Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Nejc Ramovš

Problem izomorfnega podgrafa

diplomsko delo
na univerzitetnem študiju

Mentor: prof. dr. Borut Robič

Ljubljana, 2013

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01857/2012
Datum: 03.09.2012

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

Kandidat: **NEJC RAMOVŠ**

Naslov: **PROBLEM IZOMORFNEGA PODGRAFA**
THE SUBGRAPH ISOMORPHISM PROBLEM

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Podajte širši pregled stanja na področju reševanja problema izomorfnosti podgrafov. Podrobneje opišite tri algoritme za reševanje omenjenega problema: Ullmannov algoritem, algoritem VF2 in algoritem Subsea. Preizkusite in primerjajte omenjene algoritme ali njihove različice s pomočjo standardne baze grafov.

Mentor:
prof. dr. Borut Robič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2.

Dekan:
prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Nejc Ramovš,

z vpisno številko 63070162,

sem avtor diplomskega dela z naslovom:

Problem izomorfnega podgrafa

S svojim podpisom zagotavljam, da:

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

V Ljubljani, 10. 3. 2013 Podpis avtorja:

Zahvaljujem se mentorju prof. dr. Borutu Robiču in asistentu dr. Urošu Čibeju za nasvete in napotke pri izdelavi diplomske naloge.

Še posebej se zahvaljujem moji zaročenki Blanki za veliko mero spodbude in razumevanja ter staršem za podporo skozi vsa leta študija.

Kazalo

Povzetek

V diplomski nalogi smo predstavili problem iskanja izomorfnih podgrafov. To je ena najbolj osnovnih operacij nad grafih in je NP-težek problem. Podrobno smo opisali Ullmannov algoritem in algoritem VF2, ki se na tem področju največ uporabljata, ter nov algoritem Subsea. Algoritem VF2 smo izboljšali z idejami iz algoritma Subsea. Izboljšavo smo implementirali v programskem jeziku C++ in dobljeni algoritem eksperimentalno primerjali z navadnim algoritmom VF2 iz programske knjižnice vflib in z izboljšanim Ullmannovim algoritmom. Delovanje smo preizkusili na bazi 9000 naključno generiranih parov testnih grafov. Ugotovili smo, da je izboljšan Ullmannov algoritem za faktor 5, izboljšan algoritem VF2 pa za faktor 30 hitrejši od navadnega VF2. Pri majhnih grafih je najhitrejši izboljšan Ullmannov algoritem, pri večjih grafih in grafih z več povezavami pa je hitrejši naš izboljšan algoritem VF2.

Ključne besede:
graf, podgraf, izomorfizem, algoritem

Abstract

This thesis describes the problem of finding subgraph isomorphism. This is one of the most basic operations performed on graphs and is an NP-hard problem. We describe in detail the Ullmann algorithm and VF2 algorithm, the most commonly used and state-of-the art algorithms in this field, and a new algorithm called Subsea. We improved the VF2 algorithm using some principles from Subsea. The improvement was implemented in C++ and then compared against VF2 algorithm implementation from the program library vflib and an implementation of improved Ullmann algorithm. We tested the algorithms on a database of 9000 pairs of randomly generated graphs. In compariton to the VF2 algorithm the results show a speedup factor of at least 5 for the improved version of Ullmann algorithm and a speedup factor of at least 30 for the improved version of VF2 algorithm. The improved version of Ullmann algorithm was the fastest algorithm for small graphs, while for large graphs and graphs with many connections out improved version of VF2 algorithm proved the fastest.

Key words:
graph, subgraph, isomorphism, algorithm

Poglavje 1

Uvod

Grafi se na veliko področjih uporabljajo za predstavitev različnih informacij. Ena najosnovneših operacij nad grafi je iskanje in prepoznavanje vzorcev. Najbolj splošna oblika je za dani vzorec poiskati ujemanje v večjem grafu – poiskati izomorfen podgraf.

Uporaba iskanja podgrafov sega na različna področja. V računalniškem vidu npr. opravimo dekompozicijo slike, razmerja med posameznimi deli pa predstavimo z grafi. Vzorci predstavljajo dekompozicije znanih objektov, ki jih z iskanjem lahko prepoznavamo v sliki [2]. V kemiji iščemo pojavitve posameznih kemijskih struktur v večjih molekulah [5], kar je primer iskanja po sicer majhnih grafih a v obširnih zbirkah grafov. Primera takih podatkovnih baz sta SMARTS in ZINK. V biologiji lahko iščemo kombinacije proteinov v proteinskih interakcijskih mrežah, ki lahko vsebujejo več tisoč vozlišč [9], primeri takih baz so DIP, BioGRID, STRING in ConsensusPathDB. Novejši in zelo aktualni problemi so podatkovno rudarjenje v spletnih in socialnih omrežjih, in uporaba v CAD aplikacijah.

Zlasti v računalniškem vidu se uporabljajo neeksaktni algoritmi, ki iščejo približke vzorcev in se ukvarjajo s podobnostjo in razdaljami med grafi, vendar se rešitve takih algoritmov ne približajo iskanju eksaktnega ujemanja. Obstajajo tudi variante iskanja v grafih s specifičnimi lastnostmi, npr. v drevesih, grafih z omejeno stopnjo, planarnih grafih in drugih. Nekateri od teh specifičnih problemov so rešljivi tudi v polinomskem času, na splošnih grafih pa je iskanje izomorfnih

podgrafov NP-težek problem.

Prvi algoritem, ki je omejil prostor preiskovanja v primerjavi s pregledovanjem vseh možnih stanj je bil Ullmannov algoritem [13]. Kljub starosti je še vedno eden najbolj znanih algoritmov in se precej uporablja. Kasnejši algoritem VF2 [2, 3, 4] bolje izkorišča informacije o že pregledanem delu grafa in danes velja za *de facto* standard na splošnem iskanju podgrafnih izomorfizmov. Opravljen je bil tudi poskus polinomskega iskanja [10]; če imamo fiksno bazo grafov, lahko iz njih generiramo odločitveno drevo, samo iskanje pa ima časovno zahtevnost $O(n^4)$. Vendar ima odločitveno drevo eksponentno velikost glede na število vozlišč in je metoda uporabna samo za zelo majhne grafe (nekaj 10 vozlišč), ko potrebujemo zelo hitro iskanje. Algoritem LAD rešuje problem na principu programiranja z omejitvami [12, 14]. Novejši algoritmi QuickSI, GADDI, GraphQL in SPath so bili pred kratkim primerjani v članku [8]. Algoritem Subsea [9] je še eden novejših in naj bi bil primeren zlasti za iskanje majhnih vzorcev v zelo velikih grafih. Širši pregled algoritmov za iskanje ujemanj v grafih ponuja članek [1].

V tem delu opisujemo in primerjamo eksaktne algoritme za splošne grafe. Omejili smo se na Ullmannov algoritem, algoritem VF2 in algoritem Subsea. Prva dva sta klasična algoritma, pri katerih preverimo možnost njunega izboljšanja, tretji algoritem pa kaže veliko praktično uporabnost v sodobnih problemih iskanja vseh podgrafnih izomorfizmih pri majhnih vzorcih in velikih ciljnih grafih.

Poglavje 2

Definicija problema

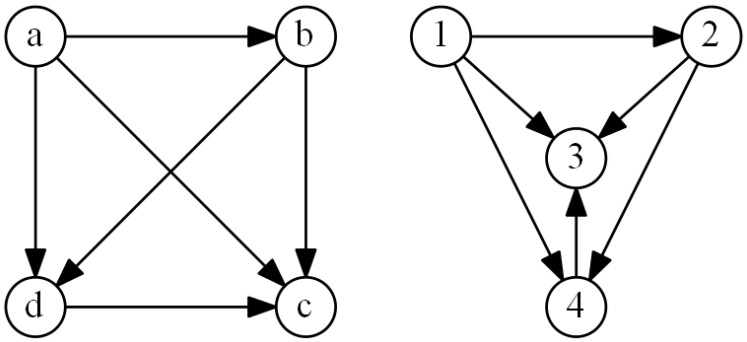
2.1 Graf

Graf $G = \langle V, E \rangle$ je definiran z množico vozlišč V (angl. vertices) in množico povezav E (angl. edges). Povezava je par vozlišč: $E \subseteq V \times V$. Vozlišči, ki sta vsebovani v povezavi, sta sosednji (angl. adjacent). Graf je lahko usmerjen (angl. directed) ali neusmerjen (angl. undirected). V neusmerjenem grafu je ena povezava neurejen par vozlišč $u, v \in V$ in jo označimo z $\{u, v\}$. V usmerjenem grafu je povezava urejen par vozlišč $u, v \in V$, kjer je prvo vozlišče začetek (angl. head), drugo pa konec (angl. tail) povezave. Označimo jo z (u, v). Graf z oznakami $G = (V, E, \alpha, \beta)$ sestavlja graf (V, E), funkcija $\alpha : V \to \mathbb{N}$, ki pripisuje oznako vozliščem, in funkcija $\beta : E \to \mathbb{N}$, ki pripisuje oznako povezavam.

2.2 Podgrafni izomorfizem

Graf $G' = \langle V', E' \rangle$ je podgraf danega grafa $G = \langle V, E \rangle$, če velja $V' \subseteq V \wedge E' \subseteq E$. Graf $G' = \langle V', E' \rangle$ je induciran podgraf danega grafa $G = \langle V, E \rangle$, če je podgraf grafa G in vsebuje vse povezave iz G, pri katerih sta robni vozlišči v V', oz. če velja $E' = E \cap (V' \times V')$.

Grafa $G_p = \langle V_p, E_p \rangle$ in $G_t = \langle V_t, E_t \rangle$ sta izomorfna, če obstaja bijektivna preslikava $f : V_p \to V_t$, da velja: $(a, b) \in E_p \Leftrightarrow (f(a), f(b)) \in E_t$; oznaka p pomeni



Slika 2.1: Primer izomorfnih grafov z bijektivno funkcijo $f = \{(a,1),(b,2),(c,3),(d,4)\}$.

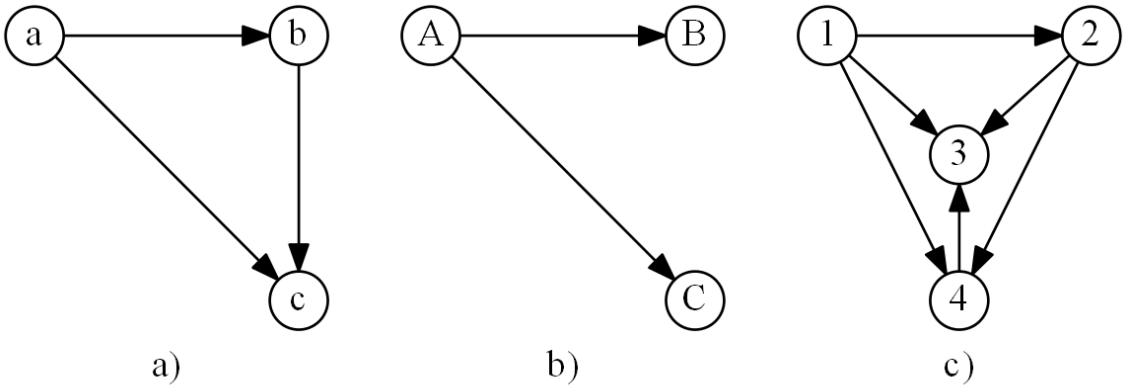
vzorčni (angl. pattern) graf, oznaka t pa ciljni (angl. target) graf. V primeru grafa z oznakami, mora preslikovalna funkcija ohranjati oznake. Primer izomorfizma grafov je podan na sliki 2.1.

Graf G_p je izomorfen podgraf grafa G_t, če obstaja podgraf G'_t grafa G_t, ki je izomorfen grafu G_p. Med grafoma G_p in G_t obstaja parcialen podgrafni izomorfizem, če je funkcija $f : V_p \to V_t$ injektivna in velja $(a,b) \in E_p \Rightarrow (f(a), f(b)) \in E_t$. Med grafoma G_p in G_t obstaja induciran podgrafni izomorfizem, če je funkcija $f : V_p \to V_t$ injektivna in velja $(a,b) \in E_p \Leftrightarrow (f(a), f(b)) \in E_t$. Na sliki 2.2 vidimo, da je med b) in c) samo parcialni podgrafni izomorfizem, ker v vzorčnem grafu ni povezave (B,C), medtem ko preslikavi teh vozlišč $(2,3)$ tvorita povezavo v c). Induciran podgrafni izomorfizem lahko obstaja samo, če se ohranijo tudi ne-povezave.

2.3 Problem izomorfnega podgrafa

Obstajajo štiri različice problema izomorfnega podgrafa.

- Odločitveni problem: ugotovi obstoj izomorfnega podgrafa.
- Preštevalni problem: ugotovi število izomorfnih podgrafov.
- Iskalni problem: poišči en izomorfen podgraf (preslikavo f).



Slika 2.2: Med a) in c) obstaja induciran podgrafni izomorfizem z injektivno funkcijo $f = \{(a,1),(b,2),(c,3)\}$. Med b) in c) obstaja samo parcialen podgrafni izomorfizem z injektivno funkcijo $f = \{(A,1),(B,2),(C,3)\}$.

- Naštevalni problem: poišči vse izomorfne podgrafe (preslikave f).

Najtežje je iskanje vseh možnih preslikav – to različico rešujejo opisani algoritmi. Dodatno lahko rešujemo problem še na usmerjenih ali neusmerjenih ter označenih ali neoznačenih grafih.

2.4 Druge definicije

Na tem mestu so zbrane pomožne definicije, ki jih uporabljamo pri opisih algoritmov za iskanje izomorfnih podgrafov.

V neusmerjenem grafu je število povezav, ki vsebujejo vozlišče v, stopnja (angl. degree) vozlišča v. Označimo jo z $d(v)$. Množica vozlišč $N(v) = \{u \in V \big| \{v,u\} \in E\}$ je množica sosedov vozlišča v. Množica sosednjih vozlišč podgrafa $S = (V_S, E_S)$ je definirana kot $N(S) = \{n \in V \setminus V_S \big| \exists m \in V_S : \{n,m\} \in E\}$.

