciji matrike M^0. V teh korakih izbiramo še neobiskano vrstico, nato pa še neobiskan stolpec z vrednostjo 1 v izbrani vrstici. Vse enice v ostalih stolpcih spremenimo v ničle in nadaljujemo z naslednjim korakom, izbiro naslednje še neobiskane vrstice. Če v trenutni vrstici ni nobenega neobiskanega stolpca več, se generiranje vrne na prejšnji korak, kjer izberemo naslednji neobiskani stolpec. Ko ni več neobiskanih vrstic, imamo na globini n zgenerirane vse kandidate za podgrafni izomorfizem, nad katerimi preverimo pogoj (3.2).

Slika 3.1 prikazuje del postopka generiranja matrik M, ki se uporablja v Ullmannovem algoritmu.

Ullmann v članku [2] uporabi naslednje podatkovne strukture:

- spremenljivko d, ki predstavlja trenutno globino v iskalnem drevesu,
- spremenljivko k, ki hrani indeks zadnjega izbranega stolpca v trenutni vrstici,
- m-bitni dvojiški vektor $F = \{F_1, ..., F_m\}$, ki pove, kateri stolpci so bili do danega trenutka že izbrani,
- n-bitni vektor $H = \{H_1, ..., H_n\}$, ki pove, kateri stolpec je bil izbran na določeni globini; če je $H_d = k$, je bil k-ti stolpec izbran na globini d,
- vektor matrik $M_v = \{M_1, ..., M_n\}$, ki hrani zadnje generirane matrike na vsaki izmed globin iskalnega drevesa; M_i torej predstavlja zadnjo generirano matriko na globini i,
- trenutno matrika M.

Poenostavljena psevdokoda naštevalnega Ullmannovega algoritma iz [3] je navedena v Alg. 3.1. Ramovš je Ullmannov algoritem iz izvirnega članka podal v bolj pregledni obliki s tem, ko je preuredil različico iz [8].



Slika 3.1: Del postopka generiranja matrik M za preslikavo grafa G_A v graf G_B (brez prečiščevanja). Izhajamo iz matrike M^0, v kateri sistematično izbiramo še neobiskane vrstice in stolpce v teh vrsticah. Na ta način se na globini n (število n predstavlja število vozlišč grafa G_A) iskalnega drevesa zgenerirajo vse matrike, ki predstavljajo kandidate za podgrafni izomorfizem iz G_A v G_B. Te nato testiramo s pogojem (3.2). Vir: [1]

V vrstici 1 inicializiramo trenutno matriko M z matriko M^0, postavimo globino iskanja na 1, zapišemo, da ni bil še noben stolpec izbran na tej globini ($H_1 \leftarrow 0$) oz. v trenutni vrstici ($k \leftarrow 0$). Vrstica 2 postavi vse vrednosti vektorja F na 0, saj na začetku ni izbran še noben stolpec. Pred prvim korakom shranimo začetno matriko (vrstica 3) v vektor matrik M_v. Vstop v zanko je v četrti vrstici, v njej pa

ostanemo, dokler drevo iskanja ni v celoti pregledano. To se zgodi, ko v prvi vrstici matrike M_1 ni več neobiskanih stolpcev in dobi spremenljivka d vrednost 0. V peti vrstici zagotovimo, da se izvede sestopanje v primeru, če ni izpolnjen pogoj, ki ga preverjamo v vrstici 6. Ta išče stolpec, ki je še neizbran v trenutni vrstici ($j > k$), je s to vrstico združljiv ($m_{dj} = 1$) in ni bil izbran že v kateri od prejšnjih vrstic ($F_j = 0$), saj moramo zagotoviti injektivnost preslikave. V vrstici 7 poskrbimo, da se sestopanje onemogoči, ker bomo v naslednji iteraciji zanke ostali na isti globini ali pa bomo globino povečali. V 8. vrstici izbrani stolpec označimo v matriki M,

Algoritem 3.1 Ullmannov algoritem

Vhod: začetna matrika M^0 ter matriki sosednosti A in B

Izhod: vse matrike M dimenzije $n \times m$, ki predstavljajo podgrafni izomorfizem

1: $M \leftarrow M^0$; $d \leftarrow 1$; $H_1 \leftarrow 0$; $k \leftarrow 0$; $backtrack \leftarrow true$;
2: **for** $i \in [1, n]$ **do** $F_i \leftarrow 0$;
3: $M_1 \leftarrow M^0$;
4: **while** $d \neq 0$ **do**
5: $backtrack \leftarrow true$;
6: **if** $(\exists j : j > k \wedge m_{dj} = 1 \wedge F_j = 0)$ **then**
7: $backtrack \leftarrow false$;
8: $\forall j \neq k : m_{dj} \leftarrow 0$;
9: **if** REFINE(M, A, B) **then**
10: **if** $d < n$ **then**
11: $H_d \leftarrow k$; $F_k \leftarrow 1$; $d \leftarrow d + 1$; $k \leftarrow 0$; $M_d \leftarrow M$;
12: **else**
13: **if** $\langle$condition(3.2)$\rangle$ **then** STORE(M);
14: $M \leftarrow M_d$;
15: **else**
16: $M \leftarrow M_d$;
17: **if** $backtrack$ **then**
18: $F_k \leftarrow 0$; $d \leftarrow d - 1$;
19: **if** $d > 0$ **then**
20: $M \leftarrow M_d; k \leftarrow H_d$;

tako da preostale stolpce postavimo na 0. Na vrstico 9 se bomo vrnili v nadaljevanju, ko bomo podrobneje obravnavali enega izmed ključnih delov algoritma. Sledi preverjanje globine iskanja: če smo na globini n, preverimo, če matrika predstavlja

podgrafni izomorfizem. Če je torej pogoju (3.2) zadoščeno (vrstica 13), matriko M shranimo in jo obnovimo (vrstica 14) za morebitne preostale neizbrane stolpce v trenutni vrstici. V primeru, da z iskanjem še nismo na globini n, pa je potrebno v vrstici 10 shraniti zadnji izbrani stolpec na trenutni globini ($H_d \leftarrow k$), ga označiti kot že izbranega ($F_k \leftarrow 1$), povečati globino iskanja ($d \leftarrow d+1$), ponastaviti spremenljivko za izbiro stolpca ($k \leftarrow 0$) in pred naslednjim korakom shraniti matriko M ($M_d \leftarrow M$). Tudi vrstico 16 bomo natančneje opisali v naslednjem podpoglavju, zato jo zaenkrat ignoriramo. V vrstici 18 se začne sestopanje, če le-to ni bilo v vrstici 7 preprečeno. Sestop začnemo z razveljavitvijo zadnje izbire stolpca ($F_k \leftarrow 0$) in zmanjšanjem globine za 1 ($d \leftarrow d-1$). Če sestopimo vse do korena iskalnega drevesa, se algoritem zaključi, v nasprotnem primeru ($d > 0$) pa v vrstici 20 obnovimo matriko M za iskanje naslednjega primernega stolpca na globini sestopa ($M \leftarrow M_d$) in v spremenljivko k zapišemo zadnji izbrani stolpec na tej globini.

3.1.2 Ullmannov algoritem s postopkom prečiščevanja

Pri iskanju podgrafnega izomorfizma z generiranjem vseh možnih matrik M, ki izhajajo iz matrike M^0, ugotovimo, da je za rešitev potrebno relativno veliko računanja. S tem namenom uvedemo postopek *prečiščevanja* (angl. refinement procedure), ki s spreminjanjem enic v ničle v matriki M dodatno omeji iskalno drevo.

Rešitev za zmanjšanje obsega računanja temelji na dejstvu, da se vozlišče i lahko preslika v vozlišče j samo v primeru, če se tudi vsak sosed vozlišča i preslika v enega izmed sosedov vozlišča j:

$$\forall u \in N_{G_p}(i) \exists v \in N_{G_t}(j) : m_{ij} = 1. \quad (3.4)$$

Pri prečiščevanju torej preverjamo pogoj (3.4) za vsak i, j, kjer $m_{ij} = 1$. Če je pogoju zadoščeno, se vrednost m_{ij} ne spremeni, v nasprotnem primeru pa se postavi na 0. Ker lahko sprememba vrednosti elementa v matriki M vpliva na preverjanje pogoja (3.4) na nekam drugem elementu v tej matriki, postopek ponavljamo toliko časa, da v celotni matriki ni več nobenega elementa, kateremu bi lahko spremenili vrednost iz 1 v 0. To pa je tudi potreben in hkrati zadosten pogoj za preverjanje

obstoja podgrafnega izomorfizma, zato lahko z njim zamenjamo pogoj (3.2).

Med samim postopkom vseskozi tudi preverjamo, če vsaka vrstica v matriki M vsebuje vsaj eno 1. Če za katero izmed njih to ne drži, se prečiščevanje neuspešno konča, saj nadaljevanje nima smisla, ker nobene 0 ne moremo spremeniti v 1. S tem pa je pogoj injektivnosti preslikave (3.1) neizpolnjen, zato takšna matrika ne more predstavljati podgrafnega izomorfizma.

Psevdokoda postopka prečiščevanja, ki je kot procedura REFINE uporabljen v Alg. 3.1 (vrstica 9), je zapisana v razdelku Alg. 3.2. Ta procedura vsebuje zanko (vrstice 2-9), ki za vsak element matrike M z vrednostjo 1 (vrstica 4) preverja pogoj (3.4) (vrstica 5). Če pogoju ni zadoščeno, v vrstici 6 postavimo element m_{ij} na 0, saj se vozlišče i ne more preslikati v vozlišče j. Ker se je s tem vsebina matrike M spremenila, je potrebno prečiščevanje ponoviti ($fixpoint \leftarrow false$). Vedno, ko v matriki spremenimo 1 v 0, obstaja možnost, da v trenutni vrstici ni več nobene enice. Če se to res zgodi, procedura REFINE vrne vrednost $false$ (vrstica 8), to pa posledično pomeni, da smo v iskalnem drevesu prišli do matrike, katera se ne more več razviti v podgrafni izomorfizem. Zato to matriko završemo, poleg tega pa moramo poskrbeti, da za naslednji izbrani stolpec razveljavimo spremembe v matriki M, povzročene v proceduri REFINE (vrstica 16 v Alg. 3.1). V primeru, da procedura prečiščevanje konča brez vrstice samih ničel, vrne vrednost true (vrstica 10).

Algoritem 3.2 Postopek prečiščevanja

Vhod: matrika M ter matriki sosednosti A in B

Izhod: *true*, če je bilo število enic v matriki M zmanjšano, $false$, če v kateri izmed vrstic ni nobene enice

```
1: procedure REFINE(M, A, B)
2:     repeat
3:         fixpoint ← true
4:         for ∀(i, j) : m_ij = 1 do
5:             if !condition (3.1)  then
6:                 m_i,j ← 0; fixpoint ← false;
7:                 if ∀k : m_ik = 0 then
8:                     return false
9:     until fixpoint
10:    return true
```

3.1.3 Zahtevnost algoritma

Časovna zahtevnost procedure REFINE je $O(nm\Delta_p\Delta_t d_p)$, kjer sta Δ_p in Δ_t največji stopnji v vzorčnem oziroma ciljnem grafu, d_p pa premer vzorčnega grafa. Zahtevnost celotnega Ullmannovega algoritma je v najslabšem primeru eksponentna. Ta se pojavi v primeru, kadar je vzorčni graf klika ciljnega grafa. Število podgrafnih izomorfizmov med grafoma je tedaj vsaj toliksno, kolikor je permutacij vozlišč vzorčnega grafa.

Vektor M_v v algoritmu shranjuje n matrik velikosti $n \times m$, saj na vsaki globini iskalnega drevesa shranimo po eno matriko. Prostorska zahtevnost celotnega algoritma je torej $O(n^2m)$.

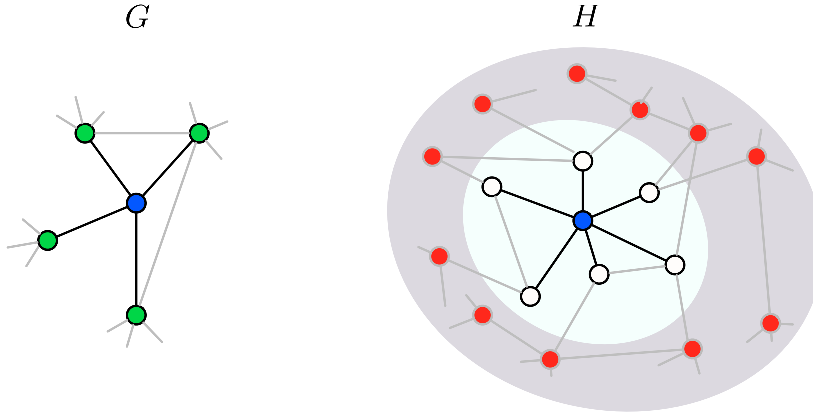
3.2 Izboljšani Ullmannov algoritem

Čibej in Mihelič sta leta 2012 v [9, 10] predstavila nekaj izboljšav Ullmannovega algoritma, katera zmanjšajo tako časovno kot tudi prostorsko zahtevnost. Čeprav se Ullmannov algoritem še vedno precej pogosto uporablja, se po učinkovitosti ne more več kosati z novejšimi algoritmi kot so FocusSearch, VF ali VF2. Predlagane izboljšave se osredotočajo predvsem na čimprejšnjo eliminacijo neobetavnih poti v iskalnem drevesu in zmanjšanje pomnilniškega prostora, potrebnega za izvajanje algoritma. V nadaljevanju so ti predlogi tudi natančneje opisani.

3.2.1 Filtriranje soseščine

V originalnem Ullmannovemu algoritmu s postopkom filtriranja matrike M z izbranim parom vzorčnega in ciljnega vozlišča omejimo prostor iskanja rešitve. Izbira elementa m_{ij} pomeni, da se vozlišče i lahko preslika samo v vozlišče j ter obratno, da je vozlišče j lahko slika edino vozlišča i. Sledi, da lahko vse ostale elemente v i-ti vrstici in j-temu stolpcu postavimo na 0. Predlagana izboljšava filtriranja pa vpliva na vso soseščino izbranega para vozlišč, ne le na vozlišči sami.