V usmerjenem grafu je število povezav, ki imajo vozlišče v za začetek povezave, izhodna stopnja (angl. out-degree) vozlišča. Označimo jo z $d^+(v)$. Število povezav, ki imajo vozlišče v za konec povezave, je vhodna stopnja (angl. in-degree) vozlišča v. Označimo jo z $d^-(v)$. Množica naslednikov $N^+(v) = \{u \in V \big| (v,u) \in E\}$ so

končna vozlišča povezav, ki imajo za začetek povezave vozlišče v. Množica predhodnikov $N^-(v) = \{u \in V \big| (u,v) \in E\}$ so začetna vozlišča povezav, ki imajo za konec povezave vozlišče v. Množica izhodnih sosedov (angl. out-neighbors) podgrafa $S = (V_S, E_S)$ je definirana kot $N^+(S) = \{n \in V \setminus V_S \big| \exists m \in V_S : (m,n) \in E\}$. Množica vhodnih sosedov (angl. in-neighbors) podgrafa $S = (V_S, E_S)$ je definirana kot $N^-(S) = \{n \in V \setminus V_S \big| \exists m \in V_S : (n,m) \in E\}$.

Rez grafa G je razdelitev vozlišč V v dve disjunktni množici $(A, \bar{A})$. Množico povezav, kjer je en konec povezave v A in drugi v $\bar{A}$, označimo z $e(A, \bar{A}) = \{(u,v) \in E \big| u \in A, v \notin A\}$. Minimalna bisekcija grafa G je rez, ki minimizira velikost reza $|e(A, \bar{A})|$ po vseh množicah A velikosti $\lceil |V|/2 \rceil$.

2.5 Zahtevnost problema

Odločitveni problem izomorfnega podgrafa spada v razred NP-polnih problemov [7]. Za dokaz NP-polnosti nanj prevedemo odločitveni problem iskanja polnega podgrafa oz. klike (angl. clique problem). Pri slednjem, ki je znano NP-poln, imamo podan graf G in število k, preverjamo pa obstoj polnega podgrafa s k vozlišči (v polnem grafu obstaja povezava med vsakim parom vozlišč). Ta problem lahko rešimo z iskanjem izomorfnega podgrafa tako, da najprej generiramo poln graf H s k vozlišči, nato pa preverimo, če v G obstaja podgraf, ki je izomorfen H. Ker je problem iskanja polnega podgrafa NP-poln in smo ga v polinomskem času prevedli na problem iskanja izomorfnega podgrafa, je tudi slednji problem NP-poln.

Poglavje 3

Ullmannov algoritem

Prvi algoritem za iskanje podgrafnih izomorfizmov je leta 1976 predstavil J. R. Ullmann. Kljub starosti se algoritem še vedno množično uporablja in je tudi najbolj znan. Celoten princip preiskovanja temelji na predstavitvi z matrikami. V osnovni verziji algoritem ne upošteva lokalne strukture vzorčnega grafa, s čimer npr. druga dva opisana algoritma hitreje izločata neobetavne poti preiskovanja.

3.1 Predstavitev problema podgrafnega izomorfizma z matrikami

V tem algoritmu predstavimo grafe in problem reševanja podgrafnih izomorfizmov z matrikami. Matriki $P = [p_{i,j}] : i, j \in [1, n_p]$ in $T = [t_{i,j}] : i, j \in [1, n_t]$ sta matriki sosednosti grafov G_p in G_t, kjer vrednost 1 v vrstici i in stolpcu j pomeni, da v grafu obstaja povezava iz i v j, na primer:

$$p_{i,j} = \begin{cases} 1 & (i,j) \in E_p \\ 0 & \text{sicer.} \end{cases} \tag{3.1}$$

Matrika $M = [m_{i,j}] : i \in [1, n_p] \wedge j \in [1, n_t]$ predstavlja preslikavo $f : V_p \to V_t$. Vrednost 1 v vrstici i in stolpcu j pomeni preslikavo vozlišča $i \in V_p$ v $j \in V_t$:

$$m_{i,j} = \begin{cases} 1 & f(i) = j \\ 0 & \text{sicer.} \end{cases} \tag{3.2}$$

Matrika M je injektivna, če je v vsaki vrstici natanko ena 1 in v vsakem stolpcu največ ena 1. Samo preslikavo $f : V_p \to V_t$ lahko zapišemo v matrični obliki:

$$C = [c_{i,j}] = (M(MT)^T)^T, \tag{3.3}$$

kjer eksponent T pomeni operacijo transponiranja matrike. M predstavlja podgrafni izomorfizem iz G_p v G_t, če je dobljena matrika C enaka matriki sosednosti P:

$$c_{i,j} = p_{i,j} \quad \forall i, j \in [1, n_p] \tag{3.4}$$

Podana enačba velja za inducirani izomorfizem, pri enostavnem izomorfizmu je pogoj v enačbi 3.4 $p_{i,j} = 1 \Rightarrow c_{i,j} = 1$.

Ullmannov algoritem v bistvu generira različne matrike M in jih preverja s pogojem (3.4). Vseh možnih matrik M je $\frac{n_t!}{(n_t - n_p)!}$ (ob upoštevanju pogoja za injektivnost: natanko ena 1 v vrstici in največ ena 1 v stolpcu), kar je preveč za izčrpno preiskovanje. Algoritem zato začne s pred-procesirano matriko M^0 (razdelek 3.5), na vsakem koraku pa jo še dodatno omeji (razdelek 3.6).

3.2 Algoritem

Matriko M generiramo postopoma, po korakih. V danem koraku nam vrednost $m^k_{i,j}$ pove, ali se vozlišče $i \in V_p$ lahko preslika v $j \in V_t$ (ima vrednost 1) ali ne (ima vrednost 0). Izhajamo iz začetne matrike M^0. Pri generiranju te matrike upoštevamo dejstvo, da se lahko vozlišče i preslika v j samo, če je stopnja vozlišča

v vzorčnem grafu manjša ali enaka stopnji vozlišča v ciljnem grafu:

$$m^0_{i,j} = \begin{cases} 1 & d(i) \leq d(j) \\ 0 & \text{sicer.} \end{cases} \tag{3.5}$$

Pri usmerjenih grafih mora pogoj veljati tako za vhodno kot za izhodno stopnjo.

V vsakem naslednjem koraku izberemo še neobiskano vrstico. V vrstici izberemo stolpec, ki ima vrednost 1 in še ni bil izbran v nobeni od prejšnjih vrstic. Vse ostale vrednosti v vrstici postavimo na 0 in gremo v naslednji korak. Če stolpca z vrednostjo 1 v trenutni vrstici ni, se vrnemo v prejšnji korak in izberemo drug stolpec. Ko obdelamo vse vrstice, se preveri pogoj (3.4).

Algoritem je Ullmann v izvirnem članku [13] opisal s stavki GOTO, zaradi česar je težje razumljiv. Tukaj ga podajamo v bolj pregledni obliki, ki je popravljena in bolj podrobna različica algoritma iz [14]. Potrebujemo naslednje podatkovne strukture:

- spremenljivko d, ki označuje trenutno globino v preiskovalnem drevesu;
- spremenljivko k, ki označuje trenutno izbrani stolpec;
- binaren vektor $F = \langle F_1, \ldots, F_i, \ldots, F_{n_p} \rangle$, v katerem z vrednostjo 1 na i-tem mestu označimo, da je bil stolpec i že izbran – z njim zagotovimo injektivnost preslikave;
- vektor $H = \langle H_1, \ldots, H_d, \ldots, H_{n_p} \rangle$, kjer $H_d = j$ pomeni, da je na globini d izbran stolpec j – od tu obnovimo spremenljivko k pri sestopanju;
- matriko M, ki predstavlja trenutno matriko združljivih parov;
- vektor matrik $M_v = \langle M_1, \ldots, M_d, \ldots, M_{n_p} \rangle$, kjer je matrika M_d zadnja generirana matrika M na globini d – od tu obnovimo matriko M pri sestopanju.

Prevdokoda algoritma je podana v (Alg. 3.1). V vrsticah 1–2 inicializiramo vse spremenljivke. V vrstici 3 shranimo matriko M za prvi korak – matriko vedno shranjujemo pred vstopom v naslednji korak. Zanka v vrstici 4 se izteče, ko sestopimo iz obdelave prve vrstice, torej ko v prvi vrstici zmanjka neobiskanih

stolpcev. V vrstici 5 poskrbimo, da algoritem sestopi, če ne najdemo ustreznega stolpca. V vrstici 6 preverimo, če v trenutni vrstici obstaja še neobiskan stolpec. Ta stolpec mora biti še neizbran v trenutni vrstici ($j > k$), mora biti združljiv s trenutno vrstico ($m_{d,j} = 1$) in ne sme biti izbran v katerem od prejšnjih korakov ($F_j = 0$; pogoj za injektivno preslikavo). Sama izbira stolpca poteka v vrsticah 8–10, ko vrednost k postavimo na izbrani stolpec, v vrstici 7 pa preprečimo sestopanje, saj bomo v naslednji ponovitvi zanke ali povečali globino ali pa ostali na isti globini. V vrstici 11 postavimo vse neizbrane stolpce v vrstici na 0, torej izbrani stolpec označimo tudi v matriki M. Pogoj v vrstici 12 zaenkrat ignorirajmo. Če nismo prišli do dna preiskovalnega drevesa (vrstica 13), gremo v naslednji korak (vrstica 14): shranimo zadnji izbrani stolpec na trenutni globini ($H_d \leftarrow k$), stolpec označimo kot že izbranega ($F_k \leftarrow 1$), povečamo globino ($d \leftarrow d+1$), resetiramo izbiro stolpca za naslednji korak ($k \leftarrow 0$) in shranimo matriko M za naslednji korak ($M_d \leftarrow M$; od tukaj bomo obnavljali matriko M ob vračanju v tisti korak). Če smo obdelali že vse vrstice v matriki M, potem v vrstici 16 preverimo, če matrika ustreza pogoju (3.4), torej če matrika predstavlja podgrafni izomorfizem. Če pogoj drži, jo ustrezno shranimo ali izpišemo. V vrstici 18 obnovimo matriko M. Tako bomo v naslednji ponovitvi zanke preverili, če je v zadnji vrstici še kakšen primeren stolpec. Vrstici 19 in 20, tako kot vrstico 12, zaenkrat preskočimo. Vrstice 21–24 skrbijo za sestopanje, ki se izvede, če v trenutni vrstici ni nobenega primernega stolpca več. V vrstici 22 sprostimo zadnji izbrani stolpec ($F_k \leftarrow 0$) in znižamo globino. Nato v vrstici 24 obnovimo matriko M in v k obnovimo nazadnje izbrani stolpec na tej globini.

Velik del algoritma se ukvarja s sestopanjem in obnavljanjem stanja pri sestopanju, zato podajamo še rekurzivno različico algoritma (Alg. 3.2). V tej različici za sestopanje in obnavljanje stanja skrbi sam mehanizem rekurzija in je potek algoritma bolj očiten. V vrsticah 1 in 2 je inicializacija, le da sedaj potrebujemo manj spremenljivk. V vrstici 3 začnemo preiskovanje na globini 1. V vrstici 6 shranimo trenutno matriko M, ki jo bomo obnavljali ob sestopanju (ponovno zaenkrat ignoriramo del v oglatih oklepajih). V vrstici 7 je zanka, ki preveri vse stolpce. Ko pridemo do ustreznega (vrstica 8), vse ostale stolpce postavimo na 0 (vrstica

Algoritem 3.1 Ullmannov algoritem

Vhod: Matriki sosednosti P in T, začetna matrika M^0

Izhod: Vse $n_p \times n_t$ matrike M, ki predstavljajo preslikave podgrafnih izomorfizmov

```
1:  M ← M^0; d ← 1; H_1 ← 0; k ← 0; backtrack ← true
2:  for i ∈ [1, n_p] do F_i ← 0;
3:  M_1 ← M^0
4:  while d ≠ 0 do
5:      backtrack ← true
6:      if (∃j : j > k ∧ m_{d,j} = 1 ∧ F_j = 0) then
7:          backtrack ← false
8:          repeat
9:              k ← k + 1
10:         until m_{d,k} = 1 ∨ F_k = 1
11:         ∀j ≠ k : m_{d,j} ← 0
12:         if [ REFINE(M, P, T) ] then
13:             if d < n_p then
14:                 H_d ← k; F_k ← 1; d ← d + 1; k ← 0; M_d ← M
15:             else
16:                 if ⟨condition(3.4)⟩ then
17:                     ⟨store M⟩
18:                 M ← M_d
19:         else
20:             M ← M_d
21:     if backtrack then
22:         F_k ← 0; d ← d − 1;
23:         if d > 0 then
24:             M ← M_d; k ← H_d;
```

9) in označimo stolpec kot izbran (vrstica 10). Če smo v zadnji vrstici matrike M, potem preverimo pogoj 3.4, in če smo našli izomorfizem, shranimo trenutno matriko M (vrstice 11–13). Če to ni zadnja vrstica, gremo v naslednji korak (vrstica 15). Ob sestopanju obnovimo matriko M in označimo trenutni stolpec kot neizbran (vrstica 16).