Na sliki 3.2 je prikazana izbira para vzorčnega in ciljnega vozlišča. Očitno je, da se sosedi vozlišča vzorčnega grafa lahko preslikajo samo v sosede vozlišča ciljnega grafa. Iz te ugotovitve pa sledi, da so sosedi izbranega vozlišča vzorčnega grafa nezdružljivi z vozlišči ciljnega grafa, ki niso sosedi izbranega ciljnega vozlišča.



Slika 3.2: Izboljšano filtriranje na podlagi izbranih vozlišč (modra barva) vzorčnega grafa G in ciljnega grafa H. Sosedi izbranega vozlišča v G (zelena barva) niso združljivi z vozlišči, ki niso sosedi izbranega vozlišča v H (rdeča barva). Vir: [10]

Če sta torej izbrana vzorčno vozlišče i in ciljno vozlišče j, velja:

$$\forall u \in N_{G_p}(i) \forall v \notin N_{G_t}(j) : m_{uv} = 0. \tag{3.5}$$

V primeru iskanja induciranega podgrafnega izomorfizma lahko pogoj (3.5) preverjamo tudi v obratni smeri:

$$\forall v \in N_{G_t}(j) \forall u \notin N_{G_p}(i) : m_{uv} = 0. \tag{3.6}$$

3.2.2 Prilagoditev postopka prečiščevanja

S postopkom prečiščevanja matrike združljivosti M v originalnem Ullmannovemu algoritmu pregledamo vse elemente $m_{ij} = 1$ in jih poskušamo spremeniti v 0. Končamo šele takrat, ko se v enem pregledu skozi matriko ne pojavi nobena sprememba elementa iz 1 v 0. Ker je ta postopek precej časovno zahteven ($O(d_{G_p} nm \Delta_{G_p} \Delta_{G_t})$), ga poskušamo pohitriti.

Z namenom ocenitve učinkovitosti postopka prečiščevanja je bil izveden preizkus, v katerem so se na vsaki globini iskanja štele spremembe elementov matrike

M iz 1 v 0, prav tako pa tudi klici procedure, ki izvede prečiščevanje. Rezultati so pokazali, da je postopek učinkovit le na začetni matriki M_0 ter na prvih nekaj globinah iskanja rešitve, nasprotno pa je zelo neučinkovit na večjih globinah, še posebej v kombinaciji s filtriranjem. Ob eksponentni rasti števila klicev procedure se v matriki pojavi le nekaj ali celo nič redukcij enic. Izboljšan način filtriranja, ki je opisan v prejšnjem podpoglavju, skoraj odpravi potrebo po prečiščevanju celotne matrike združljivosti.

Na podlagi teh ugotovitev uvedemo postopek *delnega prečiščevanja* (angl. partial refinement), s katerim lokaliziramo prečiščevanje matrike na izbrani vozlišči vzorčnega in ciljnega grafa ter na njuni soseščini. Pogoj

$$\forall u \in N_{G_p}(i) \exists v \in N_{G_t}(j) : m_{uv} = 1 \tag{3.7}$$

torej preverjamo le na parih $N(vrstica) \times N(stolpec)$, zato se tudi zahtevnost postopka prečiščevanja zmanjša na $O(d_{G_p} \Delta^2_{G_p} \Delta^2_{G_t})$.

V nadaljevanju bomo videli, da je delno prečiščevanje bolj učinkovito v algoritmu, ki vrstni red obdelave vozlišč vzorčnega grafa tudi topološko uredi. V izboljšanemu Ullmannovemu algoritmu polno prečiščevanje izvedemo le na začetni matriki M^0, na sledečih globinah iskanja pa uporabimo delno prečiščevanje.

Ostane nam samo še odločitev o tem, kolikokrat naj se delna prilagoditev izvede, da bo še dovolj učinkovita. V primerjavi s polno prilagoditvijo sedaj naredimo manj redukcij enic. Eksperimentalni rezultati so pokazali, da postopek delne prilagoditve postane neučinkovit nekje med tretjino in polovico največje globine iskanja [9, 10].

3.2.3 Razvrščanje vzorčnih vozlišč

V dosedanjem opisu algoritma še vedno ni znano, katera vrstica je izbrana na določeni globini iskanja, saj način, na katerega bi določili vrstni red obdelave vozlišč, ni predpisan. To sicer ne vpliva na pravilnost algoritma, lahko pa ima učinek na (ne)učinkovitost le-tega. Četudi je nek par vozlišč vzorčnih in ciljnih grafov med seboj kompatibilen, še ne pomeni, da bo ta izbira kasneje pripeljala do podgrafnega izomorfizma. Žal pa je slabo izbiro para vozlišč v zgodnji fazi iskanja rešitve težko zaznati, kar vodi v nepotrebno preiskovanje neperspektivnih vej iskalnega

drevesa.

Da bi se tovrstne veje iskanja odkrile v čimbolj zgodnji fazi, uporabimo hevristiko za razvrščanje vozlišč vzorčnega grafa. Vozlišče z zaporedno številko i se torej preslika na i-ti globini iskanja. Omenimo še, da je vrstni red statičen v smislu, da je določen že pred začetkom iskanja in se med iskanjem ne spreminja.

Ullmann je predlagal, da bi algoritem najprej obdelal vozlišča z najvišjo stopnjo. Na prvi pogled je rešitev res optimalna, saj imajo vzorčna vozlišča z višjo stopnjo manj združljivih vozlišč v ciljnem grafu. Eden izmed pomislekov te hevristike pa se pojavi pri regularnih grafih, saj imajo vozlišča v tem primeru vsa isto stopnjo in je naključni vrstni red izbire povsem enako učinkovit. Metoda tudi ne upošteva lokalnosti, saj z izbiro para vozlišč prispevamo k dodatni omejitvi iskanja preslikave njunih sosedov.

Sledi nekaj zapisov formalnih definicij. Razvrstitev vozlišč je bijektivna preslikava $\pi : V_p \to \{1, 2, ..., n\}$. Definiramo tudi množico vozlišč, ki so na vrsti za obdelavo pred i-tim vozliščem:

$$\Pi^i_< = \{u|\pi(u) < i\}. \tag{3.8}$$

V [9, 10] so analizirani različni načini obiskovanja vozlišč, ki upoštevajo topološke lastnosti grafa. Kot smo omenili že v prejšnjem razdelku, postopek delnega prečiščevanja deluje v soseščini izbranega vozlišča in je učinkovit le v primeru, ko razvrščanje vozlišč to lokalnost upošteva.

Alg. 3.3 prikazuje psevdokodo, ki služi kot ogrodje za različne hevristike urejanja. Kot vhodni parameter podamo vzorčni graf, izhod pa predstavlja urejena množica vozlišč tega grafa.

V vrstici 1 inicializiramo števec vozlišč na 1. V zanko vstopimo v vrstici 2 in v njej ostanemo, dokler ni razvrščena celotna množica vozlišč. V vrstici 3 v množico C dodamo vsa vozlišča, ki še niso razvrščena. V naslednji vrstici med elementi množice C poiščemo vozlišče, katero glede na izbrano hevristiko najbolj ustreza. To vozlišče v 5. vrstici tudi postavimo na i-to mesto. Pred koncem obhoda zanke v vrstici 6 povečamo števec i za 1.

Pri testiranju uspešnosti hevristik je bilo uporabljeno 16 le-teh, katere so kombinacije štirih različnih pristopov razvrščanja vozlišč in štirih kriterijev $f(u)$, po katerih se izmed elementov množice C določi najprimernejše vozlišče za i-to mesto

Algoritem 3.3 Ogrodje za različne hevristike razvrščanja vozlišč vzorčnega grafa

Vhod: vzorčni graf $G_p = (V_p, E_p)$
Izhod: urejena množica vozlišč grafa G_p

1: $i = 1$;
2: **while** $i \leq n$ **do**
3: $C = izbrani\ kandidati\ iz\ V \setminus \Pi^i_<$;
4: $v = argmax_{u \in C} f(u)$;
5: $\pi(v) = i$;
6: $i = i + 1$;

v razvrstitvi.

Na podlagi rezultatov eksperimenta sta avtorja predlagala naslednja pristopa in kriterij:

- pristop 1 (oznaka DEPTH): prvo vozlišče je izbrano poljubno (ponavadi tisto z najvišjo stopnjo), vsa naslednja pa so izbrana po strategiji *BFS*. Množica C torej vsebuje vozlišča, ki so na istem nivoju BFS drevesa:

$$C = \{v \mid d(s, v) = \min_{u \in V \setminus \Pi^i_<} d(s, u)\}. \tag{3.9}$$

- pristop 2 (oznaka NEIGH): podobno kot po pristopu DEPTH je tudi tu prvo vozlišče izbrano poljubno, vsa naslednja pa so izbrana iz soseščine že razvrščenih vozlišč:

$$C = N_{G_p}(\Pi^i_<). \tag{3.10}$$

- kriterij (oznaka DEG): izbrano vozlišče z najvišjo stopnjo:

$$f(u) = \delta(u). \tag{3.11}$$

Z izbiro zgornjih pristopov in kriterija dobimo dve kombinaciji razvrščanja vozlišč, DEPTH-DEG in NEIGH-DEG [11].

Na podlagi opazovanja časa izvajanja Ullmannovega algoritma se izkaže, da v nobenem primeru razvrščanje vozlišč ni imelo znatnega učinka na hitrost izvedbe, saj postopek prečiščevanja deluje globalno, s tem pa lokalnost razvrščanja postane praktično nepomembna. Prav nasprotno pa razvrščanje učinkuje na algoritem

z delnim prečiščevanjem, kakršna je predlagana v podpoglavju 3.2.2, saj njeno izvedbo opazno pospeši.

3.2.4 Zmanjšanje prostorske zahtevnosti

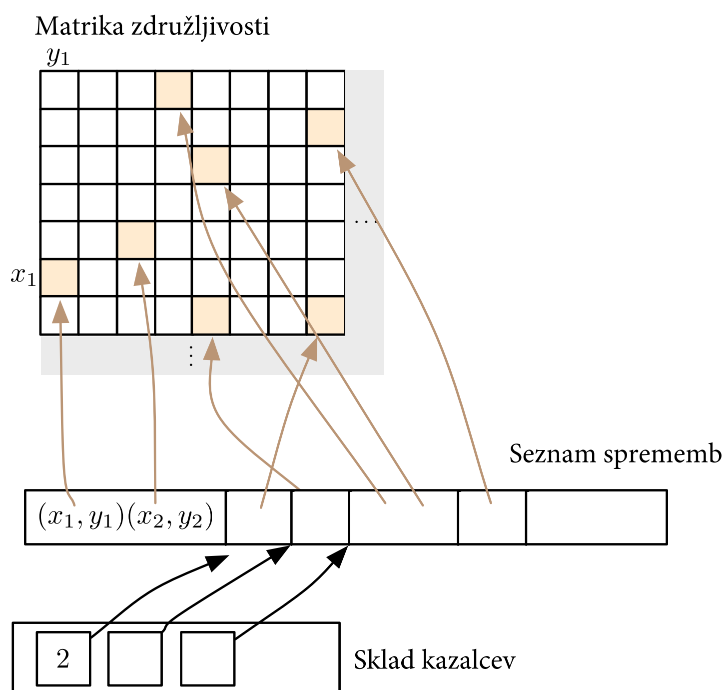
Prostorska zahtevnost Ullmannovega algoritma je $O(n^2m)$, kar je ena njegovih glavnih pomanjkljivosti. Na vsakem koraku iskanja rešitve je v matriki združljivosti dovoljeno enico spremeniti v ničlo, obratno pa to ne velja. Sledi, da je največje število sprememb nm. Z namenom prihraniti prostor v pomnilniku in obenem omogočiti sestopanje, namesto shranjevanja celotne matrike shranimo samo zgodovino sprememb v njej.

Za implementacijo te rešitve poleg matrike združljivosti M potrebujemo še dodatni podatkovni strukturi. Prvi je seznam sprememb največje velikosti nm, ki vsebuje koordinate elementov iz matrike M, kateri so bili spremenjeni iz 1 v 0. Druga struktura predstavlja sklad kazalcev, ki kažejo v seznam sprememb. Največja velikost sklada je n. Kazalec predstavlja mejo med dvema različicama matrike M. Grafična predstavitev rešitve je prikazana na sliki 3.3.

V Ullmannovemu algoritmu je bilo potrebno spremembo le zabeležiti v matriki M, z izboljšano rešitvijo pa dodatno shranjujemo spremembe še v za to namenjen seznam. Ko izvedemo korak naprej v iskalnem drevesu, s funkcijo PUSH na sklad shranimo kazalec na zadnji vnos v seznamu sprememb. Ta funkcija v našem primeru nadomesti shranjevanje celotne matrike za primer sestopanja, katerega izvedemo s klicem POP. S tem se razveljavijo vse spremembe, narejene na globini, iz katere sestopamo.

Opisana izboljšava pa ne pomeni samo prihranek pomnilnika, ampak tudi pozitivno vpliva na hitrost algoritma. Sedaj pri koraku naprej v iskalnem drevesu ni potrebno narediti varnostne kopije celotne matrike M, ampak samo seznam sprememb, kar nam prihrani čas. Negativna stran rešitve pa se pokaže pri sestopanju. Namesto da bi matriko M enostavno zavrgli, jo je potrebno na podlagi seznama sprememb ustrezno popraviti, kar pomeni počasnejše sestopanje.

Medtem ko je pohitritev algoritma zanemarljiva, pa je prihranek pomnilnika občuten. Za primer vzemimo vzorčni graf s 600 vozlišči in ciljni graf s 1000 vozlišči. Brez predlagane izboljšave je poraba pomnilnika okrog 500 MB, z njo pa le približno 1 MB.