Algoritem 3.2 Ullmannov algoritem - rekurzivna različica

Vhod: Matriki sosednosti P in T, začetna matrika M^0

Izhod: Vse $n_p \times n_t$ matrike M, ki predstavljajo preslikave podgrafnih izomorfizmov

```
 1: M ← M^0
 2: for i ∈ [1, n_p] do F_i ← 0;
 3: step(1)
```

```
procedure STEP(d)
 4: if [ !REFINE(M, P, T) ] then
 5:     return
 6: M_d ← M
 7: for all k ∈ [1, n_t] do
 8:     if m_{d,k} = 1 ∧ F_k = 0 then
 9:         ∀j ≠ k : m_{d,j} ← 0
10:         F_k ← 1
11:         if d = n_p then
12:             if ⟨condition(3.4)⟩ then
13:                 ⟨store M⟩
14:         else
15:             STEP(d + 1)
16:         F_k ← 0; M ← M_d
```

3.3 Omejevanje prostora preiskovanja

Ullmann je opazil, da lahko z dodatnim procesiranjem matrike M na vsakem koraku dodatno omejimo prostor preiskovanja, torej da več vrednosti 1 postavimo na 0. Če se bo vozlišče i preslikalo v vozlišče j, potem se mora tudi vsako sosednje vozlišče vozlišča i preslikati v eno od sosednjih vozlišč vozlišča j. Za vsak $m_{i,j} = 1$

preverimo pogoj:

$$\forall x \in N_p(i) \Rightarrow \exists y \in N_t(j) : m_{x,y} = 1 \tag{3.6}$$

oz. $$\forall x \in [1, n_p] : p_{i,x} = 1 \Rightarrow \exists y \in [1, n_t] : t_{j,y} = 1 \wedge m_{x,y} = 1$$

Če pogoj ni izpolnjen, postavimo vrednost $m_{i,j}$ na 0. Vsaka sprememba v matriki lahko vpliva na pogoj pri ostalih vozliščih, zato postopek ponavljamo, dokler pri pregledu celotne matrike ne naredimo nobene spremembe. Ta pogoj je hkrati tudi zadosten za preverjanje podgrafnega izomorfizma, zato lahko v algoritmih z njim nadomestimo pogoj (3.4).

Psevdokoda postopka je podana v (Alg. 3.3). Za vsak združljiv par (vrstica 4) preverimo pogoj (3.6) v vrstici 5. Če pogoj ni izpolnjen, označimo par kot nezdružljiv (vrstica 6). Poleg tega označimo, da bo celoten postopek potrebno ponoviti ($fixpoint \leftarrow false$). Ob spremembi v matriki M preverimo še, če trenutna vrstica sedaj vsebuje same 1 (vrstica 7). V tem primeru namreč izomorfizem ni več mogoč, zato vrnemo $false$ (vrstica 8). Za opisani postopek je Ullmann predlagal tudi implementacijo z namenskim logičnim vezjem [13].

Opisani algoritem je v (Alg. 3.1) in (Alg. 3.2) že vključen, klic funkcije $refine$ je v oglatih oklepajih, ki smo jih predhodno ignorirali.

Algoritem 3.3 Omejevanje prostora

Vhod: Matrika M in sosednostni matriki P in T

Izhod: $true$ če je bila M omejena, $false$ če kakšna vrstica ne vsebuje nobene 1

```
1:  procedure REFINE(M, P, T)
2:      repeat
3:          fixpoint ← true
4:          for ∀(i, j) : m_{i,j} = 1 do
5:              if condition (3.6) is not satisfied then
6:                  m_{i,j} ← 0; fixpoint ← false;
7:                  if ∀k : m_{i,k} = 0 then
8:                      return false
9:      until fixpoint
10:     return true
```

3.4 Časovna in prostorska zahtevnost

Prostorska zahtevnost celotnega algoritma je $O(n_p^2 n_t)$. Na vsaki globini shranimo matriko M, ki je velikosti $n_p n_t$, maksimalna globina pa je n_p. Časovna kompleksnost funkcije *refine* je $O(n_p n_t d_{max})$, kjer je d_{max} največja možna stopnja vozlišča. Pogoj preverjamo za vsak element matrike M, časovna zahtevnost pogoja pa je d_{max}, ker preverjamo samo sosede. Postopek preverjanja se sicer ponavlja do fiksne točke, ampak vsaka ponovitev dodatno poreže preiskovalno drevo, zato je ena ponovitev najslabši primer. Celoten algoritem ima časovno zahtevnost $O(n_p! n_p n_t d_{max})$. V posamezni ponovitvi zanke je funkcija *refine* najzahtevnejša, število ponovitev pa je reda $n!$.

3.5 Izboljšave

Ullmannov algoritem je v testih pokazal slabše rezultate od kasneje razvitih algoritmov [4]. Leta 2012 pa sta Jurij Mihelič in Uroš Čibej predlagala več možnih izboljšav algoritma [11].

V (Alg. 3.1) obiskujemo vozlišča iz vzorčnega grafa po vrsti, glede na sam zapis grafa z matriko. Drugačen vrsti red obiskovanja lahko pospeši algoritem, če čim hitreje ustavi preiskovanje poddreves, ki nimajo rešitve. Nekaj možnih hevristik:

- Najprej vozlišča z večjo stopnjo – taka vozlišča imajo običajno manj kandidatov za preslikavo.
- Preiskovanje v širino, znotraj iste globine pa po stopnji.
- Najboljši najprej – začnemo z vozliščem z največjo stopnjo, v vsakem naslednjem koraku pa med sosedi že obiskanih vozlišč izberemo vozlišče z največjo stopnjo.

Omejevanje prostora iskanja s funkcijo *refine* izkorišča dejstvo, da mora za vsakega soseda vozlišča i obstajati združljiv sosed vozlišča j. Podobno pa velja, če še i preslika v j, potem sosedje vozlišča i ne morejo biti združljivi z vozlišči, ki

niso sosedje vozlišča j:

$$\forall x \in N_p(i) \;\; \forall y \notin N_t(j) : m_{x,y} = 0 \tag{3.7}$$

Pri iskanju induciranega podgrafnega izomorfizma pa velja tudi v nasprotno smer:

$$\forall y \in N_t(j) \;\; \forall x \notin N_p(i) : m_{x,y} = 0 \tag{3.8}$$

Ta postopek lahko za razliko od $refine$ uporabimo samo za tiste (i, j), za katere vemo, da bo $m_{i,j}$ obdržala vrednost 1. V (Alg. 3.1) bi ga uporabili v vrstici 11 nad $m_{d,k}$. Postopek zmanjša prostor preiskovanja za sosednja vozlišča, kar lahko dobro izkoristita zadnji dve hevristiki iz prejšnjega odstavka. Pri njiju bodo namreč sosednja vozlišča hitro na vrsti za preiskovanje.

Izboljšati je mogoče tudi prostorsko zahtevnost. Osnovni algoritem namreč na vsakem koraku shrani celotno matriko M. Namesto sklada matrik lahko uporabimo persistentno matriko. Ob spremembi vrednosti v matriki iz 1 v 0 na sklad shranimo koordinate spremembe. Ob sestopanju iz sklada preberemo koordinate sprememb in na ustreznih mestih vrednost povrnemo nazaj v 1. Ker je sprememb kvečjemu $n_p n_t$, se prostorska zahtevnost zmanjša na $O(n_p n_t)$.

Poglavje 4

Algoritem VF2

Algoritem VF2 [2, 4] je novejši algoritem iz leta 2000. Izomorfizem išče iz delne rešitve, ki jo postopoma gradi z dodajanjem sosednjih vozlišč. Številka 2 v imenu pomeni posodobljeno verzijo, s katero so zmanjšali prostorsko zahtevnost [3].

Cilj algoritma VF2 je zgraditi preslikavo $M = \{(n, m) \in N_p \times N_t\}$. Medtem ko Ullmannov algoritem generira in preveri vse možne preslikave M, jih ta algoritem gradi postopoma. V vsakem stanju s algoritma imamo parcialno preslikavo $M(s)$. Ta definira podgrafa $G_p(s) \subseteq G_p$ in $G_t(s) \subseteq G_t$, ki sta si izomorfna in vsebujeta tista vozlišča, ki so tudi v $M(s)$. Uporabljali bomo oznaki $M_p(s)$ in $M_t(s)$, ki predstavljata množico vozlišč v omenjenih podgrafih. Prehod iz stanja s v naslednje stanje s' predstavlja razširitev trenutne $M(s)$ z dodatnim parom vozlišč. Nov par (n, m) izberemo med sosedi G_p in G_t tako, da tudi razširjena $M(s')$ definira izomorfna podgrafa.

Psevdokoda okvirnega poteka algoritma je podana v (Alg. 4.1). Algoritem je rekurziven in preiskuje z globino. Ob prvem klicu je $M(s_0)$ prazna. V vrstici 2 preverimo, če $M(s)$ že pokriva celoten vzorčni graf. Ker $M(s)$ po konstrukciji predstavlja izomorfizem, smo v tem primeru že našli podgrafni izomorfizem med G_p in G_t, zato M izpišemo. V vrstici 5 izračunamo vse možne kandidate za razširitev $M(s)$, kjer kandidate izbiramo med sosedi trenutnih podgrafov. Podrobnejši opis je v razdelku 4.1. V vrsticah 6–9 za vsak par preverimo, če je združljiv. Pri tem uporabljamo pravila, ki so opisana v razdelku 4.2. Če je par ustrezen, z njim

razširimo $M(s)$ (s čimer dobimo s') in rekurzivno kličemo algoritem nad novim stanjem.

Algoritem 4.1 Algoritem VF2

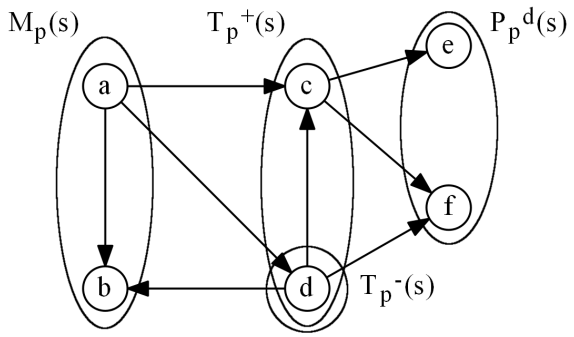
Vhod: Vmesno stanje s, začetno stanje s_0 ima $M(s_0) = \emptyset$
Izhod: Vse preslikave podgrafnih izomorfizmov

```
1: procedure VF2(s)
2:     if M(s) covers all G_p then
3:         output M(s)
4:     else
5:         Generate candidates P(s)
6:         for all p ∈ P(s) do
7:             if feasible(p) then
8:                 Compute s' by adding p to M(s)
9:                 VF2(s')
10:    restore data structures
```

4.1 Izbira kandidatov

Množico kandidatov $P(s)$ izbiramo med neposrednimi sosedi podgrafov $G_p(s)$ in $G_t(s)$. Naj bosta $T_p^+(s)$ in $T_t^+(s)$ množici vozlišč iz G_p in G_t, ki še niso v preslikavi $M(s)$ in so izhodni sosedi katerega izmed vozlišč v $M(s)$. Podobno naj bosta $T_p^-(s)$ in $T_t^-(s)$ množici vozlišč, ki še niso v preslikavi $M(s)$ ter imajo izhodne sosede v $M(s)$. Množico $P(s)$ sestavljajo pari (n, m), kjer velja $n \in T_p^+(s) \wedge m \in T_t^+(s)$. Če takega para ni, se upoštevata množici T_p^- in T_t^-. Če tudi takega para ni, upoštevamo vsa vozlišča, ki še niso v preslikavi $M(s)$. Slednji primer se zgodi, če je graf sestavljen iz več nepovezanih delov in čisto na začetku, ko je $M(s)$ prazna. $P(s)$ torej postane ena od naslednjih množic:

- $T^+(s) = \{min\ N^+(M_p(s))\} \times N^+(M_t(s))$;
- $T^-(s) = \{min\ N^-(M_p(s))\} \times N^-(M_t(s))$, če je $T^+(s) = \emptyset$;
- $P^d(s) = \{min\ (V_p \setminus (M_p(s) \cup T_p(s)))\} \times (V_t \setminus (M_t(s) \cup T_t(s)))$, če je $T^-(s) = \emptyset$.



Slika 4.1: Primer razporeditve vozlišč po množicah med izvajanjem algoritma VF2.

Primer opisanih množic je na sliki 4.1. Če bo algoritem iz trenutnega stanja prišel do konca (torej bo našel podgrafni izomorfizem), mora obstajati preslikava za vsak element iz vzorčnega grafa. Zato je dovolj, da v parih, ki jih vstavimo v množico $P(s)$, nastopa samo eno vozlišče iz vzorčnega grafa. Če algoritem ne bo našel podgrafnega izomorfizma z uporabo tega vozlišča, ga ne bo niti z nobenim drugim. Zato iz vzorčnega grafa vzamemo vedno samo najmanjši element, kar v enačbah predstavlja oznaka min. Na tak način se tudi izognemo generiranju enakih stanj. Podobno razmišljanje nas privede do zaključka, da lahko prekinemo trenutno pot preiskovanja, če je velikost množice T_t^+ manjša od velikosti T_p^+ (in isto za T_t^-, T_p^-). Omenimo še, da v sami implementaciji algoritma ni potrebe po eksplicitnem generiranju $P(s)$, naslednji par izračunamo na podlagi prejšnjega.

4.2 Izračun združljivosti kandidatov

Pri preverjanju posameznega para (n, m) iz $P(s)$ upoštevamo pet pravil. Z enačbama 4.1 in 4.2 preverimo, če bo razširjena parcialna preslikava še vedno predstavljala izomorfizem obdelanih podgrafov. Vsak sosed vozlišča iz vzorčnega grafa v parcialni preslikavi se mora preslikati v soseda vozlišča iz ciljnega grafa. Pogoj ločimo za vhodne in izhodne sosede. V oglatih oklepajih je zapisan še obratni

pogoj, ki mora veljati pri iskanju induciranega podgrafa.

$$\begin{array}{c}(\forall n' \in N_p^-(n) \cap M_p(s)\ \exists m' \in N_t^-(m)\ :\ (n', m') \in M(s)) \\ [\ \wedge\ (\forall m' \in N_t^-(m) \cap M_t(s)\ \exists n' \in N_p^-(n)\ :\ (n', m') \in M(s))\ ]\end{array} \tag{4.1}$$

$$\begin{array}{c}(\forall n' \in N_p^+(n) \cap M_p(s)\ \exists m' \in N_t^+(m)\ :\ (n', m') \in M(s)) \\ [\ \wedge\ (\forall m' \in N_t^+(m) \cap M_t(s)\ \exists n' \in N_p^+(n)\ :\ (n', m') \in M(s))\ ]\end{array} \tag{4.2}$$

Ta pogoj je zadosten za pravilno preiskovanje, z dodatnimi pogoji pa dodatno skrčimo prostor preiskovanja. Preverjamo število sosedov v posameznih množicah. Prejšnji enačbi že preverjata sosede iz množice $M(s)$. Z enačbama 4.3 in 4.4 preverjamo število sosedov v množici $T(s)$. Posebej preverimo za $T^-(s)$ in $T^+(s)$ ter za vhodne in izhodne sosede. Število sosedov mora biti v vzorčnem grafu manjše ali enako številu sosedov v ciljnem grafu.

$$\begin{array}{rcl}|N_p^-(n) \cap T_p^-(s)| & \leq & |N_t^-(m) \cap T_t^-(s)| \wedge \\ |N_p^+(n) \cap T_p^-(s)| & \leq & |N_t^+(m) \cap T_t^-(s)|\end{array} \tag{4.3}$$

$$\begin{array}{rcl}|N_p^-(n) \cap T_p^+(s)| & \leq & |N_t^-(m) \cap T_t^+(s)| \wedge \\ |N_p^+(n) \cap T_p^+(s)| & \leq & |N_t^+(m) \cap T_t^+(s)|\end{array} \tag{4.4}$$

Enačba 4.5 preveri še število sosedov, ki niso v množici $M(s)$ ali $T(s)$, torej sosede iz množice $P^d(s)$.

$$\begin{array}{rcl}|N_p^-(n) \setminus (M_p(s) \cup T_p(s))| & \leq & |N_t^-(n) \setminus (M_t(s) \cup T_t(s))| \wedge \\ |N_p^+(n) \setminus (M_p(s) \cup T_p(s))| & \leq & |N_t^+(n) \setminus (M_t(s) \cup T_t(s))|\end{array} \tag{4.5}$$

Pri grafih z oznakami enačbi 4.1 dodamo še pogoj kompatibilnosti oznak povezav, oznake vozlišč pa lahko preverjamo že pri izgradnji množice $P(s)$.

4.3 Časovna in prostorska zahtevnost

Za hitro izvajanje algoritma in čim manjšo porabo prostora je pomembna ustrezna izbira podatkovnih struktur. Implementacija avtorjev algoritma poleg po-

datkovnih struktur za hrambo grafov uporablja šest vektorjev. Parcialno preslikavo $M(s)$ predstavljata vektorja `core_p` in `core_t`. Če $(n, m) \in M(s)$, potem ima `core_p[n]` vrednost m in `core_t[m]` vrednost n. V nasprotnem primeru imata oba vrednost *null*. Množice T_p^-, T_p^+, T_t^-, T_t^+ predstavimo z vektorji `in_p`, `out_p`, `in_t`, `out_t`. Vrednost `in_p[n]` je pozitivna, če velja $n \in T_p^- \ \vee \ n \in M_p(s)$, drugače je *null*. Podobno so definirane tudi ostale množice. Za ekskluzivno pripadnost vozlišča n množici T_p^- mora veljati `in_p[n] > 0` in `core_p[n] == null`.

Omenjene vektorje si vsa stanja preiskovanja delijo. Velja namreč, da ob prehodu v novo stanje zapise v vektorjih samo dodajamo oz. jim spreminjamo vrednosti iz *null* v neko pozitivno vrednost. Ob sestopanju jih lahko zato obnovimo. Obema vektorjema `core` spremenimo vsakemu samo eno vrednost in si to vrednost zapomnimo. Pri ostalih vektorjih je ob spremembi pomembno samo, da imajo vrednost večjo od 0. Lahko jim damo vrednost, ki ustreza trenutni globini preiskovanja. Ob sestopanju izbrišemo vrednosti s trenutno globino. Pri tem tudi ni potreben obhod celotnega vektorja, ampak je zadosti, da preverimo samo sosede izbranega para. Samo za ta vozlišča je namreč bila sprememba možna. Tako ni potrebe po hranjenju kopije vektorja pri vsakem stanju. Ker velja $n_t \geq n_p$, je torej prostorska zahtevnost celotnega algoritma $O(n_t)$.

V najslabšem primeru bo algoritem moral zgenerirati vse možna stanja, ki jih je reda $n_p!$. Vsako stanje mora narediti troje:

- Preveriti, če držijo pravila iz razdelka 4.2. To je možno narediti z enim prehodom po vseh sosedih izbranega para, vse potrebne operacije pa se zaradi izbranih podatkovnih struktur izvedejo v konstantnem času. Časovna zahtevnost tega dela je torej $O(d_p(n) + d_t(m))$, kjer sta n in m izbran par.

- Izračunati nove množice T_p^-, T_p^+, T_t^-, T_t^+. Tudi to lahko opravi z enim prehodom po vseh sosedih – nastavlja vrednost na tistih indeksih, ki vrednosti še nimajo. Časovna zahtevnost je ravno tako $O(d_p(n) + d_t(m))$.

- Generirati množico parov, ki so kandidati za vključitev v $M(s)$. Iz vzorčnega grafa vzame prvi element, ki v ustreznem vektorju (`in_p`, `out_p`) še nima

vrednosti. V množico doda še vse elemente iz komplementarne množice, ki jih dobi z enim prehodom ustreznega vektorja. Časovna zahtevnost je torej $O(n_t)$

Ker sta d_p in d_t navzgor omejena z n_t, je skupna časovna zahtevnost $O(n_p!n_t)$

Poglavje 5

Algoritem Subsea

Algoritem Subsea [9] je najnovejši izmed predstavljenih algoritmov. Deluje po principu deli in vladaj:

1. Generiramo vse zgodovine pregledov za vzorčni graf (razdelek 5.2).

2. Ciljni graf razbijemo z bisekcijo (aproksimacijski algoritem - razdelek 5.1).

3. Na vsaki povezavi med obema deloma bisekcije preverimo obstoj podgrafnega izomorfizma z uporabo zgodovin pregledov (razdelek 5.3).

4. Na vsakem delu bisekcije ponovimo postopek od koraka 2 naprej; končamo, ko ima posamezen del bisekcije manj vozlišč kot vzorčni graf.

Pri samem iskanju uporabi hevristiko, ki poskuša čim hitreje najti negativne primere in končati iskanje v taki veji.

Zaradi predprocesiranja se najbolje obnese pri iskanju vseh primerkov podgrafov. Za razliko od prvih dveh algoritmov ima Subsea nekaj omejitev. Dobro deluje predvsem, če je vzorčni graf mnogo manjši od ciljnega, poleg tega pa mora biti vzorčni graf povezan. Ker oboje drži za večino praktičnih problemov, omejitve niso posebno omejujoče.

5.1 Bisekcija grafa

Algoritem Subsea med delovanjem razdeli graf na dve enako veliki množici (particiji) – naredi bisekcijo. Algoritem deluje hitreje, če je bisekcija minimalna, torej če je število povezav med obema deloma grafov minimalno. Soroden problem je problem minimalnega reza, le da pri njem ni pomembno število elementov v particijah. Problem minimalne bisekcije je NP-težek, vendar pravilnost algoritma Subsea ni odvisna od natančnosti minimalne bisekcije, zato se uporabljajo hitre aproksimacijske metode.

Prva taka metoda je algoritem "črnih lukenj"(Black holes bisection). Začne s praznima particijama B_1 in B_2. Potem v vsako particijo izmenično dodaja po eno vozlišče. Vozlišče izbira med sosednjimi vozlišči trenutne particije. Torej v B_1 doda naključno vozlišče iz $V \setminus (B_1 \cup B_2)$, ki ima soseda v B_1. Podobno tudi za drugo particijo. Če takega vozlišča ni, doda naključno še ne izbrano vozlišče. Algoritem temelji na dejstvu, da, če sta particiji ("črni luknji") trenutno na nasprotnih straneh minimalne bisekcije, bo več sosedov z iste strani minimalne bisekcije in posledično večja verjetnost, da dodamo ustrezno vozlišče. Psevdokoda je podana v (Alg. 5.1).

Druga metoda je požrešni algoritem. Ta začne z že obstoječo bisekcijo, ki pa je lahko kar rezultat prve metode, ter jo lokalno optimizira. Za vsako vozlišče izračuna notranjo ceno $I(x)$, ki je število sosedov vozlišča x znotraj iste particije, in zunanjo ceno $E(x)$, ki je število sosedov v drugi particiji. Potem za vsak par $x \in B$, $y \in \bar{B}$ izračuna spremembo cene reza ob morebitni zamenjavi para: $gain = E(x) - I(x) + E(y) - E(y) - 2w(x, y)$, kjer $w(x, y) = 1$, če sta vozlišči povezani, in $w(x, y) = 0$, če vozlišči nista povezani (w je korekcija morebitne medsebojne povezave – slednja je zunanja in za razliko od ostalih ostane zunanja tudi po zamenjavi). Algoritem nato menja pare vozlišč z največjo pridobitvijo $gain$, pri vsaki zamenjavi pa popravi $I(v')$ in $E(v')$ za vsakega soseda zamenjanega para. Ustavi se, ko ni več para z $gain > 0$.

Algoritem 5.1 Bisekcija grafa - "črne luknje"

Vhod: Graf $G = (V, E)$
Izhod: Rez $(B, \bar{B})$, ki je aproksimacija minimalne bisekcije

1: $B_1 \leftarrow B_2 \leftarrow \emptyset$
2: $B_0 \leftarrow V \setminus (B_1 \cup B_2)$
3: **repeat**
4: ADD2HOLE(1)
5: ADD2HOLE(2)
6: **until** $B_0 = \emptyset$

procedure ADD2HOLE(i)
7: **if** $B_0 = \emptyset$ **then return**
8: $E_0 \leftarrow \{(u, v) : u \in B_i, v \in B_0\}$
9: **if** $E_0 \neq \emptyset$ **then**
10: **choose randomly** $e = (u, v) \in E_0 : v \in B_0$
11: **else**
12: **choose randomly** $v \in B_0$
13: $B_i \leftarrow B_i \cup \{v\}$
14: $B_0 \leftarrow B_0 \setminus \{v\}$

5.2 Zgodovina pregleda grafa

Algoritem Subsea med iskanjem podgrafnega izomorfizma ne uporablja vzorčnega grafa, ampak zgodovino pregleda. Ta enolično določa vrstni red obiskovanja vozlišč. Motivacija za tak pristop je, da v fazi predprocesiranja določimo vrstni red preiskovanja, ki bi čim hitreje ugotovil, da izomorfizem ni mogoč.

Zaporedno številko obiskovanja označuje preslikava $d : V \to \mathbb{N}$. Oznaka l_i je oznaka tistega vozlišča, ki smo ga obiskali kot i-tega zaporednega, torej $l_i = l(v) : d(v) = i$. Definiramo še $N_i = \{d(u) < i : u \in N_G(v) \wedge d(v) = i\}$, ki za i-to vozlišče predstavlja njegove sosede, ki smo jih že obiskali. Zgodovina pregleda grafa je zaporedje $\langle (l_1, N_1), (l_2, N_2), \ldots (l_{|V|}, N_{|V|}) \rangle$. To zaporedje bomo kasneje uporabili za preverjanje izomorfizma v ciljnem grafu. Vozlišča bomo preiskovali v vrstnem redu, ki ga določa zgodovina pregleda v vzorčnem grafu. Za vsako obiskano vozlišče iz ciljnega grafa bomo preverili, če ustreza oznaki vozlišča iz zgodovine pregleda (ima enak l_i) in če struktura trenutnega podgrafa ustreza strukturi vzorčnega grafa

(N_i vzorčnega grafa mora biti podmnožica N_i ciljnega grafa oziroma morata biti množici enaki v primeru iskanja induciranega podgrafa). Če bi imeli usmerjene grafe, bi morali hraniti ločeni množici N_i za vhodne in izhodne povezave.

Psevdokoda algoritma, ki izračuna zgodovino pregleda z začetkom v dveh podanih vozliščih, je v (Alg. 5.2). Deluje po principu preiskovanja v globino, pri čemer naslednje vozlišče izbere na podlagi hevristike. V vrsticah 1–4 inicializiramo potrebne podatkovne strukture in obiščemo prvo vozlišče. V vrstici 6 obiskanemu vozlišču določimo zaporedno številko. V vrstici 7 gradimo samo zgodovino pregleda grafa. V vrstici 7 zagotovimo, da kot drugega obiščemo vozlišče v_2. Začetek na vozliščih v_1, v_2 je potreben, ker moramo zgenerirati zgodovino pregleda za vsako povezavo v vzorčnem grafu. V vrsticah 9–13 obiščemo vsa naslednja vozlišča. Kandidati so sosedi trenutnega vozlišča (vrstica 9). Vrstni red določa hevristika (vrstica 11). Najboljšega kandidata rekurzivno obiščemo (preiskovanje v globino, vrstica 12) in odstranimo s seznama kandidatov (13).

Hevristika daje prednost vozliščem, ki so tesno povezani z že pregledanimi vozlišči. Funkcija za izračun hevristike vrača par koordinat. Prva predstavlja število korakov, ki jih potrebujemo iz kandidata w do poljubnega že pregledanega vozlišča, pri čemer jasno ne smemo uporabiti povezave (v, w). Druga koordinata predstavlja število že obiskanih vozlišč, ki jih s toliko koraki lahko dosežemo (to število pomnožimo z -1). Med kandidati izberemo tistega z minimalnim parom koordinat, torej tistega, ki potrebuje najmanj korakov (je najbližje že pregledanim vozliščem), med izenačenimi pa tistega, preko katerega pridemo do največjega števila že obiskanih vozlišč. Koordinati izračunamo s preiskovanjem v širino. Začnemo z množico, v kateri je samo kandidat w, dolžina poti pa je 1 (vrstici 14 in 15). Nato v zanki povečujemo globino. V vrstici 17 izračunamo vse sosede trenutne množice S (in ignoriramo omenjeno povezavo (w, v)). V vrstici 18 preštejemo, koliko že pregledanih vozlišč smo dosegli. Če smo dosegli vsaj eno, vrnemo rezultat v vrstici 19, sicer povečamo korak (vrstici 20 in 21). Če ni več novih sosedov (vrstica 22), pomeni, da iz w ni mogoče doseči že pregledanih vozlišč, razen preko v. Take kandidate bomo pregledali nazadnje, zato prvo koordinato nastavimo na ∞, drugo pa na velikost množice vozlišč, dosegljivih iz w.