Slika 3.3: Zmanjšanje prostorske zahtevnosti algoritma z uvedbo seznama sprememb in sklada kazalcev v seznam sprememb. Vir: [10]

3.3 Algoritem RI

Skupen cilj sodobnih algoritmov za reševanje problema podgrafnega izomorfizma je omejiti prostor iskanja rešitve oziroma izločiti neobetavne delne rešitve karseda hitro in po čimnižji ceni. Eden izmed poskusov realizacije te sposobnosti je tudi algoritem, imenovan *RI*, ki je bil leta 2013 predstavljen v [12]. Poglavitni del algoritma predstavlja vrstni red obiskovanja vozlišč vzorčnega grafa, kateri se tvori na način, da zadosti točno določenim topološkim kriterijem. Strategija iskanja rešitve je neodvisna od ciljnega grafa in obenem statična, saj se vrstni red obiskovanja vozlišč določi pred samim začetkom iskanja in se vse do konca ne spreminja.

3.3.1 Statična strategija iskanja

Vsako vozlišče v iskalnem drevesu predstavlja možno preslikavo vozlišča u grafa G_p v vozlišče v grafa G_t. Naj bo $f : V_p \rightarrow V_t$ preslikava vozlišč vzorčnega grafa v vozlišča ciljnega grafa, n število vozlišč vzorčnega grafa in

$$p_t = ((u_0, f(u_0)), (u_1, f(u_1)), ..., (u_x, f(u_x))) \tag{3.12}$$

pot po iskalnem drevesu z začetkom v korenu drevesa. Če je $x < n-1$, je p_t delna, v primeru $x = n-1$ pa celotna rešitev.

Pred začetkom iskanja rešitve določimo vrstni red obiskovanja vzorčnih vozlišč, tako da zavržemo neobetavne delne rešitve čim hitreje. To zaporedje zapišemo z $\mu = (u_0, u_1, \ldots, u_{n-1})$. Prvo vozlišče v zaporedju je tisto z najvišjo stopnjo, vsa naslednja pa so v μ umeščena na podlagi rezultata *funkcije vrednotenja* (angl. scoring function). Za vsako nerazvrščeno vozlišče v hranimo njegovega starša, tj. vozlišče u_i v zaporedju μ z najmanjšim indeksom i, za katerega velja $(u_i, v) \in E$. Psevdokoda algoritma razvrščanja, imenovanega *GreatestConstraintFirst* (katerega uporablja RI), je zapisana v Alg. 3.4.

Naj bo m naslednji korak pri razvrščanju vozlišč in $\mu = (u_0, u_1, ..., u_{m-1})$ zaporednje že razvrščenih vozlišč. Z u_m označimo kandidata za umestitev v zaporedje na m-tem koraku (m-to mesto). Funkcija vrednotenja izbere najprimernejše vozlišče za m-to mesto s pomočjo naslednjih treh množic (vrstice 10 – 16 v Alg. 3.4):

- $V_{m,vis} = \{u_i : 0 \leq i < m : \langle u_m, u_i \rangle \in E\}$ - množica vozlišč iz μ, ki so sosedi u_m,
- $V_{m,neig} = \{u_i : 0 \leq i < m, \exists j > m : (u_i, u_j) \in E, (u_m, u_j) \in E\}$ - množica vozlišč iz μ, katera so sosednja vsaj enemu vozlišču, ki ni v μ in je povezan z u_m,
- $V_{m,unv} = \{u_j : j > m, (u_m, u_j) \in E, \forall i < m : (u_i, u_j) \notin E\}$ – množica vozlišč, ki niso v μ, niti niso sosedi vozlišč v μ, so pa sosedi u_m.

Največjo težo pri vrednotenju kandidata u_m ima $|V_{m,vis}|$, manjšo $|V_{m,neig}|$, najmanjšo pa $|V_{m,unv}|$. Če v nekem trenutku izbiramo med kandidatoma u_a in u_b, je izbran u_a v naslednjih primerih:

1. $|V_{a,vis}| > |V_{b,vis}|$,
2. $|V_{a,vis}| = |V_{b,vis}|$ in $|V_{a,neig}| > |V_{b,neig}|$,
3. $|V_{a,vis}| = |V_{b,vis}|$, $|V_{a,neig}| = |V_{b,neig}|$ in $|V_{a,unv}| > |V_{b,unv}|$.

Če imata kandidata enak rezultat vrednotenja, se izbere poljubnega izmed njiju, neizbranega pa začasno shranimo.

Slika 3.4 prikazuje primer razvrščanja vzorčnih vozlišč v algoritmu RI. Prvo je izbrano vozlišče 4, ker ima največjo stopnjo. Naj bo v naslednjem koraku $\mu = \{4, 1\}$. Kandidati za tretje mesto v razvrstitvi so sedaj vozlišča 0, 2, 6, 5 in 7.

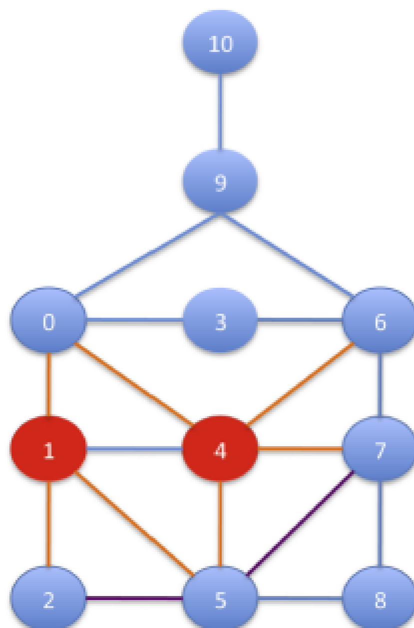
Algoritem 3.4 Algoritem razvrščanja vozlišč vzorčnega grafa GreatestConstraintFirst

Vhod: vzorčni graf $G_p = (V_p, E_p)$, $n = |V_p|$

Izhod: urejen seznam vzorčnih vozlišč $\mu = (u_0, u_1, ..., u_{n-1})$, urejen seznam staršev vzorčnih vozlišč $pt_\mu = (pt(u_0), pt(u_1), ...pt(u_n))$

1: poišči u_0 (vozlišče z najvišjo stopnjo);
2: $V = V_p \setminus \{u_0\}$;
3: $\mu = (u_0)$;
4: $pt(u_0) = null$;
5: $pt_\mu = (pt(u_0))$;
6: $u_m = null$;
7: $u_{rank} = (-\infty, -\infty, -\infty)$;
8: **while** $V \neq 0$ **do**
9:     $m = |\mu|$;
10:     **for all** $u \in V \setminus \mu$ **do**
11:         $V_{u,vis} = \{u_i : 0 \leq i < m : \langle u, u_i \rangle \in E\}$;
12:         $V_{u,neig} = \{u_i : 0 \leq i < m, \exists v \neq u \in V : \langle u_i, v \rangle \in E, \langle u, v \rangle \in E\}$;
13:         $V_{u,unv} = \{v : v \neq u, \langle u, v \rangle \in E, \forall i < m : \langle u_i, v \rangle \notin E\}$;
14:         **if** $u_{rank} \leq (|V_{u,vis}|, |V_{u,neig}|, |V_{u,unv}|)$ **then**
15:             $u_m = u$;
16:             $u_{rank} = (|V_{u,vis}|, |V_{u,neig}|, |V_{u,unv}|)$;
17:     najdi najmanjši i, da velja $u_i \in \mu : \langle u_m, u_i \rangle \in E$;
18:     $pt(u_m) = u_i$;
19:     dodaj u_m v μ;
20:     dodaj $pt(u_m)$ v pt_μ;
21:     $V = V \setminus \{u_m\}$;
22: **return** μ in pt_μ;

Naslednje vozlišče, izbrano v zaporedje, je 5. Čeprav je po kriteriju 1 izenačen z vozliščem 0, ima vozlišče 5 po kriteriju 2 višji rezultat vrednotenja. Po kriteriju 3 ima vozlišče 0 sicer višji rezultat vrednotenja, vendar ima ta kriterij manjšo težo, zato je izbrano vozlišče 5.



Slika 3.4: Primer razvrščanja vozlišč v algoritmu RI. Vir: [12]

Velja omeniti, da opisana strategija razvščanja ne favorizira vozlišča iz gostejšega dela grafa. Ni torej nujno, da so ta vozlišča v začetku zaporedja. Prav tako središčna vozlišča grafa pri razvrščanju nimajo nobene prednosti pred ostalimi vozlišči. V primeru, predstavljenem na sliki 3.4, središčni vozlišči 0 in 6 sploh nista uvrščeni na začetek zaporedja.

3.3.2 Omejevanje prostora iskanja

Vsako vozlišče iskalnega drevesa predstavlja mapiranje vozlišča vzorčnega grafa v vozlišče ciljnega grafa. Pri iskanju rešitev je potrebno za vsak par $i, j : i \in V_p, j \in V_t$ preveriti, če zadosti pogojem za obstoj podgrafnega izomorfizma.

Na vsakem koraku (globini) i preiskovanja drevesa določimo vozlišča ciljnega grafa, ki so kandidati za preslikavo vozlišča u_i vzorčnega grafa, med sosedi slik

staršev u_i. V ta namen v Alg. 3.4 tvorimo tudi urejen seznam staršev vozlišč vzorca.

Preverjanje naslednjih pogojev izloča neobetavne poti (delne rešitve):

1. tako u_i kot tudi kandidat za $f(u_i)$ na trenutni poti še nista mapirana,

2. stopnja kandidata za $f(u_i)$ je večja ali enaka stopnji vozlišča u_i:

$$|\{(v, f(u_i)) \in E_t\}| \geq |\{(u, u_i) \in E_p\}| \wedge \\ |\{(f(u_i, v) \in E_t\}| \geq |\{(u_i, u) \in E_p\}|, \tag{3.13}$$

   kjer je u poljubno vozlišče vzorčnega grafa, v pa poljubno vozlišče ciljnega grafa,

3. na trenutni poti do trenutnega vozlišča iskalnega drevesa veljajo omejitve, ki izhajajo iz topologije vzorčnega grafa:

$$\forall u_i, u_j \in V_p : 0 \leq j \leq i, (u_i, u_j) \in E_p \Rightarrow (f(u_i), f(u_j)) \in E_t. \tag{3.14}$$

Algoritem 3.5 Iskanje podgrafnih izomorfizmov v algoritmu RI

Vhod: vzorčni graf $G_p = (V_p, E_p)$, ciljni graf $G_t = (V_t, E_t)$, $n = |V_p|$, urejen seznam vzorčnih vozlišč $\mu = (u_0, u_1, ..., u_{n-1})$, urejen seznam staršev vzorčnih vozlišč $pt_\mu = (pt(u_0), pt(u_1), ..., pt(u_{n-1}))$

Izhod: število podgrafnih izomorfizmov med grafoma G_p in G_t

1: **for all** pot p_j **do**
2:     **for** $i = 0$ **to** $n - 1$ **do**
3:         najdi vozlišče $x \in V_t \wedge x \notin p_j$ med sosedi slike vozlišča $pt(u_i)$;
4:         **if** x zadosti pogojem za obstoj izomorfizma **then**
5:             dodaj ujemanje (u_i, x) v pot p_j;
6:         **if** ne obstaja ustrezen x **then**
7:             zavrzi pot p_j;
8:     **if** $i = n$ **then**
9:         povečaj števec najdenih podgrafnih izomorfizmov;
10: **return** število najdenih podgrafnih izomorfizmov med grafoma G_p in G_t;

Le v primeru, ko je $(i-1)$–temu pogoju zadoščeno, se preveri tudi i-ti pogoj. Pogoja 1 in 3 zagotavljata izomorfizem, medtem ko ima 2. pogoj vlogo filtriranja kandidatov za sliko posameznega vzorčnega vozlišča. Pravkar naveden postopek mapiranja je prikazan v Alg. 3.5.

Algoritem RI omejuje iskalni prostor zgolj s preverjanjem pogojev podgrafnega izomorfizma brez zahtevnih postopkov, kakršno je npr. filtriranje. Z upoštevanjem ciljnega grafa pri omejevanju bi bilo le-to še bolj učinkovito, vendar tudi (časovno) zahtevnejše.

Poglavje 4

Integracija v sistem ALGator

4.1 ALGator

ALGator [15] je sistem za izvajanje algoritmov na danih testnih podatkih in analizo rezultatov izvajanja. Uporabniku (razvijalcu algoritmov) omogoča dodajanje večjega števila različnih algoritmov za poljuben problem. V okviru projekta je potrebno definirati problem, testne množice vhodnih podatkov in način reševanja danega problema. Na podlagi izhodnih podatkov izvajanja algoritmov istega projekta lahko le-te med seboj primerjamo in ocenjujemo njihovo učinkovitost na testnih množicah z različnimi lastnostmi.

4.2 Delovanje sistema

Sistem ALGator za svoje delovanje uporablja korenski imenik <ALGator_data_root> in njegove podimenike, v katerih so konfiguracijske datoteke tipa JSON ali CSV, javanski razredi z opisom projekta in posameznih algoritmov ter druge datoteke, potrebne za izvedbo testov (npr. vhodni podatki). V tem datotečnem drevesu se v času testiranja generirajo tudi datoteke z eksperimentalnimi rezultati, katere lahko kasneje podrobneje analiziramo. Slika 4.1 prikazuje primer imenika <ALGator_data_root>/projects, kateri vsebuje definicije problemov v sistemu.

```
PROJ-ProjName             // project root folder
  proj           // folder for all project files
    ProjName.atp          // project configuration file
    ProjName.atrd         // result description file
    src                   // folder for project source files
      [Project]AbsAlgorithm.java
    [Project]TestSetIterator.java
    [Project]TestCase.java
    bin                   // compiled classes of the project
      [Project]AbsAlgorithm.class
    [Project]TestSetIterator.class
    [Project]TestCase.class
  algs                    // folder for algorithms
    ALG-AlgName       // algorithm root folder
      AlgName.atal        // algorithm configuration file
      src             // algorithm source files
      [Algorithm][Project]AbsAlgorithm.java
      bin             // compiled algorithm classes
        [Algorithm][Project]AbsAlgorithm.class

  tests             // project-specific test files folder
    TestSetName1.atts     // one or more testset files
  results                 // folder with results (attr files)
  queries             // folder for predefined queries
```

Slika 4.1: Primer imenika <ALGator_data_root>/projects. Vir: [15]

4.3 Opredelitev problema

Administrator projekta v imenik <ALGator_data_root>/projects/PROJ-<project_name>/proj doda dve konfiguracijski datoteki. Prva vsebuje splošne informacije o projektu, kot so opis, avtor ter imena algoritmov za reševanje problema, testnih množic, javanskih datotek algoritma in opcijskih datotek JAR. V drugi konfiguracijski datoteki opišemo želeno obliko rezultatov testov s tem, da naštejemo vse vhodne in izhodne parametre, navedemo njihove opise in podatkovne tipe.

Imenik vsebuje tudi podimenik src, v katerem najdemo naslednje tri javanske datoteke:

- <project_name>TestCase.java – razred, ki opisuje posamezen testni primer oziroma navaja podatkovne strukture, potrebne za shranjevanje podatkov o testnem primeru (vhod, izhod).

- <project_name>TestSetIterator.java – iterator po posameznih testnih primerih v danih množicah. Tipična naloga tega razreda je prebrati (ali pripraviti za branje) vhodne podatke (npr. iz datoteke) glede na vsebino konfiguracijske datoteke in nastaviti ustrezne parametre testnega primera. Iterator predvideva, da posamezna vrstica v opisni datoteki testne množice predstavlja en testni primer. Če se administrator projekta odloči za drugačno strukturo, mora biti implementirani iterator izpeljan iz razreda AbstractTestSetIterator in s tem implementirane tudi vse metode tega razreda.

- <project_name>AbsAlgorithm.java – razred, v katerem podatke testnega primera pripravimo za izvajanje s katerimkoli algoritmom, umeščenim v sistemu. V tem razredu tudi definiramo glavo metodo execute(), katero pokličemo v metodi run(). Ker je čas izvajanja algoritma pravzaprav čas izvajanja metode run(), moramo paziti, da so parametri, ki jih podamo metodi execute() pripravljeni že izven metode run(). Za to poskrbi metoda init(). V tej javanski datoteki je potrebno zbrati tudi vse parametre in jih oblikovati v končen rezultat izvedenega testa.

4.4 Implementacija algoritmov

Opisi posameznih algoritmov, ki rešujejo dani problem, se nahajajo v imeniku <ALGator_data_root>/projects/PROJ-<project_name>/algs. Naj bo v tem imeniku podimenik ALG-<algorithm_name>, v kateri so podatki o določenem algoritmu. Tu se nahaja konfiguracijska datoteka, ki vsebuje informacije, vezane na sam algoritem (npr. ime, avtor, opis, naziv javanske datoteke algoritma ipd.). Poleg te datoteke ustvarimo še podimenik src, kjer umestimo datoteko ALG-<algorithm_name>. Ker je ta razred izpeljan iz abstraktnega razreda <project_name>AbsAlgorithm, je nujna tudi implementacija njegovih abstraktnih metod, torej tudi metode execute(). V tej metodi se izvede dejansko iskanje rešitve za dani problem.

4.5 Integracija problema izomorfnega podgrafa

Pred samo integracijo smo izbrane algoritme implementirali v programskem jeziku Java, v pomoč pa so nam bile implementacije v C-ju. Vse razrede, potrebne za pripravo podatkov pred samim iskanjem rešitve, smo zapakirali v datoteko JAR in jo vključili v datotečno strukturo, katero za svoje delovanje uporablja sistem ALGator.

Sledila je definicija problema v samem sistemu. V konfiguracijsko datoteko SubIso.atp (slika 4.3) smo zapisali splošne informacije projekta in navedli imena kasneje implementiranih algoritmov, pripravljenih testnih množic, uporabljenih datotek JAR ter javanskih razredov, ki omogočajo izvedbo testiranja algoritmov za naš problem.

Na nivoju projekta smo odločili tudi, kateri vhodni in izhodni parametri so tisti, ki nas bodo zanimali pri testiranju algoritmov in kasnejšo medsebojno primerjavo. Izbrane parametre smo vnesli v konfiguracijsko datoteko SubIso-em.atrd, katere del je prikazan na sliki 4.2.

Za vhodne podatke smo določili tako splošne informacije o testnem primeru, kakršno je ime testa in ime skupine testa, pomembnejša pa sta podatka o številu vozlišč vzorčnega in ciljnega grafa, med katerima iščemo podgrafni izomorfizem. Pričakujemo, da bosta ravno zadnja parametra tista, ki bosta močno vplivala na čas izvajanja algoritmov. V vhod je dodan še podatek o številu izomorfizmov, podan s strani avtorjev testnih baz. Naj na tem mestu omenimo, da so te informacije na voljo samo za manjše grafe. Na podlagi pravilnosti izračuna algoritmov na majhnih grafih lahko sklepamo tudi o pravilnosti izračuna na večjih grafih.

Ker je naš problem definiran kot preštevalni, je prvi izhodni podatek število najdenih izomorfnih podgrafov. Če nam je na vhodu na voljo podatek o dejanskem številu izomorfizmov, lahko preverimo, ali je opravljen izračun pravilen. Rezultat kontrole je izpis *OK* za pravilno rešitev oziroma *NOK* za napačno rešitev, če pa pravilnosti ni mogoče določiti, pa izpišemo *N/A*. Prav tako pomembni izhodni parametri so tudi časi izvajanja, v našem primeru so to najmanjši, največji, povprečni in skupni čas.

```
{
  "ResultDescription":
  {
    "Format"           : "CSV",
    "Delimiter"        : ";",
    "TestParameters"   : ["Test", "Group", "N", "M", "IsoNo"],
    "ResultParameters" : ["IsoCount", "Tmax", "Tmin", "Tavg", "Tsum", "Check"],

    "Parameters":   [
      {
       "Name":         "Test",
       "Description": "The name of the test",
       "Type":         "string"
      },
      {
       "Name"        : "Group",
       "Description" : "The name of the group to which this test belongs",
       "Type"        : "string"
      },
      {
       "Name":         "N",
       "Description": "Pattern graph size (number of vertices)",
       "Type":         "int"
      },
      {
       "Name"        : "M",
       "Description" : "Target graph size (number of vertices)",
       "Type"        : "int"
      },
      {
       "Name"        : "IsoNo",
       "Description" : "No. of subgraph isomorphisms (for correctness check)",
       "Type"        : "int"
      },
```

Slika 4.2: Del konfiguracijske datoteke SubIso-em.atrd, katera določa vhodne in izhodne parametre, ki jih želimo vključiti v rezultate testov.

```
{
  "Project" : {
    "Description"          : "Comparison of several algorithms for subgraph isomorphism problem solving",
    "Author"               : "Peter Remec",
    "Date"                 : "16/11/2014",

    "Algorithms"           : ["Ullmann","UllmannImpDD","UllmannImpND","RI"],

    "TestSets"             : ["si2_b03_m200"],

    "AlgorithmClass"       : "SubIsoAbsAlgorithm",
    "TestCaseClass"        : "SubIsoTestCase",
    "TestSetIteratorClass" : "SubIsoTestSetIterator",

    "ProjectJARs"          : ["si.jar"],
    "AlgorithmJARs"        : ["si.jar"]
  }
}
```

Slika 4.3: Primer konfiguracijske datoteke SubIso.atp s podatki projekta v sistemu ALGator, definiranega za problem podgrafnega izomorfizma.

4.5.1 Razred SubIsoTestCase

Naslednja naloga administratorja je implementacija javanskih razredov, značilnih za izbran problem. Zgradbo testnega primera v ALGator-ju določa razred SubIsoTestCase, predstavljen v dodatku A.1. Njegova atributa patternGraph in targetGraph sta seznama seznamov povezav iz vseh vozlišč vzorčnega in ciljnega grafa, napolnimo pa jih z branjem datoteke. Za to strukturo smo se odločili zaradi enostavnosti. Izbrani algoritmi namreč za svoje delovanje zahtevajo različne strukture, ker pa se hočemo pri branju datotek izogniti kreiranju struktur, ki jih kasneje sploh ne bomo uporabili, je to opravilo preloženo v čas izvajanja testnega primera. S tem dosežemo, da vsak algoritem iz danega seznama kreira samo tiste strukture, ki jih dejansko potrebuje za iskanje rešitve. Atribut isoCount je namenjen za zapis rezultata (števila najdenih izomorfnih podgrafov).

4.5.2 Razred SubIsoTestSetIterator

V razredu SubIsoTestSetIterator (dodatek A.2) metoda getCurrent() določa način rokovanja s podatki opisne datoteke testne množice. Ker je privzeti način »ena vrstica, en testni primer« ustrezen tudi pri naši obravnavi, je bila izbira le-tega najbolj logična, zato je naš iterator implementacija abstraktnega razreda DefaultTestSetIterator. Izbrani format ene vrstice opisne datoteke, katere primer je prikazan na sliki 4.4, je naslednji:

test_name : group_name : isoNo : file1 : file2.

Z dvopičji torej ločimo ime testa, ime skupine testa, število obstoječih podgrafnih izomorfizmov, ime datoteke vzorčnega grafa in ime datoteke ciljnega grafa. Za ime testa izberemo enostavno testX, kjer je X zaporedna številka testa v testni množici, začenši z 0. Ime skupine testa bi lahko v našem primeru izpustili, saj vsi pripadajo skupini FILE (grafi so prebrani iz datoteke), a smo zaradi prikaza prilagodljivosti sistema zapis ohranili. V metodi getCurrent() torej vse te podatke izluščimo iz posamezne vrstice opisne datoteke in jih v obliki vhodnih parametrov, določenih v konfiguracijski datoteki projekta, podamo objektu tipa SubIsoTestCase. S seznami povezav prebranih vzorčnih in ciljnih grafov napolnimo tudi atribute patternGraph in targetGraph omenjenega objekta.

```
1  test0:FILE:1:si2_b03_m200.A00:si2_b03_m200.B00
2  test1:FILE:1:si2_b03_m200.A01:si2_b03_m200.B01
3  test2:FILE:1:si2_b03_m200.A02:si2_b03_m200.B02
4  test3:FILE:1:si2_b03_m200.A03:si2_b03_m200.B03
5  test4:FILE:1:si2_b03_m200.A04:si2_b03_m200.B04
6  test5:FILE:1:si2_b03_m200.A05:si2_b03_m200.B05
7  test6:FILE:1:si2_b03_m200.A06:si2_b03_m200.B06
8  test7:FILE:1:si2_b03_m200.A07:si2_b03_m200.B07
9  test8:FILE:1:si2_b03_m200.A08:si2_b03_m200.B08
10 test9:FILE:1:si2_b03_m200.A09:si2_b03_m200.B09
```

Slika 4.4: Ena izmed opisnih datotek, ki smo jo vključili v testiranje algoritmov. Vsebina te datoteke ustreza eni testni množici, vsaka vrstica pa enemu testnemu primeru iz te množici.

4.5.3 Razred **SubIsoAbsAlgorithm**

Glavna naloga razreda SubIsoAbsAlgorithm (dodatek A.3) je klic metode execute(), ki dejansko poišče rešitev, in implementacija metode done(), v kateri kot izhodni parameter v množico parametrov dodamo število najdenih podgrafnih izomorfizmov testnega primera in rezultat preverjanja pravilnosti rešitve. V tem razredu deklariramo tudi abstraktno metodo execute(), katero bo kasneje vsak algoritem implementiral po svoje.

4.5.4 Konfiguracijske datoteke in integracija algoritmov

Ker bomo za testiranje uporabljali pare datotek z vzorčnimi in ciljnimi grafi, je potrebno le-te skupaj z opisnimi in konfiguracijskimi datotekami vključiti v imenik <ALGator_data_root>/projects/PROJ-SubIso/tests. V posamezno opisno datoteko smo vključili 100 testov na podlagi skupnih lastnosti grafov. Z namenom čim bolj zmanjšati vpliv drugih procesov, ki tečejo na računalniku med testom, smo se odločili, da vsak testni primer ponovimo trikrat. Ta parameter skupaj z ostalimi podatki, kot so opis testne množice, število testnih primerov v njej ter ime opisne datoteke, ki pripada tej množici, podamo v konfiguracijski datoteki s končnico .atts (primer na sliki 4.5). Podrobneje je testiranje in priprava nanj opisana v

```
1  {"TestSet": {
2      "Description"       : "si2_b03_s20",
3      "ShortName"         : "si2_b03_s20",
4      "N"                 : 100,
5      "TestSetFiles"      : ["si2_b03_s20.*"],
6      "DescriptionFile"   : "si2_b03_s20.txt",
7      "TimeLimit"         : "100",
8      "TestRepeat"        : "3"
9    }
10 }
```

Slika 4.5: Primer konfiguracijske datoteke s končnico .atts, ki služi opisu posamezne testne množice. V njej med drugimi navedemo ime opisne datoteke testne množice, število ponovitev posameznega testnega primera in časovno omejitev izvajanja testnega primera.

naslednjem poglavju tega diplomskega dela.

Na tej točki je projekt pripravljen za sprejem algoritmov, ki bodo naš problem reševali. Naš prispevek obsega Ullmannov algoritem, RI in dve verziji izboljšanega Ullmannovega algoritma. Ti verziji se razlikujeta le v načinu razvrščanja vozlišč vzorčnega grafa pred iskanjem rešitve. Pričakujemo, da bo tudi vrstni red obdelave vozlišč vplival na učinkovitost iskanja rešitve, zato smo zaradi lažje primerjave v sistem vpeljali dva ločena algoritma.