Algoritem 5.2 Zgodovina pregleda grafa (angl. Traverse history)

Vhod: Graf $G = (V, E)$, začetni vozlišči $v_1, v_2 \in V$, da velja $(v_1, v_2) \in E$
Izhod: Zgodovina prehoda H z začetkom v v_1, v_2

1: **for all** $v \in V$ **do**
2: $\quad d(v) \leftarrow 0$
3: $vtime \leftarrow 1$
4: VISIT(v_1)
5: **return** H

procedure VISIT(v)
6: $d(v) \leftarrow vtime$
7: $H[vtime++] \leftarrow (l(v), \{0 < d(u_1) \leq \ldots \leq d(u_m) : u_1, \ldots, u_m \in N_G(v)\})$
8: **if** $v = v_1$ **then** VISIT(v_2)
9: $N_0 \leftarrow \{u \in N_G(v) : d(u) = 0\}$
10: **while** $N_0 \neq \emptyset$ **do**
11: $\quad$ **choose** $w \in N_0$ with miminal ESTIMATENEXT(w, v)
12: $\quad$ **if** $d(w) = 0$ **then** VISIT(w)
13: $\quad N_0 \leftarrow N_0 \setminus \{w\}$

procedure ESTIMATENEXT(w, v)
14: $S \leftarrow \{w\}$
15: $len \leftarrow 1$
16: **repeat**
17: $\quad N_S \leftarrow \cup_{z \in S} N_{G \setminus (v,w)}(z)$
18: $\quad p \leftarrow |\{y \in N_S : d(y) > 0\}|$
19: $\quad$ **if** $p > 0$ **then return** $\langle len, -p \rangle$
20: $\quad len++$
21: $\quad S \leftarrow S \cup N_S$
22: **until** $N_S \neq \emptyset$
23: **return** $\langle \infty, |S| \rangle$

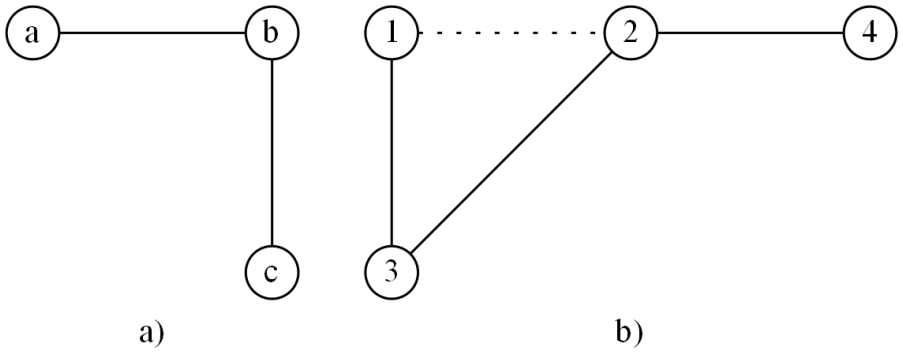
5.3 Iskanje izomorfnega podgrafa

Ko poznamo zgodovino pregleda za ciljni graf, jo uporabimo pri iskanju podgrafnega izomorfizma. Psevdokoda algoritma je podana v (Alg. 5.3). Iskanje sprožimo nad konkretno povezavo v ciljnem grafu (glej razdelek 5.4). Deli v oglatih oklepajih veljajo za primer iskanja induciranega podgrafa. V vrsticah 1–4 inicializiramo preslikavo g (v ciljnem grafu uporabljamo oznako g namesto d, da se ju ne meša) in definiramo vrednosti v preslikavi za podani vozlišči. Tako omejimo preiskovanje v ciljnem grafu na samo tiste podgrafe, ki vsebujejo podano povezavo. V vrstici 5 je prvi klic rekurzivne funkcije, ki bo od sedaj naprej sledila vrstnemu redu preiskovanja iz podane H.

V funkciji $searchVisit$ naprej dobimo vrednost zadnje zaporedne številke iz g in ustrezno vozlišče (zadnje dodano vozlišče, vrstici 6 in 7). Za to vozlišče v vrstici 8 preverimo, če ustreza vozlišču iz vzorčnega grafa z isto zaporedno številko, torej če je dodajanje tega vozlišča ohranilo izomorfizem trenutno pregledanega dela grafa. Veljati mora, da so vsa vozlišča v trenutni N_i iz zgodovine pregleda H vsebovana med pregledanimi sosedi dodanega vozlišča (primerjamo njihove preslikave iz d oz. g). Če to ne velja, tudi podgrafnega izomorfizma ni, zato sestopimo. Za primer iskanja induciranega podgrafa morata biti omenjeni množici enaki (vrstica 10). Pri usmerjenih grafih pa bi ločeno preverjali množici vhodnih in izhodnih sosedov. V vrstici 12 v trenutni preiskani podgraf dodamo še povezave, ki pripadajo prej dodanemu vozlišču. Povezave potrebujemo, da lahko vrnemo celoten izomorfen podgraf, ko smo ga našli (vrstica 13).

V drugem delu funkcije $searchVisit$ najdemo in pregledamo vse kandidate za naslednje dodano vozlišče. Iz $H[vtime + 1].N$ vemo, s katerimi že pregledanimi vozlišči mora biti naslednje vozlišče povezano. Obratno torej velja, da so kandidati tista vozlišča, ki so sosednja vsem vozliščem iz $H[vtime + 1].N$ (vrstica 14). V vrsticah 17–20 te kandidate preverimo: če kandidat še ni bil obiskan, mu v preslikavi g določimo naslednjo zaporedno številko in obiščemo novo stanje. Ob vrnitvi razveljavimo spremembo v g.

V vrstici 17 je še dodaten pogoj, ki mora veljati v primeru iskanja induciranih podgrafov. Kot bomo videli v naslednjem razdelku, med procesom iskanja vseh



Slika 5.1: Prikaz potrebe po "črnih"robovih. Graf a) je vzorčni, b) pa ciljni. Pri obdelavi povezave $(1, 2)$ najdemo inducirani izomorfen podgraf $\langle 1, 2, 4\rangle$. Če povezavo odstranimo, bi kasneje lahko našli še izomorfen podgraf $\langle 1, 3, 2\rangle$, kar bi bilo narobe – ob odstranjeni povezavi $(1, 2)$ ne bi mogli ugotoviti, da podgraf ni induciran. Kljub temu moramo povezavo nekako označiti, da med obdelavo povezave $(2, 4)$ ne bi ponovno našli grafa $\langle 1, 2, 4\rangle$.

podgrafnih izomorfizmov brišemo že obdelane povezave, s čimer preprečimo, da bi kasneje v procesu iskanja ponovno našli isti podgraf. Pri induciranih podgrafih pa povezave ne smemo brisati, ampak jo označimo kot "črno"(primer, zakaj je to potrebno, je na sliki 5.1). Zato moramo pri kandidatu preveriti, da nima nobene "črne"povezave na že pregledano vozlišče.

Odkrite podgrafne izomorfizme hranimo v seznamu S. Preslikavo med vozlišči vzorčnega in ciljnega grafa opravimo s kompozitumom $d_t^{-1} \circ d_p$.

Algoritem 5.3 Iskanje izomorfnega podgrafa

Vhod: Graf $G_t = (V_t, E_t)$, začetni vozlišči $v'_1, v'_2 \in V_t$, zgodovina prehoda H grafa G_p, [množica "črnih"povezav *Black*]

Izhod: Vsi podgrafi [ali inducirani podgrafi, ki ne vsebujejo "črnih"povezav] grafa G_t, ki so $(v_1 \to v'_1, v_2 \to_v 2')$-izomorfni z G_p in sta v_1, v_2 začetni vozlišči H

1: **for all** $v' \in V_t$ **do**
2: $\quad g(v') \leftarrow 0$
3: $g(v'_1) \leftarrow 1$
4: $g(v'_2) \leftarrow 2$
5: **return** SearchVisit$(\{v'_1, v'_2\}, \emptyset)$

procedure SearchVisit(V', E')
6: $vtime \leftarrow |V'|$
7: $v' \leftarrow g^{-1}(vtime)$
8: **if** $H[vtime].N \not\subseteq \{g(u') > 0 : (v', u') \in E\}$ **or** $H[vtime].l \neq l(v')$ **then**
9: $\quad$ **return** $false$
10: **if** [$H[vtime].N \neq \{g(u') > 0 : (v', u') \in E\}$ **or** $H[vtime].l \neq l(v')$] **then**
11: $\quad$ **return** $false$
12: $E' \leftarrow E' \cup \{(u', v') : d(u') \in H[vtime].N\}$
13: **if** $|H| = vtime$ **then return** $\{(V', E')\}$
14: $L \leftarrow \cap\{N_{G_t}(u) : u \in H[vtime + 1].N\}$
15: $S \leftarrow \emptyset$
16: **for** $w \in L$ **do**
17: $\quad$ **if** $g(w) = 0$ [**and** $\forall v \in N_{G_t}(w) : g(v) > 0 \Rightarrow (v, w) \notin$ "*Black*"] **then**
18: $\quad\quad g(w) \leftarrow vtime + 1$
19: $\quad\quad S \leftarrow S \cup$ SearchVisit$(V' \cup \{w\}, E')$
20: $\quad\quad g(w) \leftarrow 0$
21: **return** S

5.4 Celoten algoritem

V tem razdelku povežemo vse spoznane algoritme v celoto, kot je opisano na začetku tega poglavja. Psevdokoda je podana v (Alg. 5.4). V fazi predprocesiranja za vsako povezavo generiramo in shranimo zgodovino pregleda vzorčnega grafa z začetkom v tej povezavi (vrstice 1–4). Pri tem uporabimo (Alg. 5.2). Nato vstopimo v rekurzivni del. V vrstici 7 z algoritmom za bisekcijo (Alg. 5.1) razdelimo graf na dva dela. Nato preverimo vsako povezavo med obema deloma bisekcije. Tu vidimo, zakaj algoritem deluje hitreje, če je število povezav med particijama čim manjše, torej če je bisekcija čim bliže minimalni. Na posamezni povezavi poženemo algoritem za iskanje podgrafnih izomorfizmov (Alg. 5.3) v kombinaciji z vsako generirano zgodovino pregleda (vrstica 10), s čimer najdemo vse podgrafne izomorfizme, ki vsebujejo to povezavo. Zato lahko po končanem iskanju to povezavo odstranimo (oziroma jo označimo kot "črno" v primeru iskanja induciranih podgrafov). Ko preverimo vse povezave, kličemo algoritem rekurzivno nad posameznim delom bisekcije, torej nad problemom s polovičnim ciljnim grafom. Algoritem se konča, ko je v ciljnem grafu manj vozlišč kot v vzorčnem (vrstica 6).

Algoritem Subsea največ prostora porabi za hranjenje vseh zgodovin pregleda vzorčnega grafa. Ker ustvarimo eno za vsako povezavo (pravzaprav dve: eno za H_{v_1,v_2} in drugo za H_{v_2,v_1}), jih je skupno n_p^2. Vsaka ima n elementov, N_i komponenta posameznega elementa (ki vsebuje vse sosede) pa je lahko tudi velikosti n. Skupno torej $O(n_p^4)$. Iz česar vidimo, da je algoritem res specializiran za iskanje manjših vzorčnih grafov v velikih ciljnih grafih. Algoritem, kot je opisan v izvirnem članku [9] sicer izloča zgodovine pregleda, ki predstavljajo avtomorfizem vzorčnega grafa, kar nekoliko zmanjša potreben prostor, hkrati pa pospeši izvajanje algoritma. Za odkrivanje avtomorfizmov se uporabi kar (Alg. 5.3).

Algoritem 5.4 Glavni algoritem Subsea

Vhod: Vzorčni graf $G_p = (V_p, E_p)$, ciljni graf $G_t = (V_t, E_t)$
Izhod: Vsi podgrafi [ali inducirani podgrafi] grafa G_t, ki so izomorfni grafu G_p

1: $A \leftarrow \emptyset$
2: **for** $\langle v_1, v_2 \rangle \in V_p \times V_p : (v_1, v_2) \in E_p$ **do**
3: **run** (Alg. 5.2) on $G_p, v_1, v_2 \rightarrow$ traverse history H_{v_1,v_2}
4: $A \leftarrow A \cup \{H_{v_1,v_2}\}$
5: SubIso(A, G_t)

procedure SubIso(A, G_t)
6: **if** $|V_p| > |V_t|$ **then return** $\emptyset$
7: **run** (Alg. 5.1) on $G_t \rightarrow$ bisection $(B, \bar{B})$
8: **for** $(v_1, v_2) \in (B, \bar{B})$ **do**
9: **for** $H \in A$ **do**
10: **run** (Alg. 5.3) on $G_t, v_1, v_2, H \rightarrow$ set of subgraphs S
11: $G \leftarrow G \setminus (v_1, v_2)$ ["$Black$" $\leftarrow$ "$Black$" $\cup (v_1, v_2)$]
12: **return** $S \cup$ SubIso$(G_t(B)) \cup$ SubIso$(G_t(\bar{B}))$

Poglavje 6

Eksperimentalna primerjava algoritmov

Delovanje algoritmov smo preizkusili na 9000 primerih. Iskali smo število vseh induciranih podgrafnih izomorfizmov v usmerjenih grafih. Implementacija algoritma VF2 je dostopna na spletu (http://mivia.unisa.it/datasets/graph-database/vflib-graph-matching-library-version-2-0/) v obliki programske knjižnice vflib. Za osnovni Ullmannov algoritem je že bilo pokazano, da ne konkurira algoritmu VF2 [4], zato smo primerjali implementacijo Jurija Miheliča in Uroša Čibeja z opisanimi izboljšavami iz razdelka 3.5.

Avtorji članka o algoritmu Subsea so na spletu objavili implementacico tega algoritma (http://www.cs.bgu.ac.il/∼orlovn/links/SubseaProj30_11_06/). Vendar ta implementacija ne deluje pravilno, saj celo za isti testni primer vrača različne rezultate (torej različno število vseh podgrafnih izomorfizmov). Zaradi kompleksnosti algoritma pa bi ga bilo težko implementirati v celoti. Namesto tega smo iz algoritma vzeli idejo zgodovine preiskovanja (razdelek 5.2) in jo uporabili v algoritmu VF2. Slednji namreč pri izbiranju naslednjega vozlišča za preiskovanje ne izkorišča zgradbe preiskovanega grafa. Kot smo opisali v razdelku 4.1, algoritem med ne-preiskanimi sosedi že preiskanih vozlišč izbere prvo vozlišče oz. tistega z najmanjšim indeksom (pri tem sicer upošteva še pripadnost množicam T^+ in T^-). Naša ideja pa je namesto poljubnega vrstnega reda uporabiti vrstni red,

ki ga izračuna (Alg. 5.2). Motivacija je enaka kot pri algoritmu Subsea: najprej preveriti bolj skoncentrirane dele vzorčnega grafa, da bi s tem čim hitreje odkrili neobetavne veje preiskovanja. Opisani algoritem za zgodovino pregleda začne s podanima vozliščema, ker algoritem Subsea zahteva izračun zgodovine pregleda za vsako povezavo. Pri izboljšavi algoritma VF2 te zahteve ni, zato začnemo z vozliščem z največjo stopnjo.

Naša implementacija algoritma za izračun zgodovine pregleda je v dodatku A.1. Napisana je v jeziku C++ in je namenjena vključitvi v izvorno kodo programske knjižnice vflib, tako smo zagotovili nepristransko primerjavo z obstoječim algoritmom VF2. Razred `SearchTraverse` ima javno metodo `SortNodesBy-SearchTraverse`. Ta sprejme kazalec na graf (uporablja se graf iz knjižnice vflib), vrne pa urejeno tabelo vozlišč. Za integracijo izboljšave je v konstruktorju razreda za algoritem VF2 (datoteka `vf2_sub_state.cc`) potrebno klicati omenjeno metodo in shraniti tabelo vozlišč. Potem je v isti datoteki potrebno popraviti metodo `NextPair`, da upošteva izračunani vrstni red vozlišč. Vozlišče iz vzorčnega grafa dobi neposredno iz tabele izračunanega vrstnega reda preiskovanja, za indeks v tabeli uporabi trenutno globino preiskovanja (spremenljivka `core_len` v obstoječi implementaciji). Za to vozlišče preveri pripadnost množicam T^+ in T^-, da lahko vozlišča iz ciljnega grafa izbira v isti množici. Popravljena metoda je v dodatku A.2.

6.1 Baza testnih grafov

Algoritme smo testirali na bazi generiranih grafov [6], ki so jo ustvarili avtorji algoritma VF2. Dostopna je na internetnem naslovu http://mivia.unisa.it/datasets/graph-database/browse-db/. Celotna baza vsebuje 72.800 parov usmerjenih grafov (po en vzorčni in en testni graf), namenjenih testiranju algoritmov za iskanje izomorfnih grafov in podgrafov. Omejili smo se na naključno generirane grafe brez posebne strukture, ki jih je 9.000 parov. Ti se najprej ločijo po relativni velikosti vzorčnega grafa glede na ciljni graf. Oznake $si2$, $si4$ in $si6$ predstavljajo teste za iskanje podgrafnih izomorfizmov v velikosti 20 %, 40 % in 60 % ciljnega grafa.

Nadalje se testi ločijo po številu povezav. Oznake $r001$, $r005$ in $r01$ pomenijo, da je med vsakim parov vozlišč verjetnost obstoja povezave 0.01, 0.05 ali 0.10. Če je verjetnost povezave η, bo v grafu skupno $\eta N(N-1)$ povezav. V primeru, da je število povezav premajhno, da bi bil nastali graf povezan, se dodaja povezave, dokler graf ni povezan. Opisane delitve razbijejo testne primere v devet množic s 1000 pari. V vsaki množici je po 100 primerov testov, ki imajo v ciljnem grafu 20, 40, 60, 80, 100, 200, 400, 600, 800 in 1000 vozlišč.

6.2 Rezultati

Algoritme smo poganjali na računalniku s procesorjem Intel Core2 Duo P8600 2.4 GHz in operacijskim sistemom Linux. Vsa koda je bila napisana v programskem jeziku C++ in prevedena s parameterom –O3. Čas izvajanja na posameznem testnem primeru smo omejili na 60 sekund. Rezultati so podani v grafični obliki na slikah 6.1–6.6.

Grafi na slikah 6.1–6.3 predstavljajo število rešenih primerov v odvisnosti od časa izvajanja algoritma, torej koliko primerov je algoritem uspel rešiti v določenem času. Algoritem je boljši, če ima večjo površino pod krivuljo. Opazimo, da sta si na manjših vzorčnih grafih in na grafih z manj povezavami izboljšani Ullmannov algoritem in izboljšani algoritem VF2 enakovredna, oba pa sta boljša od osnovnega algoritma VF2. Ko pa se vzorčni graf povečuje in dobiva več povezav, izboljšani algoritem VF2 prehiti tudi izboljšanega Ulmannovega. Grafi na desni polovici imajo os x v logaritemskem merilu, da se bolje vidi obnašanje algoritmov pri krajšem času izvajanja.

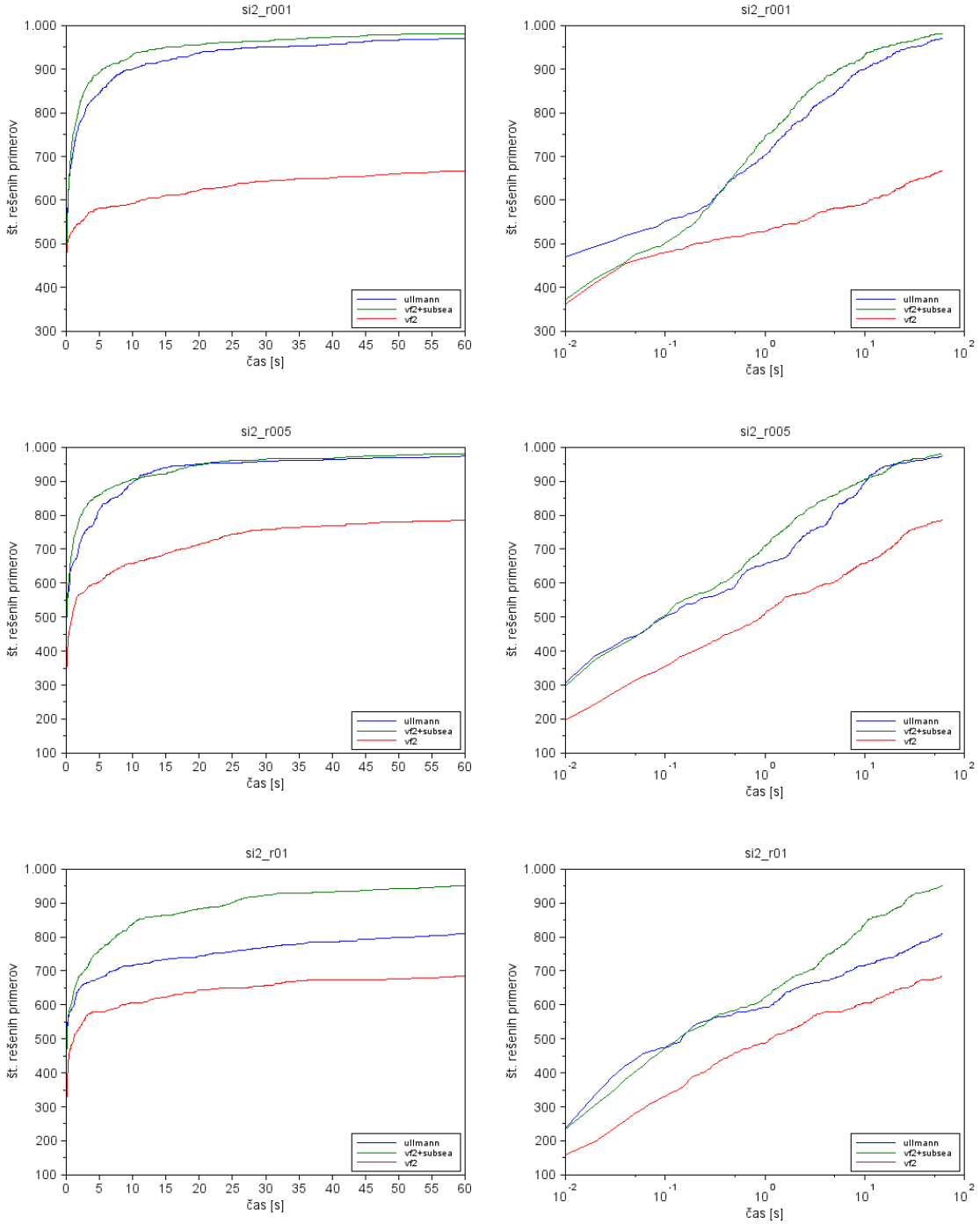
Grafi na slikah 6.4–6.6 predstavljajo povprečen čas izvajanja algoritma v odvisnosti od velikosti problema, tu so razlike med algoritmi bolj očitne. Grafi na desni polovici imajo v logaritemskem merilu obe osi, da so vidni rezultati za vsako velikost grafov posebej. Vidimo, da je na grafih z malo vozlišči boljši izboljšan Ullmannov algoritem, VF2 in izboljšani VF2 pa sta si enakovredna. Slednje je razumljivo, saj gre v osnovi za isti algoritem; na majhnih grafih je pospešitev majhna, ker ni bolj gostih delov grafa, izračun vrstnega reda pa vzame nekaj

Tabela 6.1: Časi trajanja izvajanja algoritmov po posameznih množicah testov. Ull+ pomeni izboljšan Ullmannov algoritem, VF2+ pa izboljšan algoritem VF2.

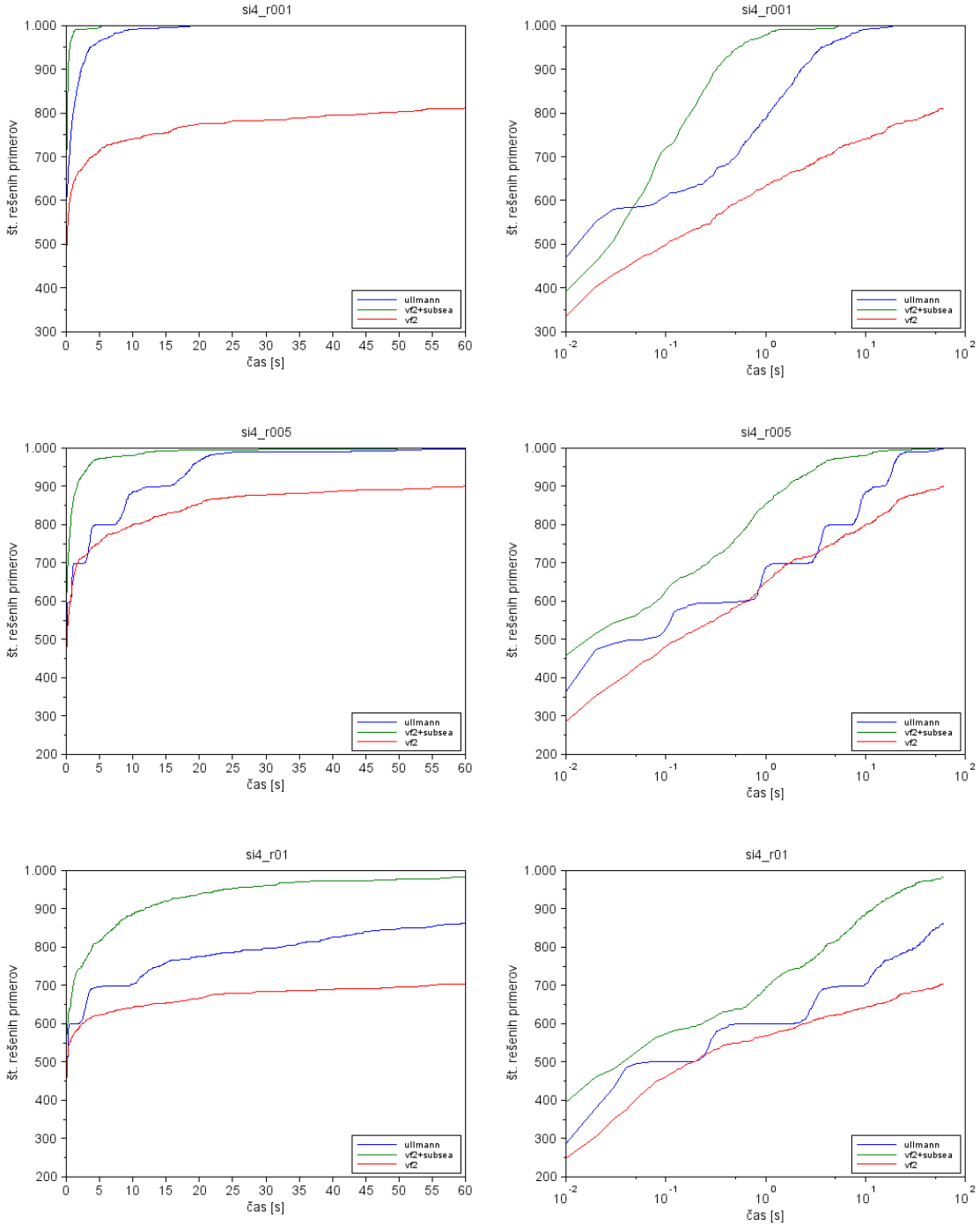
Grafi	Rešeni primeri (%)			Celoten čas (min)			Faktor pospešitve (v primerjavi z VF2)	
	Ull+	VF2+	VF2	Ull+	VF2+	VF2	Ull+	VF2+
si2_r001	99,6	100,0	76,3	12,4	6,7	269,6	22	40
si2_r005	99,9	100,0	91,4	8,1	7,4	134,0	17	18
si2_r01	95,5	99,9	79,4	103,4	16,4	249,5	2	15
si4_r001	100,0	100,0	89,8	1,1	0,2	135,7	123	678
si4_r005	100,0	100,0	97,0	5,0	1,3	54,9	11	42
si4_r01	98,1	100,0	79,8	68,0	6,3	244,5	4	38
si6_r001	100,0	100,0	95,7	1,1	0,1	71,3	65	713
si6_r005	100,0	100,0	100,0	5,2	0,6	13,8	3	23
si6_r01	99,2	100,0	81,5	65,8	3,5	220,4	3	63
skupno	99,14	99,98	87,88	270,1	42,5	1393	5	32

časa. S povečevanjem števila vozlišč je izboljšani Ullmannov algoritem še vedno boljši od VF2, izboljšani VF2 pa se najprej približa izboljšanemu Ullmannovenu in ga na koncu prehiti. Podobno kot pri prvih treh slikah lahko ugotovimo, da s povečevanjem števila povezav in povečevanjem relativne velikosti vzorčnega grafa izboljšani algoritem VF2 hitreje pridobiva prednost; npr. pri si6_r01 je izboljšani Ullmannov algoritem hitrejši samo pri grafih z 20 vozlišči.