Za integracijo algoritmov v sistem ALGator smo kreirali naslednje datoteke:

- UllmannSubIsoAlgorithm.java v imeniku <ALGator_data_root>/projects/PROJ–SubIso/algs/ALG–Ullmann/src,
- UllmannImpDDSubIsoAlgorithm.java v imeniku <ALGator_data_root>/projects/PROJ–SubIso/algs/ALG–UllmannImpDD/src,
- UllmannImpNDSubIsoAlgorithm.java v imeniku <ALGator_data_root>/projects/PROJ–SubIso/algs/ALG–UllmannImpND/src in
- RISubIsoAlgorithm.java v imeniku <ALGator_data_root>/projects/PROJ–SubIso/algs/ALG–RI/src.

Vse naštete javanske razrede smo izpeljali iz abstraktnega razreda SubIsoAbsAlgorithm, ki smo ga definirali na nivoju projekta. V njih smo implementirali metodo execute(), ki je kot abstraktna metoda napovedana že v nadrazredu. Vanjo smo vključili inicializacijo struktur, ki jih algoritem potrebuje za iskanje rešitev kot tudi sam postopek reševanja problema. Metoda execute() vrača število najdenih podgrafnih izomorfizmov in ga zapiše v atribut objekta tipa SubIsoTestCase, katerega smo poimenovali isoCount. Omenjeni atribut potrebujemo za zapis rešitve v izhodni parameter, ki se izvede v metodi done() nadrazreda SubIsoAbsAlgoritem.

Poleg javanskih razredov smo vsakemu algoritmu dodali še konfiguracijsko datoteko s končnico .atal, v njej pa so navedene informacije, specifične za algoritem, npr. njegovo ime, opis ter pripadajoč javanski razred. Datoteko smo umestili v imenik <ALGator_data_root>/projects/PROJ–SubIso/algs/ALG–<algorithm_name>, primer le-te pa je prikazan na sliki 4.6.

```
1  {
2    "Algorithm" : {
3     "ShortName"       : "RI",
4     "Description"     : "RI algorithm for subgraph isomorphism problem solving",
5     "Author"          : "Peter Remec",
6     "Date"            : "24.10.2014",
7     "Classes"         : ["RISubIsoAlgorithm.java"],
8     "MainClassName"   : "RISubIsoAlgorithm"
9    }
10 }
```

Slika 4.6: Primer konfiguracijske datoteke z informacijami o algoritmu (končnica .atal).

Poglavje 5

Eksperimentalna primerjava algoritmov

5.1 Testne množice

5.1.1 ARG Database

ARG Database[13, 14] je velika zbirka označenih in neoznačenih grafov, realizirana v okviru skupine MIVIA [30] z namenom ponuditi raziskovalcem na področju grafov standardni paket za testiranje različnih algoritmov.

Tako označeni kot neoznačeni grafi so naključno tvorjeni s šestimi različnimi modeli generiranja, vsak od njih pa vključuje drugačne parametre. Na ta način pridemo do kar 168 raznolikih podskupin grafov, združenih v celoto. Vsak izmed tipov neoznačenih grafov je predstavljen z več tisoč dvojicami grafov, med katerima obstaja podgrafni izomorfizem. Tudi različni tipi označenih grafov nastopajo v parih, med katerima obstaja netrivialen skupni podgraf. Baza skupno vsebuje 143.600 neoznačenih in 166.000 označenih grafov.

5.2 Testi

Datoteke z grafi smo za izvedbo testov grupirali podobno, kot je to že izvedeno v zbirki ARG Database (več o zgradbi baze lahko bralec prebere v [13]). Na podlagi

lastnosti grafov so nastale naslednje kategorije le-teh:

- naključno povezani grafi,
- regularne in neregularne mreže (2D, 3D in 4D),
- grafi z omejeno stopnjo vozlišč in njihovi neregularni primerki.

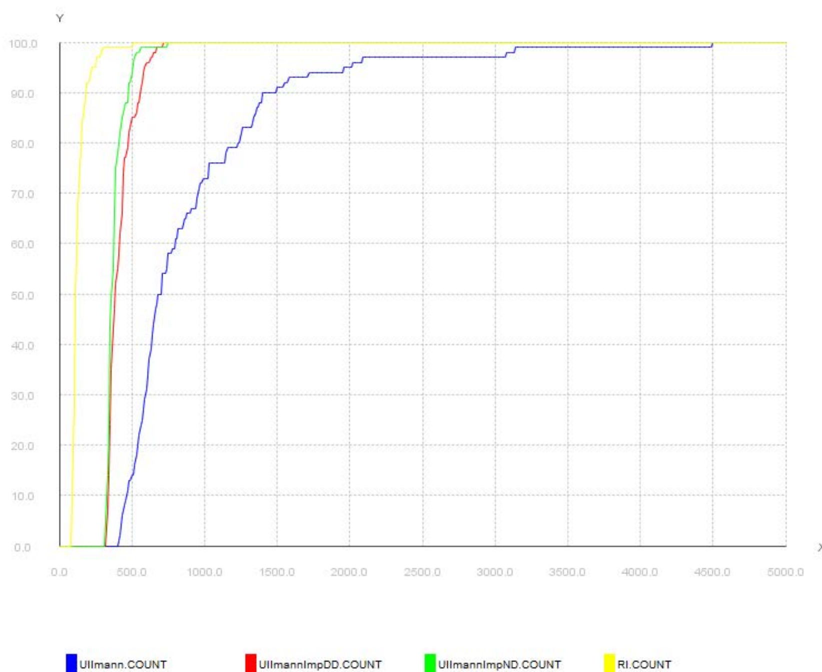
Glede na velikost in kategorijo grafov smo za testiranje pripravili 381 testnih množic, vsaka od njih pa vsebuje 100 testnih primerov (parov vzorčni graf - ciljni graf). Tako imamo na voljo kar 38100 različnih testnih primerov.

Dodatno testne primere razvrstimo še na podlagi parametrov, kot so gostota povezav, največja stopnja vozlišč in dimenzija. Na nivoju projekta določimo zgornjo časovno mejo izvajanja posameznega testnega primera na 60 sekund. Če v tem času ne najdemo rešitve, se test nadaljuje z naslednjim testnim primerom, trenutni pa je označen kot neuspešno končan. Vsak testni primer se izvede trikrat, v datoteko pa se zapiše najmanjši, največji in povprečen čas izvajanja.

Opazovali smo odstotek uspešno končanih testnih primerov, med temi pa smo izračunali tudi povprečen najmanjši čas izvajanja. Rezultate smo tabelarično in grafično (število rešenih testnih primerov v odvisnosti od časa izvajanja) predstavili v naslednjem podpoglavju. Naj pojasnimo, da smo grafični prikaz rezultatov pripravili v Microsoftovem orodju Excel, saj je ALGator za obdelavo večje količine podatkov trenutno še premalo optimiziran, da bi delo opravil v sprejemljivem času. Podatke za generiranje grafov smo v Excel uvozili iz datotek, katere smo kreirali s pomočjo enostavnega poizvedovalnega jezika sistema ALGator. Za ilustracijo grafičnega prikaza v ALGator-ju smo izvedli analizo na eni izmed testnih množic (5.1).

Celoten test smo izvedli z Ullmannovim algoritmom (v nadaljevanju ga označujemo z *Ull*), izboljšanim Ullmannovim algoritmom v dveh različicah (oznaki *UllImpND* in *UllImpDD*) in algoritmom RI. Ti so tekli na sistemu z naslednjimi specifikacijami:

- procesor Intel Core i5 2.6 GHz z dvema jedroma,
- 8 GB delovnega pomnilnika,
- operacijski sistem Microsoft Windows 7 Professional.



Slika 5.1: Primer grafičnega prikaza rezultatov ene izmed testnih množic našega testa v ALGatorju. Graf prikazuje število rešenih testnih primerov v odvisnosti od časa. S pomočjo enostavnega grafičnega vmesnika izberemo parametre za zajem podatkov (algoritem, testna množica, vhodni in izhodni parametri testa,...). Poizvedba sistema ALGator z izbranimi parametri poskrbi za prikaz podatkov v tabeli in grafu.

5.3 Rezultati eksperimenta

5.3.1 Naključno povezani grafi

Ta kategorija vsebuje 6300 testnih primerov (parov vzorčni graf - ciljni graf). Pri generiranju teh grafov je podan parameter η, ki predstavlja verjetnost prisotnosti povezave med dvema različnima vozliščema. Ob tem je bilo prevzeto, da je verjetnostna porazdelitev enakomerna. Če je n število vozlišč grafa, je število dodanih povezav $\eta N(N-1)$. Če te povezave ne tvorijo povezanega grafa, se dodajanje povezav nadaljuje, dokler graf ni povezan.

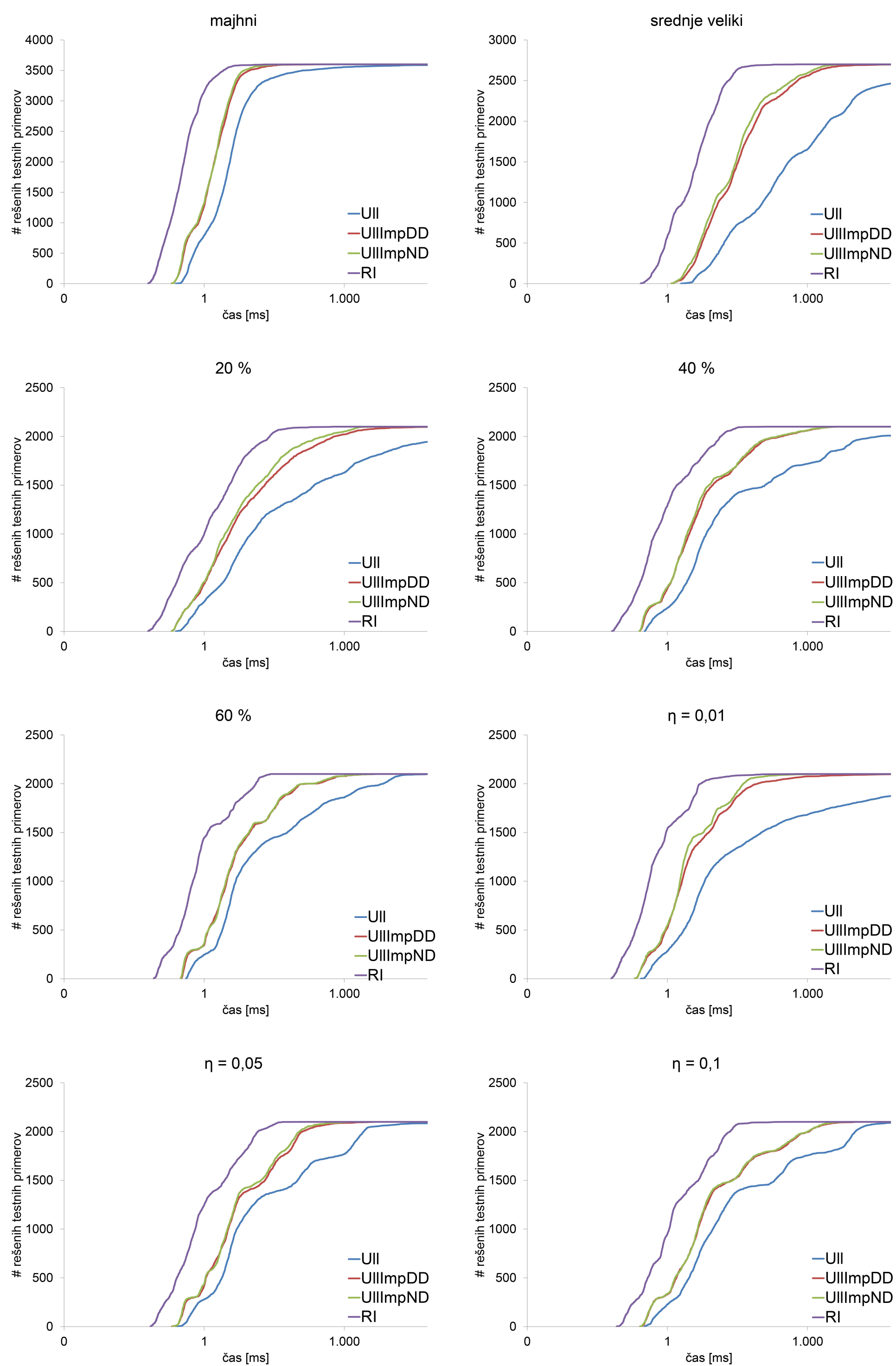
Naključno povezane grafe grupiramo v manjše množice glede na:

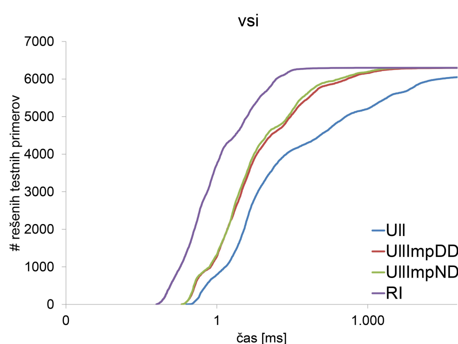
- število vozlišč ciljnega grafa (majhni grafi z 20 do 80 vozlišči, veliki grafi s 100 do 400 vozlišči),
- velikost vzorca (20, 40 in 60 odstotkov velikosti ciljnega grafa) in
- gostoto grafov (verjetnost povezave $\eta = 0.01, 0.05, 0.1$).

Primerjava števila rešenih testnih primerov v 60 sekundah in povprečnega najmanjšega časa izvajanja izbranih algoritmov je prikazana v tabeli 5.1, na sliki 5.2 pa je predstavljena odvisnost števila rešenih testnih primerov od časa izvajanja algoritmov.

Ugotovitve na podlagi rezultatov testa lahko strnemo v nekaj točkah:

- Algoritem RI se je pokazal kot daleč najbolj zmogljiv, Ull pa je svoje delo opravil najslabše in hkrati najpočasneje, tudi za faktor 100 in več v primerjavi z RI. Razlika v času izvajanja je najbolj očitna pri največjih grafih. RI in UllImpND v tej kategoriji rešita prav vse testne primere v določeni časovni meji, Ull pa pričakovano najmanj.
- Pri majhnih grafih vsi algoritmi rešitev najdejo praktično v trenutku, izjema je le Ull, ki ima tudi že nekaj nerešenih primerov.
- Pri vseh algoritmih opazimo, da se z rastjo odstotka velikosti vzorčnega grafa v primerjavi s ciljnim grafom krajša najmanjši čas izvajanja. Največja razlika je vidna pri UllImpDD. Posledično se poviša tudi odstotek uspešno rešenih primerov.
- Do zanimive ugotovitve pridemo pri grupiranju grafov glede na gostoto. Če za RI in UllImpND velja, da hitreje najdejo rešitev v redkejših grafih, pa to ne velja povsem za Ull in UllImpDD. Slednja se časovno najslabše izkažeta pri $\eta = 0.05$, medtem ko so pri ostalih dveh vrednosti η najmanjši časi primerljivi. Odstotek rešenih primerov z Ull se z gostoto grafov opazno poviša.
- Kategorija naključno povezanih grafov je tista, v kateri se prav vsi algoritmi v primerjavi z drugima dvema kategorijama odrežejo najslabše.


majhni
srednje veliki
20 %
40 %
60 %
η = 0,01
η = 0,05
η = 0,1
rešenih testnih primerov
čas [ms]
0
1
1.000
UlI
UlIImpDD
UlIImpND
RI



Slika 5.2: Grafična predstavitev rezultatov testiranja algoritmov na naključno povezanih grafih.

Grafi	# testnih primerov	% rešenih testnih primerov				povprečen najmanjši čas izvajanja [ms]			
		Ull	UllImpND	UllImpDD	RI	Ull	UllImpND	UllImpDD	RI
majhni	3600	99,64	100	100	100	88,62	2,32	2,74	0,69
srednje veliki	2700	91,22	100	99,89	100	2668,45	140,39	272,72	7,58
20 %	2100	92,57	100	99,86	100	1506,62	75,75	237,41	5,76
40 %	2100	95,62	100	100	100	1137,82	62,04	66,56	2,83
60 %	2100	99,90	100	100	100	799,17	46,54	51,32	2,34
$\eta = 0,01$	2100	89,29	100	99,86	100	1307,28	12,69	134,48	2,35
$\eta = 0,05$	2100	99,29	100	100	100	519,98	26,75	41,32	3,45
$\eta = 0,1$	2100	99,52	100	100	100	1605,24	144,88	179,34	5,13
skupaj	6300	96,03	100	99,95	100	1138,89	61,44	118,37	3,65

Tabela 5.1: Prikaz rezultatov testiranja algoritmov na naključno povezanih grafih z različnimi parametri.

5.3.2 Večdimenzionalne mreže

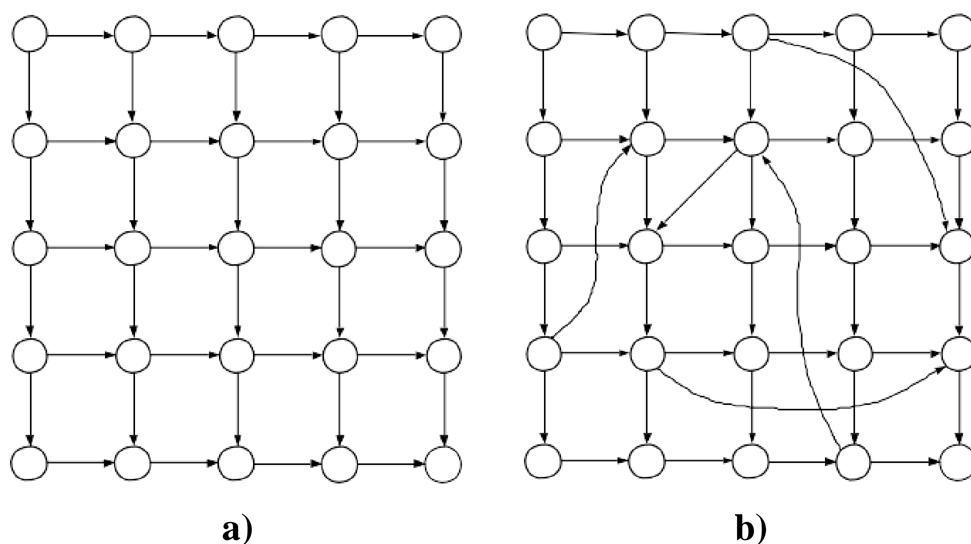
V skupini večdimenzionalnih mrež imamo 19200 testnih primerov, te pa nadalje grupiramo na podlagi:

- števila vozlišč ciljnega grafa (majhni grafi s 16 do 81 vozlišči, veliki grafi s 100 do 512 vozlišči),
- velikosti vzorca (20, 40 in 60 odstotkov velikosti ciljnega grafa),
- dimenzije (2D, 3D in 4D mreže) in
- regularnosti (regularni in neregularni grafi s stopnjo neregularnosti $\rho = 0.2, 0.4, 0.6$).

Že samo ime nakazuje, da so mreže grafi z vozlišči, ki so povezani z vsemi ostalimi vozlišči iz svoje soseščine. Slika 5.3 prikazuje dvodimenzionalno regularno in neregularno mrežo velikosti 5×5. Vozlišča mreže so povezana z vsemi štirimi sosednjimi vozlišči. Izjeme so robna vozlišča, ki so povezana le z notranjimi sosedi. Sledi, da je v 3D mrežah teh sosedov šest, v 4D mrežah pa osem.

Neregularna mreža se tvori iz regularne z dodajanjem $N\rho$ dodatnih povezav, kjer je N število vozlišč regularne mreže, ρ pa stopnja neregularnosti (konstanta, večja od 0). Neregularno 2D mrežo dimenzije 5×5 z $\rho = 0,2$ torej dobimo tako, da regularni 2D mreži iste velikosti dodamo še pet naključnih povezav ($25 \cdot 0,2 = 5$).

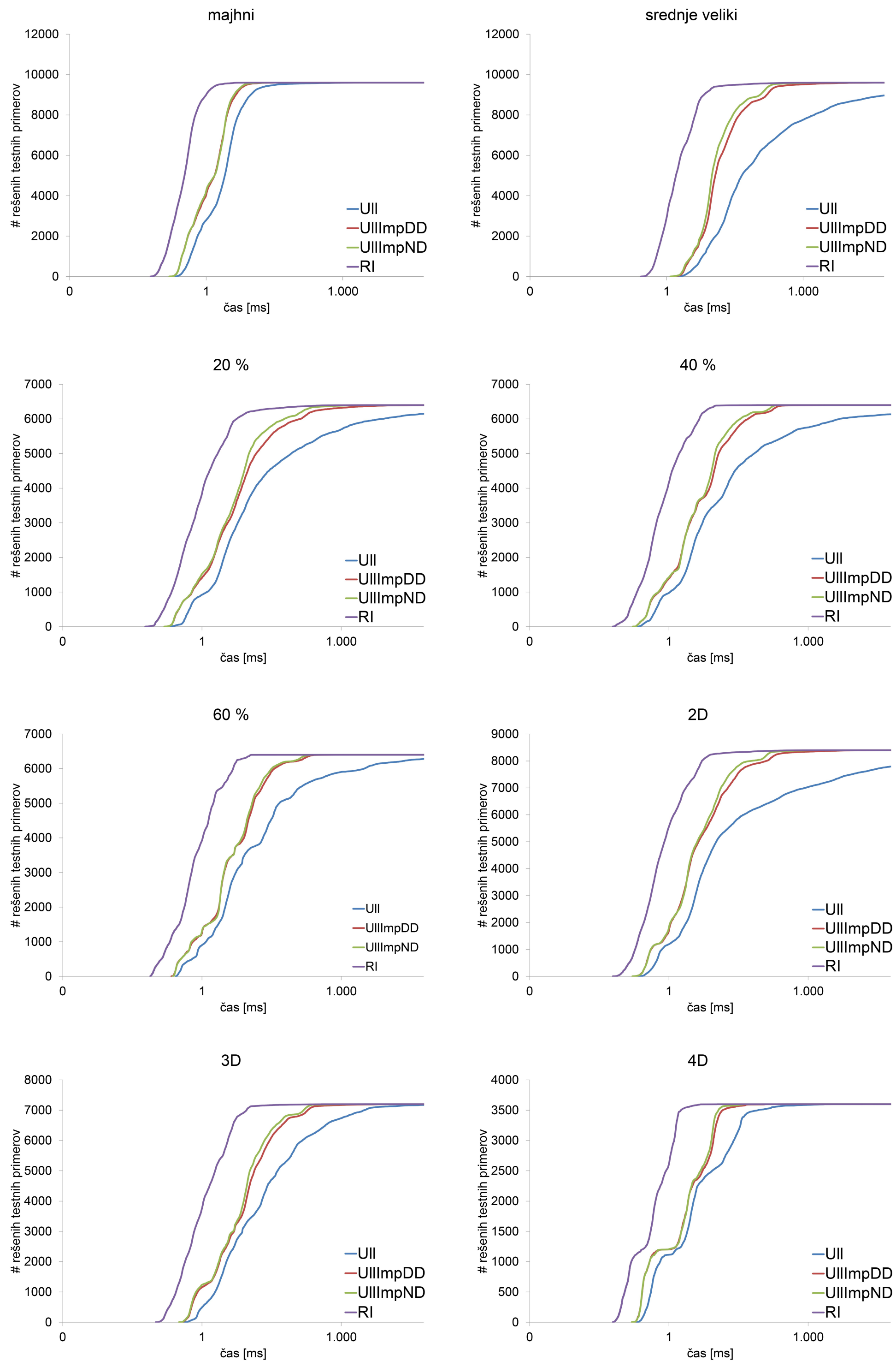
Tabela 5.2 vsebuje rezultate testnih primerov iz kategorije večdimenzionalnih mrež glede na že omenjene parametre. Slika 5.4 te podatke prikazuje v grafični obliki. Ugotovitve so naslednje:

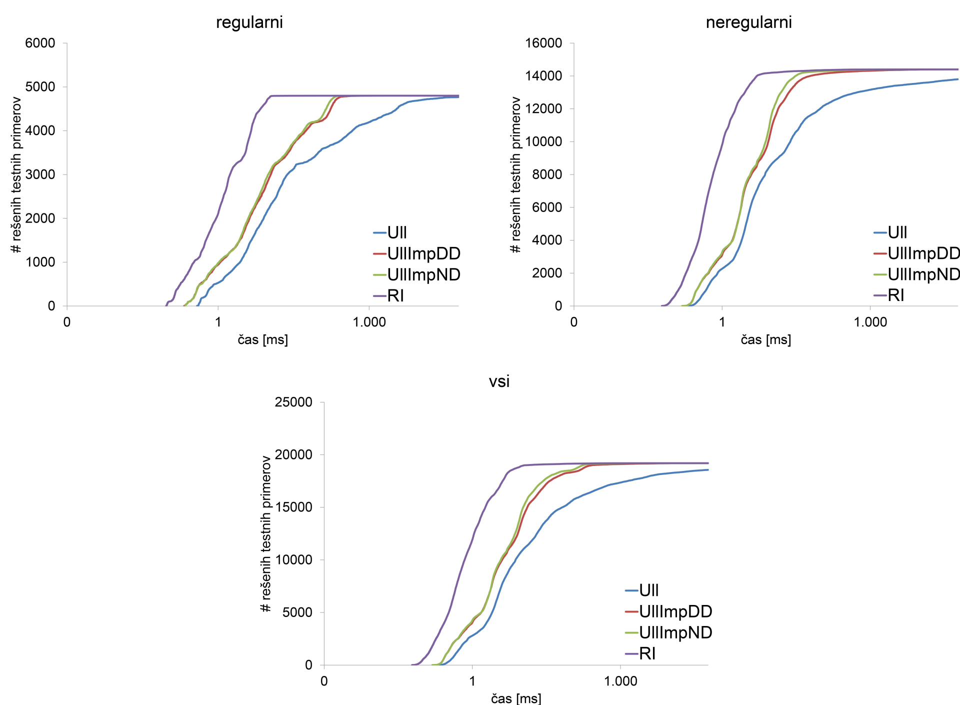


Slika 5.3: Dvodimenzionalna a) regularna in b) neregularna mreža velikosti 5×5. Stopnja neregularnosti slednje je $0,2$. Vir: [13].

- Ri se je spet pokazal kot najhitrejši, sledita UllImpND in UllImpDD, daleč zadaj pa je Ull. Prvi trije so uspešni izvedli skoraj vse testne primere, Ull pa nekoliko manj.

- Na majhnih grafih so stoodstotni vsi štirje algoritmi, tudi časi izvajanja se bistveno ne razlikujejo. Drastično pa se razlika poveča na srednje velikih


majhni
srednje veliki
20 %
40 %
60 %
2D
3D
4D
rešenih testnih primerov
čas [ms]
UII
UIIImpDD
UIIImpND
RI



Slika 5.4: Grafična predstavitev rezultatov testiranja algoritmov na večdimenzionalnih mrežah.

grafih, kjer je Ull počasnejši od RI za več kot 250-krat.

- Pri vseh algoritmih ob naraščanju velikosti vzorčnega grafa raste tudi odstotek uspešnih testov. UllImpND in UllImpDD z večjim vzorcem delo opravita hitreje, RI in Ull pa se pri 60 % vzorcu nekoliko presenetljivo upočasnita.