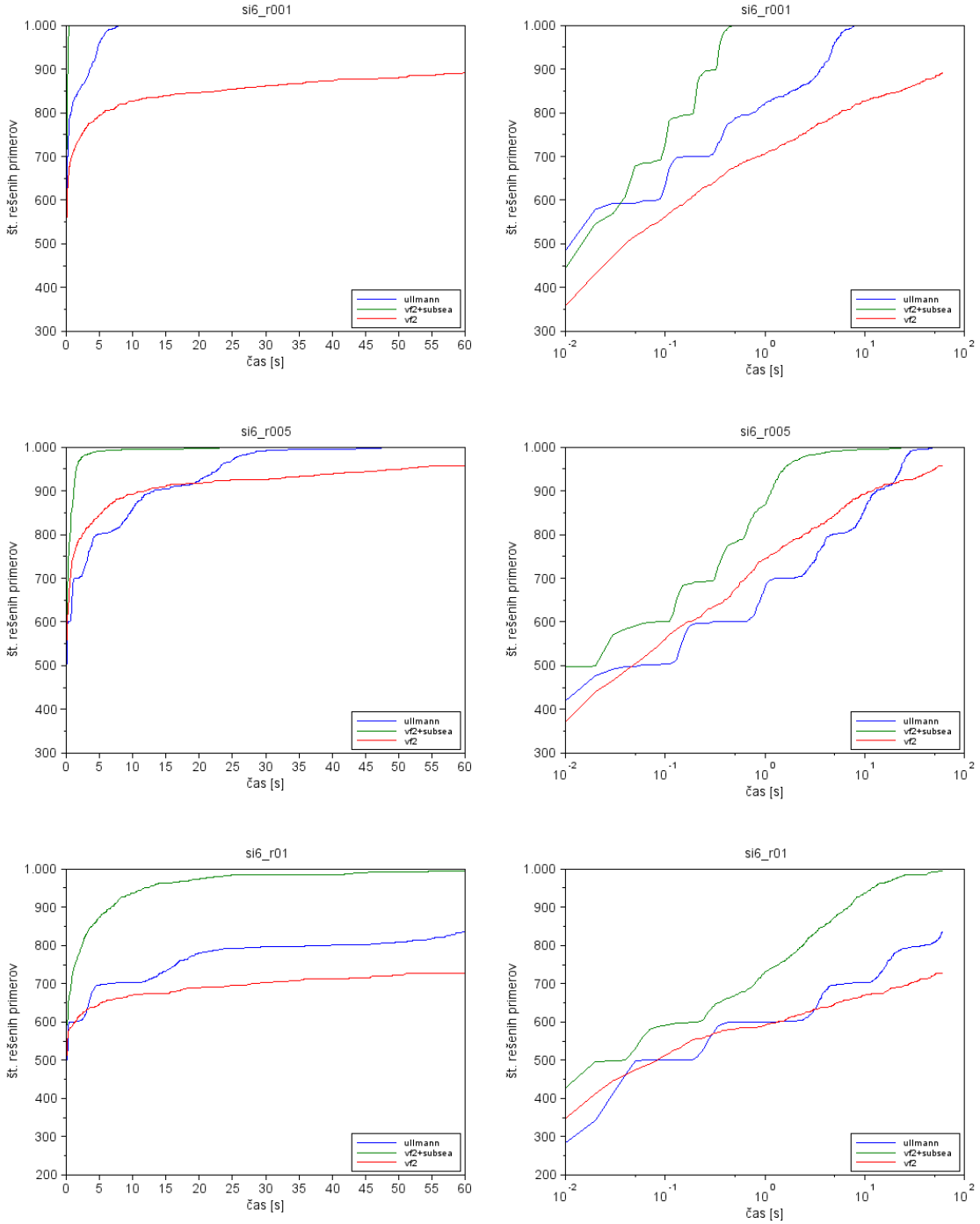
Kot je že bilo omenjeno, smo izvajanje posameznega testa omejili na 60 sekund. V računanju povprečnega časa izvajanje smo za testne primere, ki jih algoritmi niso končali, upoštevali čas 60 sekund. Že s to omejitvijo sta izboljšani Ullmannov algoritem in izboljšanji algoritem VF2 vsaj za velikostni razred hitrejša od osnovnega algoritma VF2 in pri večjih grafih še izboljšani VF2 za velikostni razred hitrejši od izboljšanega Ullmannovega algoritma. Ob daljšem času poganjanja algoritmov bi bila prednost pred osnovnim algoritmov VF2 še večja. To se na grafih vidi kot negativen drugi odvod krivulje za osnovni algoritem VF2, ker je navzgor omejena z vrednostjo 60 sekund. Časi izvajanja algoritmov po posameznih množicah so podani v tabeli 6.1.



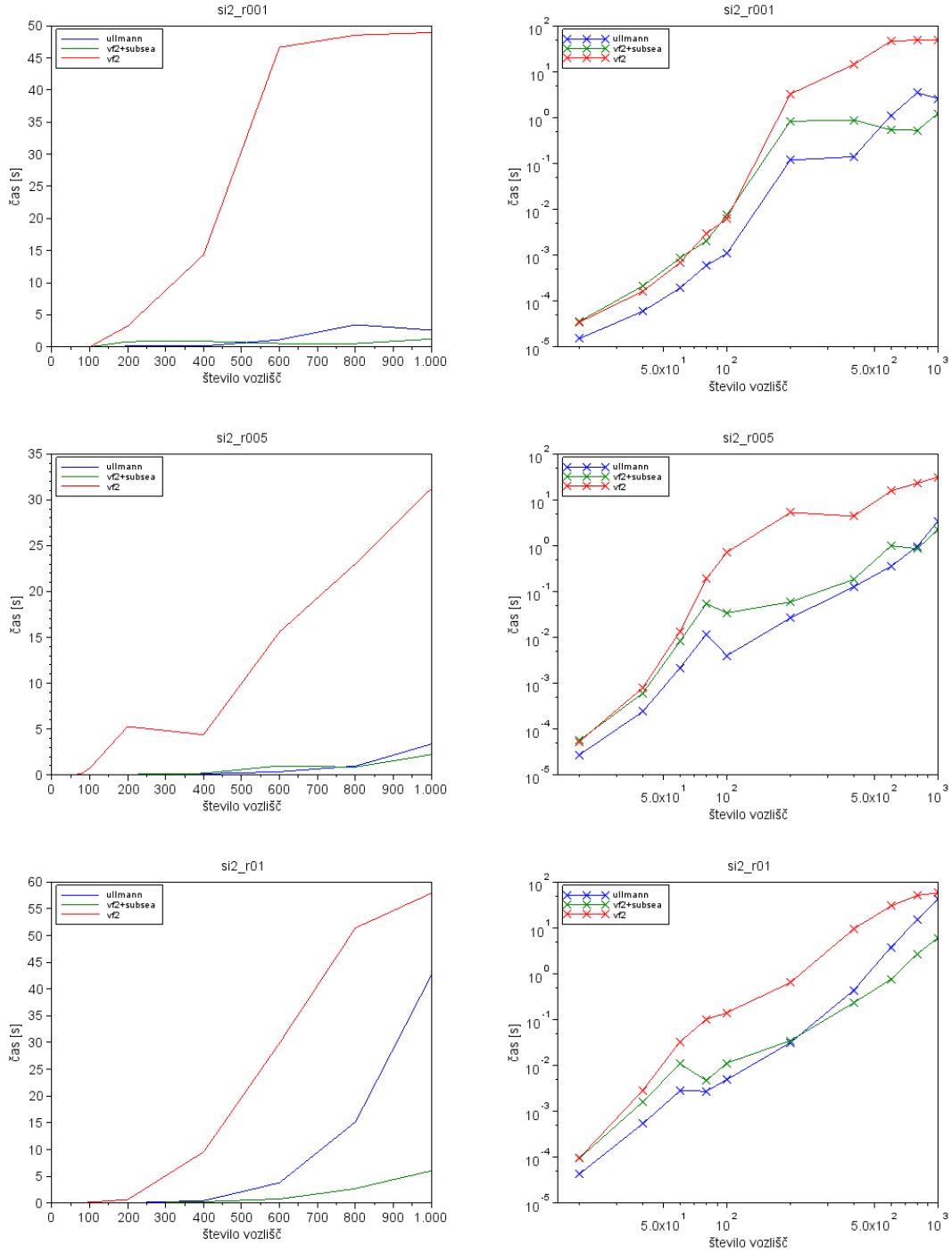
Slika 6.1: Število rešenih primerov za grafe tipov si2_r001, si2_r005 in si2_r01.



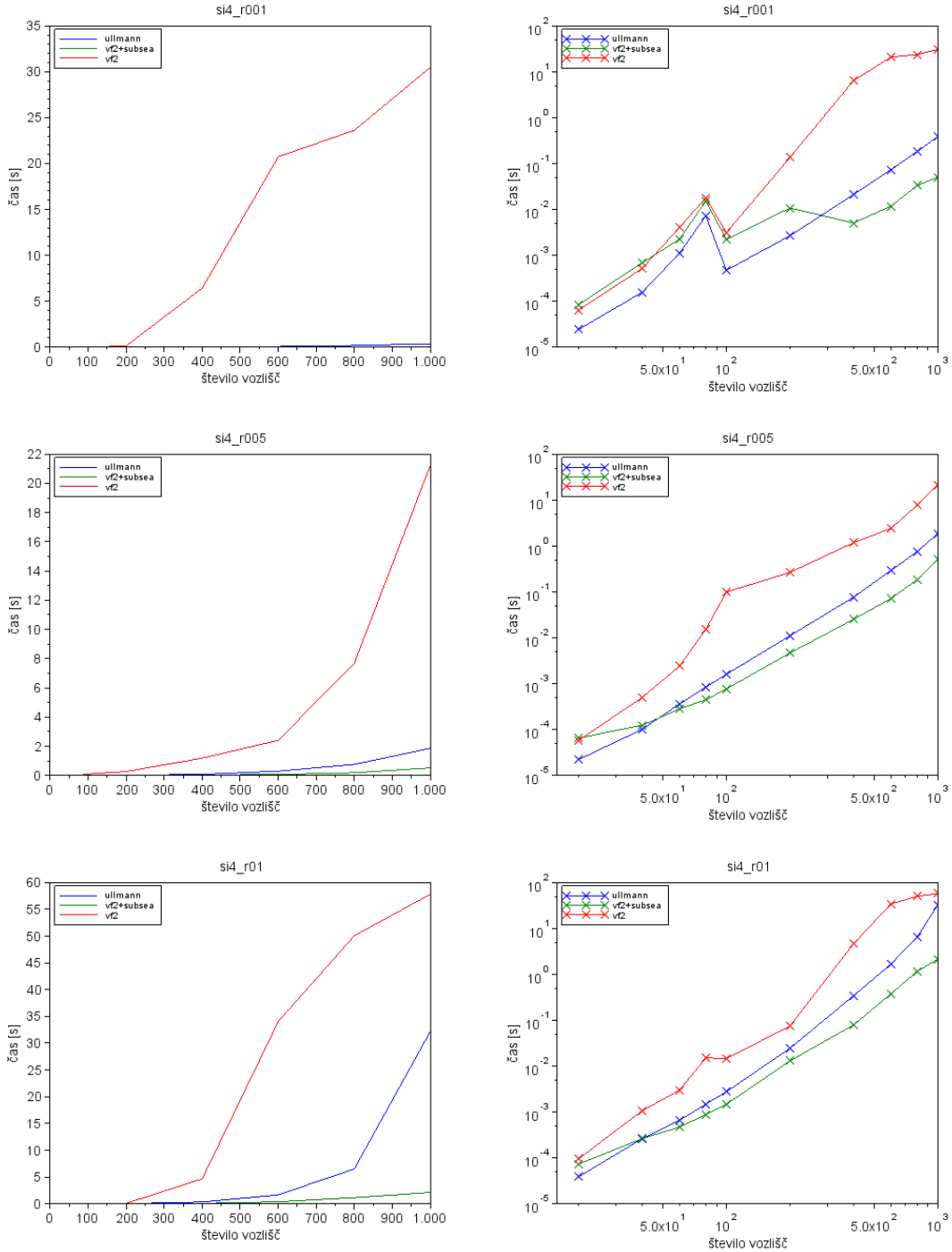
Slika 6.2: Število rešenih primerov za grafe tipov si4_r001, si4_r005, si4_r01.