Grafi	# testnih primerov	% rešenih testnih primerov				povprečen najmanjši čas izvajanja [ms]			
		Ull	UllImpND	UllImpDD	RI	Ull	UllImpND	UllImpDD	RI
majhni	9600	100	100	100	100	6,74	1,71	1,79	0,41
srednje veliki	9600	93,39	100	99,98	100	1315,59	31,46	80,48	5,09
20 %	6400	96,08	100	99,97	100	708,59	26,89	94,09	5,16
40 %	6400	95,84	100	100	100	536,50	11,48	16,00	1,33
60 %	6400	98,16	100	100	100	670,31	11,38	13,31	1,75
2D	8400	92,76	100	99,99	100	1165,08	16,61	57,18	3,04
3D	7200	99,64	100	99,99	100	376,60	22,90	40,67	3,08
4D	3600	99,97	100	100	100	21,90	3,89	4,58	1,39
regularni	4800	99,31	100	100	100	737,89	27,62	33,82	2,79
neregularni	14400	95,82	100	99,99	100	604,54	12,90	43,56	2,74
skupaj	19200	96,69	100	99,99	100	638,78	16,58	41,13	2,75

Tabela 5.2: Prikaz rezultatov testiranja algoritmov na večdimenzionalnih mrežah z različnimi parametri.

- Ull ima najmanjši odstotek uspešnosti pri 2D mrežah in je ob tem izrazito počasnejši od ostalih treh. Čas izvajanja se pri vseh algoritmih ob dodatni dimenziji grafov krajša.

- Vsi algoritmi razen UllImpDD so svoje delo bolje opravili na neregularnih kot regularnih grafih.

5.3.3 Grafi z omejeno stopnjo vozlišč

Algoritme smo izvedli tudi na 12600 parih grafov z omejeno stopnjo vozlišč. Z Δ označimo zgornjo mejo stopnje, katero ne sme preseči nobeno vozlišče v teh grafih. V našem primeru so grafi generirani tako, da se v njih naključno dodajajo povezave, dokler vsa vozlišča ne dosežejo stopnje Δ.

Spremenjeni grafi z omejeno stopnjo vozlišč so tvorjeni tako, da se naključno premakne 10 % obstoječih povezav. S tem seveda stopnje nekaterih vozlišč presežejo Δ, a je razmerje števila povezav proti številu vozlišč še vedno omejeno.

Testne scenarije smo določili po naslednjih parametrih:

- število vozlišč ciljnega grafa (majhni grafi z 20 do 80 vozlišči, veliki grafi s 100 do 400 vozlišči),

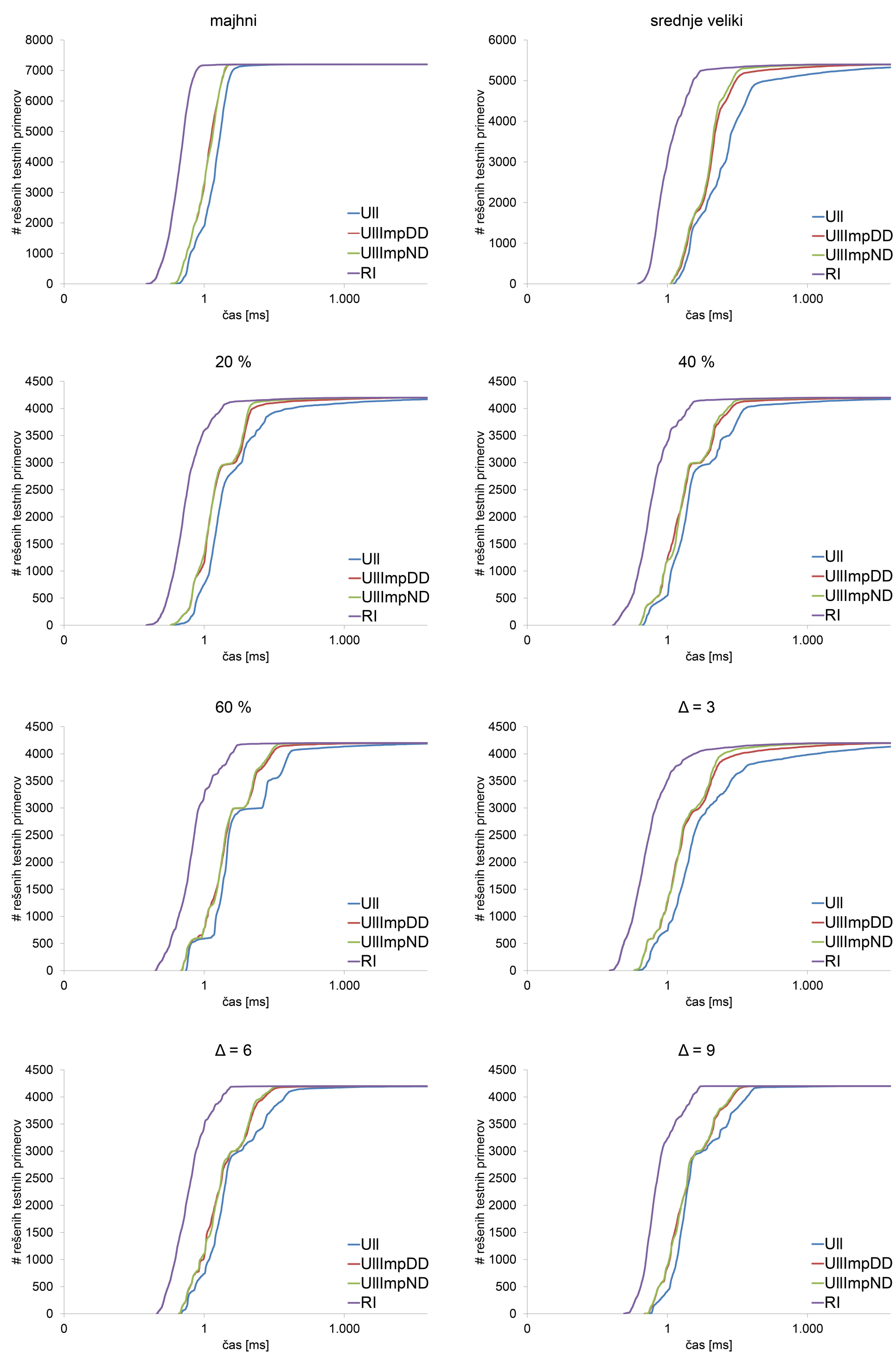
- velikost vzorca (20, 40 in 60 odstotkov velikosti ciljnega grafa),

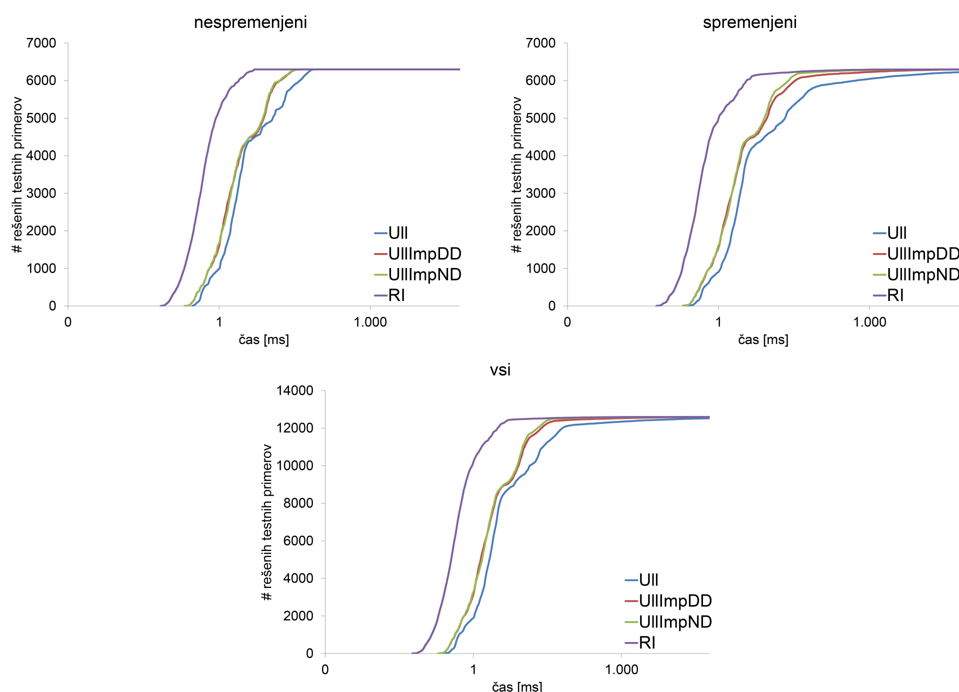
- zgornja meja stopnje vozlišč - Δ in

- spremembe na grafih (nespremenjeni in spremenjeni grafi z 10 % premaknjenih povezav).

Tabela 5.3 in slika 5.5 predstavljata rezultate testiranja algoritmov na grafih z omejeno stopnjo vozlišč. Sledi kratek komentar le-teh:

- Tako kot v kategoriji večdimenzionalnih mrež so tudi tu algoritmi na majhnih grafih brez neuspešnega iskanja rešitve. Časi izvajanja so primerljivi, razlike


majhni
srednje veliki
20 %
40 %
60 %
Δ = 3
Δ = 6
Δ = 9
rešenih testnih primerov
čas [ms]
UII
UIIImpDD
UIIImpND
RI



Slika 5.5: Grafična predstavitev rezultatov testiranja algoritmov na grafih z omejeno stopnjo vozlišč.

pa se povečajo pri srednje velikih grafih, kjer je tudi nekaj testnih primerov neuspešno izvedenih.

- RI ob povečanju velikosti vzorca izvede testne primere hitreje, na Ull pa ta sprememba skorajda ne vpliva. UllImpND in UllImpDD sta bila najhitrejša na vzorcih s 60 % velikostjo glede na ciljni graf.

- Z dvigovanjem meje stopenj vozlišč se tudi odstotek rešenih primerov povečuje, čas izvajanja pa pada. To velja za vse algoritme, katere smo testirali.

- Testi na nespremenjenih grafih so tako odstotkovno kot časovno uspešnejši od tisti s premaknjenimi povezavami (velja za vse algoritme).

- RI je vse teste v tej kategoriji opravil uspešno, bil pa je tudi najhitrejši. Ull je bil občutno hitrejši kot v ostalih dveh kategorijah in je imel boljšo uspešnost, a še vedno precej zaostaja od ostalih treh algoritmov.

Grafi	# testnih primerov	% rešenih testnih primerov				povprečen najmanjši čas izvajanja [ms]			
		Ull	UllImpND	UllImpDD	RI	Ull	UllImpND	UllImpDD	RI
majhni	7200	100	100	100	100	3,98	1,25	1,26	0,34
srednje veliki	5400	98,63	99,98	99,91	100	381,73	34,79	107,25	13,46
20 %	4200	99,24	99,98	99,95	100	184,62	16,61	39,38	9,70
40 %	4200	99,31	100	99,95	100	142,36	23,40	75,75	6,90
60 %	4200	99,69	100	99,98	100	166,82	6,87	24,85	1,30
$\Delta = 3$	4200	98,38	99,98	99,88	100	442,06	37,24	127,18	16,24
$\Delta = 6$	4200	99,86	100	100	100	36,91	4,47	7,33	0,74
$\Delta = 9$	4200	100	100	100	100	19,13	5,16	5,56	0,91
nespremenjeni	6300	100	100	100	100	7,83	4,12	4,26	0,69
spremenjeni	6300	98,83	99,98	99,92	100	323,23	27,13	89,10	11,24
skupaj	12600	99,41	99,99	99,96	100	164,60	15,63	46,66	5,96

Tabela 5.3: Prikaz rezultatov testiranja algoritmov na grafih z omejeno stopnjo vozlišč po različnih scenarijih.

Poglavje 6

Sklepne ugotovitve

Z uporabo sistema ALGator smo na bazi različnih grafov opazovali delovanje štirih algoritmov za reševanje problema podgrafnega izomorfizma in izvedli analizo rezultatov. V prav vseh testnih scenarijih se je pričakovano najbolje obnesel algoritem RI, najslabše pa Ullmannov algoritem. Slednji kljub starosti še vedno služi kot odlična osnova za razvoj novih algoritmov. Dokaz za to je izboljšani Ullmannov algoritem, katerega smo obravnavali v dveh različicah. Opazili smo, da tudi izbira načina razvrščanja vozlišč grafov za obdelavo nekoliko vpliva na zmogljivost samega algoritma.

Baza grafov, ki smo jo uporabili, sicer vsebuje tudi večje grafe, zato bi veljalo algoritme testirati tudi na njih. Z naraščanjem števila vozlišč grafov pa se občutno povečuje tudi čas izvajanja in posledično čas izvedbe celotnega testa. Algoritme smo sicer poskusno izvedli tudi na nekaj velikih grafih, a smo hitro ugotovili, da je uporaba le-teh (predvsem v kombinaciji z Ullmannovim algoritmom, v manjši meri pa tudi z različicami izboljšanega Ullmannovega algoritma in algoritma RI) preveč časovno potratna. S tem razlogom smo se omejili le na pare majhnih in srednje velikih grafov (do približno 500 vozlišč), za katere smo imeli na voljo tudi podatke o pravilnem številu podgrafnih izomorfizmov, katere smo lahko primerjali z našimi rezultati.

Zanimiva bi bila tudi uporaba algoritmov na kateri izmed realnih zbirk grafov (npr. Graemlin, ki je opisan v [12]) in primerjava z ARG bazo, ki je programsko generirana. Za to bi bilo potrebno ustrezno strukturirati pare, med katerimi bi iskali podgrafne izomorfizme. Poleg tega bi bili dobrodošli tudi podatki o pravilnem

številu izomorfizmov, kateri pa trenutno (še) niso na voljo.

Integracija problema izomorfnega podgrafa v ALGator je eden izmed prvih projektov v sistemu. S tem smo ostalim uporabnikom ponudili testno ogrodje, katero je na voljo za uporabo na novih algoritmih, ki rešujejo isti problem. Projekt istočasno služi kot referenca pri kreiranju okolja za testiranje algoritmov poljubnega problema. S širitvijo nabora vsebovanih projektov pa bi lahko v prihodnje ALGator postal testno orodje ne le v študijske, ampak tudi v raziskovalne namene.

Kljub dobri zasnovi sistema pa smo ob uporabi opazili tudi nekaj pomanjkljivosti. Nekaj izmed njih je bilo tekom izdelave projekta že odpravljenih (npr. omejevanje časa izvajanja testnega primera, uporaba datotek JAR v projektu), priložnosti za izboljšavo pa so opazne predvsem v analitičnem delu ALGator-ja. Pri izboru večjega števila testnih množic za analizo se je v grafičnem vmesniku pokazalo počasno zbiranje rezultatov testov. Posledično je bil upočasnjen tudi grafični prikaz, zato smo večino grafov izrisali v programu Excel. V ta namen smo z uporabo ukazov v konzoli in poizvedbami sistema ALGator grupirali oziroma filtrirali tiste podatke, ki smo jih za posamezen graf potrebovali in jih izvozili v datoteko CSV, to pa smo nato uvozili v Excel.

Dodatek A

Implementacija javanskih razredov v ALGator-ju

Algoritem A.1: SubIsoTestCase.java

```
import java.util.ArrayList;
import si.fri.algotest.entities.TestCase;

public class SubIsoTestCase extends TestCase {

        // vhodna grafa - vzorcni in ciljni - kot seznama
        // seznamov vozlisc
        public ArrayList<ArrayList<Integer>> patternGraph;
        public ArrayList<ArrayList<Integer>> targetGraph;

        // rezultat - st. najdenih izomorfnih podgrafov
        public int isoCount;

        @Override
        public String toString() {
           return super.toString();
        }
}
```

Algoritem A.2: SubIsoTestSetIterator.java

```
import java.io.File;
import si.fri.algotest.entities.EParameter;
import si.fri.algotest.entities.EResultDescription;
import si.fri.algotest.entities.ETestSet;
import si.fri.algotest.entities.ParameterType;
import si.fri.algotest.entities.TestCase;
import si.fri.algotest.execute.DefaultTestSetIterator;
import si.fri.algotest.global.ErrorStatus;
import si.fri.algotest.tools.ATTools;
import utils.GraphReader;

public class SubIsoTestSetIterator extends
   DefaultTestSetIterator {

 String filePath;
 String testFileName;

 private void reportInvalidDataFormat(String note) {
     String msg = String.format("Invalid input data in file %
        s in line %d.",
              testFileName, lineNumber);
     if (!note.isEmpty()) {
         msg += " (" + note + ")";
     }

     ErrorStatus.setLastErrorMessage(ErrorStatus.ERROR, msg);
 }

 @Override
 public void initIterator() {
     super.initIterator();

     String fileName = testSet.getTestSetDescriptionFile();
     filePath = testSet.entity_rootdir;
     testFileName = filePath + File.separator + fileName;
 }

 @Override
 public TestCase getCurrent() {
     if (currentInputLine == null) {
         ErrorStatus.setLastErrorMessage(ErrorStatus.ERROR,
                 "No valid input!");
         return null;
     }

     SubIsoTestCase tCase = new SubIsoTestCase();
     EParameter testIDPar = EResultDescription.
```

```
   getTestIDParameter("Test-"
        + Integer.toString(lineNumber));
tCase.addParameter(testIDPar);

String[] fields = currentInputLine.split(":");

// vrstica v nasem primeru v resnici vsebuje 5
   parametrov: ime
//testa, ime skupine testa, st. izomorfizmov (za
   testiranje
// pravilnosti) ter datoteki z vzorcnim in ciljnim
   grafom
if (fields.length < 3) {
    reportInvalidDataFormat("Premalo parametrov v
       vrstici!");
    return null;
}

// ime testnega primera
String testName = fields[0];
// podatek o st. podgrafnih izomorfizmov
int noOfIso;

try {
    noOfIso = Integer.parseInt(fields[2]);

} catch (Exception e) {
    reportInvalidDataFormat("Parse error!");
    return null;
}
// skupina testnega primera (FILE v nasem primeru)
String group = fields[1];

EParameter testIDParameter = EResultDescription
        .getTestIDParameter("Test-" + Integer.toString(
           lineNumber));

EParameter parameter1 = new EParameter("Test", "Test
   name",
        ParameterType.STRING, testName);
EParameter parameter2 = new EParameter("Group",
        "A name of a group of tests", ParameterType.
           STRING, group);
EParameter parameter5 = new EParameter("IsoNo",
        "No. of isomorphisms (for correctness checking
           purposes)",
        ParameterType.INT, noOfIso);

tCase.addParameter(testIDParameter);
```

```
    // vhodni parametri
    tCase.addParameter(parameter1);
    tCase.addParameter(parameter2);
    tCase.addParameter(parameter5);

    // v nasem primeru vhodne podatke dobimo le iz datotek,
       tako da
    // drugih nacinov branja oz. generiranja podatkov nismo
       implementirali
    switch (group) {
        case "INLINE":
            break;
        case "RND":
            break;
        case "SORTED":
            break;
        case "INVERSED":
            break;
        case "FILE":
            try {
                if (fields.length != 5) {
                    throw new Exception("Premalo parametrov!
                       ");
                }
                String testFile1 = filePath + File.separator
                     + fields[3];
                String testFile2 = filePath + File.separator
                     + fields[4];
                tCase.patternGraph = new GraphReader().
                   readGraph(testFile1);
                tCase.targetGraph = new GraphReader().
                   readGraph(testFile2);
            } catch (Exception e) {
                reportInvalidDataFormat(e.toString());
            }
    }

    EParameter parameter3 = new EParameter("N", "Pattern
       graph size",
            ParameterType.INT, tCase.patternGraph.size());
    EParameter parameter4 = new EParameter("M", "Target
       graph size",
            ParameterType.INT, tCase.targetGraph.size());
    tCase.addParameter(parameter3);
    tCase.addParameter(parameter4);

    return tCase;
}
```

```
    // v main metodi izpisemo vse testne primere (s tem
       preverimo pravilnost
    // konfiguracije projekta)
    public static void main(String args[]) {
        String root = "C:/algator/test_data";
        String projName = "SubIso";

        ETestSet testSet = ATTools.getFirstTestSetFromProject(
           root, projName);
        SubIsoTestSetIterator stsi = new SubIsoTestSetIterator()
           ;
        stsi.setTestSet(testSet);

        ATTools.iterateAndPrintTests(stsi);
    }
}
```

Algoritem A.3: SubIsoAbsAlgorithm.java

```
import java.util.ArrayList;
import si.fri.algotest.entities.EParameter;
import si.fri.algotest.entities.ParameterSet;
import si.fri.algotest.entities.ParameterType;
import si.fri.algotest.entities.TestCase;
import si.fri.algotest.execute.AbsAlgorithm;
import si.fri.algotest.global.ErrorStatus;

public abstract class SubIsoAbsAlgorithm extends AbsAlgorithm
    {

 SubIsoTestCase subIsoTestCase;

 @Override
 public ErrorStatus init(TestCase test) {
     if (test instanceof SubIsoTestCase) {
         subIsoTestCase = (SubIsoTestCase) test;
         return ErrorStatus.STATUS_OK;
     } else {
         return ErrorStatus
                 .setLastErrorMessage(ErrorStatus.
                    ERROR_CANT_PERFORM_TEST,
                        "Invalid test:" + test);
     }
 }

 @Override
 public void run() {
     // v polje isoCount testnega primera zapisemo rezultat
        izvedbe algoritma
     subIsoTestCase.isoCount = execute(subIsoTestCase.
        targetGraph, subIsoTestCase.patternGraph);
 }

 @Override
 public ParameterSet done() {
     ParameterSet result = new ParameterSet(subIsoTestCase.
        getParameters());

     // stevilo najdenih izomorfizmov
     EParameter isoCount = new EParameter("IsoCount", "",
        ParameterType.INT,
             subIsoTestCase.isoCount);
     result.addParameter(isoCount, true);

         // pravilnost resitve (primerjava IsoCount z IsoNo)
     // OK - resitev pravilna
     // NOK - resitev nepravilna
```

```
    // N/A - resitev ni mogoce preveriti (ni podatka o
       pravem st. podgrafnih izomorfizmov)
    EParameter isoNo = subIsoTestCase.getParameters().
       getParamater("IsoNo");
    EParameter check = new EParameter("Check", "",
       ParameterType.STRING,
           (isoNo != null) && (int) isoNo.getField("Value")
               == -1 ? "N/A" : (isoNo != null) && (int)
              isoNo.getField("Value") == subIsoTestCase.
              isoCount ? "OK" : "NOK");
    result.addParameter(check, true);

    return result;
 }

 // podpis metode execute, ki jo implementiramo za vsak
    algoritem posebej
 protected abstract int execute(ArrayList<ArrayList<Integer>>
     g2, ArrayList<ArrayList<Integer>> g1);

}
```

Literatura

[1] (2015) W. D. Vogl, *Graph Matching Algorithms*. Dostopno na: http://oldwww.prip.tuwien.ac.at/teaching/ss/strupr/vogl.pdf

[2] J. R. Ullmann, "An Algorithm For Subgraph Isomorphism", *Journal of the Association for Computing Machinery*, 23(1), 1976, str. 31–42.

[3] N. Ramovš, "Problem izomorfnega podgrafa", Diplomsko delo, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Slovenija, 2013.

[4] (2015) G. Valiente, *Subgraph Isomorphism and Related Problems*. Dostopno na: http://www.cs.upc.edu/~valiente/riga-gra.pdf

[5] (2014) Homomorphism. Dostopno na: http://en.wikipedia.org/wiki/Homomorphism

[6] (2015) Graph (abstract data type). Dostopno na: http://en.wikipedia.org/wiki/Graph_(abstract_data_type)

[7] R. J. Wilson, J. J. Watkins, *Uvod v teorijo grafov*, Ljubljana, Društvo matematikov, fizikov in astronomov Slovenije, 1997.

[8] S. Zampelli, "A Constraint Programming Approach to Subgraph Isomorphism", Doktorska disertacija, Université catholique de Louvain, Département d'Ingénierie Informatique, Belgija, 2008.

[9] J. Mihelič, U. Čibej, "Izboljšave Ullmannovega algoritma za problem iskanja podgrafnih izomorfizmov", *Zbornik enaindvajsete mednarodne Elektrotehniške in računalniške konference ERK 2012*, Portorož, Slovenija, 2012.

[10] J. Mihelič, U. Čibej, "Improvements to Ullmann's Algorithm for the Subgraph Isomorphism Problem" (poslano v objavo).

[11] J. Mihelič, U. Čibej, "Search strategies for subgraph isomorphism algorithms", *Lecture Notes in Computer Science Volume 8321*, 2014, str. 77–88.

[12] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data", *BMC Bioinformatics 2013*, 14(7), 2013.

[13] P. Foggia, C. Sansone, M. Vento, "A Database of Graphs for Isomorphism and Sub-Graph Isomorphism Benchmarking", *CoRR*, 2001, str. 176–187.

[14] M. De Santo, P. Foggia, C. Sansone, M. Vento, "A large database of graphs and its use for benchmarking graph isomorphism algorithms", *Pattern Recognition Letters - Special issue: Graph-based representations in pattern recognition*, 24(8), 2003, str. 1067–1079.

[15] (2015) T. Dobravec, "ALGator - izvajanje in analiza algoritmov".
Dostopno na: `https://github.com/ALGatorDevel/Algator/`

[16] (2015) Clique (graph theory). Dostopno na:
`http://en.wikipedia.org/wiki/Clique_(graph_theory)`

[17] (2015) Isomorphic Graphs. Dostopno na:
`http://mathworld.wolfram.com/IsomorphicGraphs.html`

[18] B. Robič, *Aproksimacijski algoritmi*, Fakulteta za računalništvo in informatiko, 2008.

[19] (2015) Caliper - Microbenchmarking framework for Java. Dostopno na:
`https://code.google.com/p/caliper/`

[20] (2015) Perf4J. Dostopno na:
`http://perf4j.codehaus.org/`

[21] W. Fan, "Graph Pattern Matching Revised for Social Network Analysis", v zborniku *Proceedings of the 15th International Conference on Database Theory*, Berlin, Nemčija, marec 2012, str. 8–21.

[22] J. Lischka, H. Karl, "A Virtual Network Mapping Algorithm based on Subgraph Isomorphism Detection", v zborniku *Proceedings of the 1st ACM*

workshop on Virtualized infrastructure systems and architectures, Barcelona, Španija, avg. 2009, str. 81–88.

[23] W. S. Han, J. Lee, J. H. Lee, "Turbo$_{\text{ISO}}$: Towards UltraFast and Robust Subgraph Isomorphism Search in Large Graph Databases", v zborniku *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, ZDA, jun. 2013, str. 337–348.

[24] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10), 2004, str. 1367–1372.

[25] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, "Performance Evaluation of the VF Graph Matching Algorithm", v zborniku *Proceedings of the 10th International Conference on Image Analysis and Processing*, Benetke, Italija, sept. 1999, str. 1172–1177.

[26] J. Lee, W. S. Han, R. Kasperovics, J. H. Lee, "An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases", *Proceedings of the VLDB Endowment*, 6(2), 2013, str. 133–144.

[27] V. Lipets, N. Vanetik, E. Gudes, "Subsea: an efficient heuristic aglorithm for subgraph isomorphism", *Data Mining and Knowledge Discovery*, 19(3), 2009, str. 320–350.

[28] (2015) D. Conte, P. Foggia, C. Sansone, M. Vento, "Graph matching applications in pattern recognition and image processing".
Dostopno na:
`http://pdf.aminer.org/000/235/216/verification_of_engineering_models_based_on_bipartite_graph_matching_for.pdf`

[29] (2015) R. de Cássia Nandi, A. L. Pires Guedes, "Graph Isomorphism applied to Fingerprint Matching". Dostopno na:
`http://euler.mat.ufrgs.br/~trevisan/workgraph/regina.pdf`

[30] (2015) MIVIA - Laboratorio di Macchine Intelligenti per il riconoscimento di Video, Immagini e Audio. Dostopno na:
`http://mivia.unisa.it/`