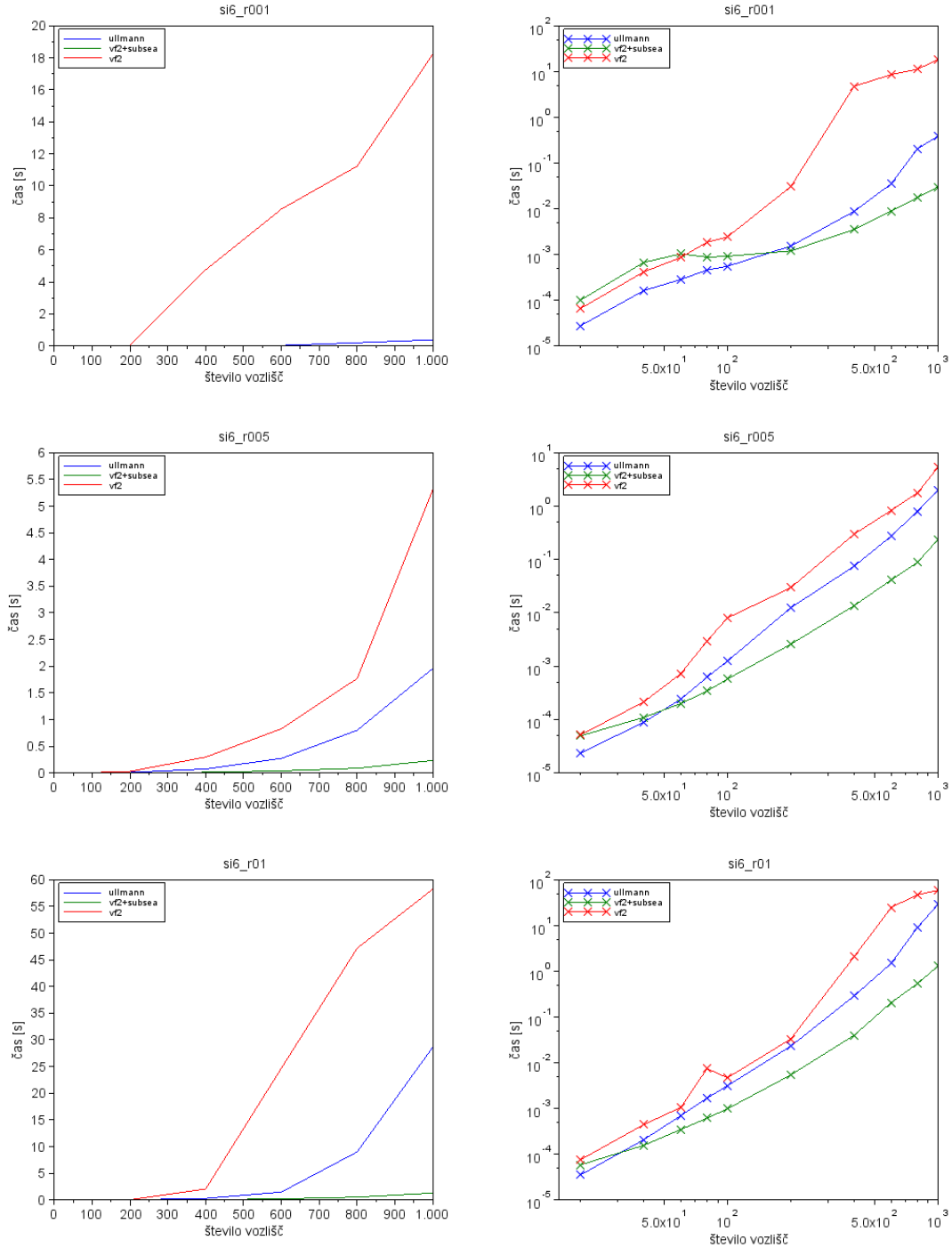
Slika 6.3: Število rešenih primerov za grafe tipov si6_r001, si6_r005 in si6_r01.



Slika 6.4: Povprečen čas reševanja za grafe tipov si2_r001, si2_r005 in si2_r01.



Slika 6.5: Povprečen čas reševanja za grafe tipov si4_r001, si4_r005, si4_r01.



Slika 6.6: Povprečen čas reševanja za grafe tipov si6_r001, si6_r005 in si6_r01.

Poglavje 7

Sklepne ugotovitve

V delu smo primerjali tri algoritme za iskanje izomorfnih podgrafov in jih preizkusili na bazi 9000 parov grafov. Na manjših testnih primerih je najhitrejši izboljšani Ullmannov algoritem, kar ima praktično vrednost npr. pri iskanju na kemijskih grafih – ti so običajno majhni, iskati pa je potrebno v velikih bazah grafov [5]. Na večjih grafih in na grafih z več povezavami je boljši algoritem, ki smo ga ustvarili sami kot kombinacijo algoritma VF2 in algoritma Subsea. Oba izboljšana algoritma sta vsaj za velikostni red hitrejša od obstoječega VF2, ki se trenutno v praksi najbolj uporablja.

Naslednji korak bi bila implementacija celotnega algoritma Subsea. Preveriti je potrebno, če se tudi celoten algoritem obnaša podobno kot hibrid med VF2 in Subsea. Lahko bi poskusili tudi obratno od naše izboljšave, torej da bi v algoritmu Subsea uporabili dele algoritma VF2. Konkretno bi lahko v fazi iskanja uporabili preverjanje kandidatov iz VF2 (opisano v razdelku 4.2). Subsea namreč preverja samo kandidatove že preiskane sosede, VF2 pa z relativno majhnim vplivom na čas računanja preveri tudi neobiskane sosede. Preverili bi lahko tudi možnost vzporednega računanja opisanih algoritmov.

Algoritme bi morali preizkusiti tudi na več testnih primerih. Najprej na preostalih grafih iz uporabljene baze grafov, potem pa še na realnih grafih iz prakse kot npr. proteinih in socialnih omrežjih.

Dodatek A

Implementacija izboljšave algoritma VF2

Algoritem A.1: Zgodovina pregleda grafa za algoritem VF2 (vflib)

```
#include <algorithm>
#include <vector>
#include <cstdio>
#include <climits>
#include <cstring>
#include <iostream>
#include "argraph.h"

// This class calculates order as described in Subsea algorithm -
// depth first search and choose best with breadth first search
class SearchTraverse {
private:
  int n;
  node_id* order;
  bool *processed;
  bool *estimateProcessed;
  int index;
  Graph *g;

  struct NodeEstimate {
    node_id id;
    int steps;
    int size;
  };

  // a < b
  bool nodeEstimateCompare(NodeEstimate a, NodeEstimate b) {
    if (a.steps == b.steps) {
      return a.size < b.size;
```

```
30      } else {
31        return a.steps < b.steps;
32      }
33    }
34
35    // Breadth first search
36    NodeEstimate getNodeEstimate(node_id start, node_id from) {
37      memset(estimateProcessed, 0, sizeof(bool) * n);
38      NodeEstimate e;
39      e.id = start;
40      e.steps = 1;
41      e.size = 0;
42      int total = 0;
43      std::vector<node_id> s;
44      std::vector<node_id> n;
45      s.push_back(start);
46      do {
47        int p = 0;
48        bool skippedEdge = false;
49        // Loop visited nodes
50        for (unsigned int idx = 0; idx < s.size(); idx++) {
51          // Loop node neighbors
52          for (int i = 0; i < g->InEdgeCount(s.at(idx)); i++) {
53            node_id n1 = g->GetInEdge(s.at(idx), i);
54            if (s.at(idx) == start && n1 == from && skippedEdge ==
                  false) { // Do not use edge start->from
55              skippedEdge = true; // Only skip one edge
56              continue;
57            }
58            if (!estimateProcessed[n1]) { // Add to visited
59              n.push_back(n1);
60              estimateProcessed[n1] = true;
61              if (processed[n1])
62                p++;  // Count, if node is in goal
63            }
64          }
65          for (int i = 0; i < g->OutEdgeCount(s.at(idx)); i++) {
66            node_id n1 = g->GetOutEdge(s.at(idx), i);
67            if (s.at(idx) == start && n1 == from && skippedEdge ==
                  false) {
68              skippedEdge = true;
69              continue;
70            }
71            if (!estimateProcessed[n1]) {
72              n.push_back(n1);
73              estimateProcessed[n1] = true;
74              if (processed[n1])
75                p++;
76            }
```

```
77          }
78        }
79        if (p > 0) {
80          e.size = -p;
81          return e;
82        } else {
83          e.steps++;
84          total += n.size();
85          s.clear();
86          s.swap(n);
87        }
88      } while (s.size() > 0);
89      e.steps = 99999999;
90      e.size = total;
91      return e;
92    }
93
94    // Depth first visit, in best-first order
95    void Visit(node_id node) {
96      order[index++] = node;
97      processed[node] = true;
98
99      // Find all non-processed neighbors of added node
100     std::vector<node_id> neighbors;
101     for (int i = 0; i < g->InEdgeCount(node); i++) {
102       node_id n1 = g->GetInEdge(node, i);
103       if (!processed[n1]) {
104         neighbors.push_back(n1);
105       }
106     }
107     for (int i = 0; i < g->OutEdgeCount(node); i++) {
108       node_id n1 = g->GetOutEdge(node, i);
109       if (!processed[n1]) {
110         neighbors.push_back(n1);
111       }
112     }
113
114     // Visit all neighbors in best-first order
115     while (neighbors.size() > 0) {
116       int bestNodeIdx = -1;
117       node_id bestNode = NULL_NODE;
118       NodeEstimate bestEstimate;
119
120       // Find best neighbor
121       for (unsigned int i = 0; i < neighbors.size(); i++) {
122         node_id n1 = neighbors.at(i);
123         if (!processed[n1]) {
124           NodeEstimate nEstiname = getNodeEstimate(n1, node);
```

```
125           if (bestNode == NULL_NODE || nodeEstimateCompare(nEstiname,
                  bestEstimate)) {
126             bestNodeIdx = i;
127             bestNode = n1;
128             bestEstimate = nEstiname;
129           }
130         }
131       }
132       if (bestNode != NULL_NODE) {
133         Visit(bestNode);
134         // Remove visited neighbor
135         neighbors[bestNodeIdx] = neighbors[neighbors.size()-1];
136         neighbors.pop_back();
137       } else {
138         break;  // No more unvisited neighbors
139       }
140     }
141   }
142
143 public:
144   node_id * SortNodesBySearchTraverse(Graph *g) {
145     n = g->NodeCount();
146     order = new node_id[n];
147     processed = new bool[n];
148     estimateProcessed = new bool[n];
149     index = 0;
150     this->g = g;
151     for (int i = 0; i < n; i++) {
152       order[i] = NULL_NODE;
153       processed[i] = false;
154     }
155     // Start with highest degree node (not part of Subsea)
156     while (index < n) {
157       int first = -1;
158       int firstEdgeCnt = -1;
159       for (int i = 0; i < n; i++) {
160         if (!processed[i] && g->EdgeCount(i) > firstEdgeCnt) {
161           first = i;
162           firstEdgeCnt = g->EdgeCount(i);
163         }
164       }
165       Visit(first);
166     }
167     delete[] processed;
168     delete[] estimateProcessed;
169     return order;
170   }
171 };
```

Algoritem A.2: Sprememba kode v knjižnici vflib za vključitev izboljšave algoritma VF2; `heuristicOrder` je vrnjena tabela iz algoritma A.1

```
1  bool VF2SubState::NextPair(node_id *pn1, node_id *pn2, node_id
       prev_n1, node_id prev_n2) {
2  prev_n1 = heuristicOrder[core_len];
3  if (prev_n2 == NULL_NODE)
4    prev_n2 = 0;
5  else
6    prev_n2++;
7
8  if (out_1[prev_n1] != 0 && in_1[prev_n1] != 0) {
9    while (prev_n2 < n2 && (core_2[prev_n2] != NULL_NODE || out_2[
         prev_n2] == 0 || in_2[prev_n2] == 0)) {
10     prev_n2++;
11   }
12 } else if (out_1[prev_n1] != 0) {
13   while (prev_n2 < n2 && (core_2[prev_n2] != NULL_NODE || out_2[
         prev_n2] == 0)) {
14     prev_n2++;
15   }
16 } else if (in_1[prev_n1] != 0) {
17   while (prev_n2 < n2 && (core_2[prev_n2] != NULL_NODE || in_2[
         prev_n2] == 0)) {
18     prev_n2++;
19   }
20 } else {
21   while (prev_n2 < n2 && core_2[prev_n2] != NULL_NODE) {
22     prev_n2++;
23   }
24 }
25
26 if (prev_n1 < n1 && prev_n2 < n2) {
27   *pn1 = prev_n1;
28   *pn2 = prev_n2;
29   return true;
30 }
31
32 return false;
33 }
```

Literatura

[1] D. Conte, P. Foggia, C. Sansone, M. Vento, "Thirty Years Of Graph Matching In Pattern Recognition", *International Journal of Pattern Recognition and Artificial Intelligence*, št. 18, zv. 3, 2004, str. 265–298.

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

[3] L. Cordella, P. Foggia, C. Sansone, M. Vento, "An improved algorithm for matching large graphs", *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, 2001, str. 149–159.

[4] L. Cordella, P. Foggia, C. Sansone, M. Vento, "Performance evaluation of the VF graph matching algorithm", *Proceedings of the 10th International Conference on Image Analysis and Processing*, 1999, str. 1372–1177.

[5] H. C. Ehrlich, M. Rarey, "Systematic benchmark of substructure search in molecular graphs – From Ullmann to VF2", *Journal of Cheminformatics*, št. 4, zv. 1, 2012.

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

[7] M. R. Garey, D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", Freeman and Company, 1979.

[8] J. Lee, W. Han, R. Kasperovics, J. Lee, "An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases" *Proceedings of the VLDB Endowment (PVLDB)*, št. 6, zv.2, 2012, str. 133–144.

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

[10] B. T. Messmer, H. Bunke, "Subgraph isomorphism detection in polynomial time on preprocessed model graphs", *Recent Developments in Computer Vision*, 1996, str. 373–382.

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

[12] C. Solnon, "AllDifferent-based filtering for subgraph isomorphism", *Artificial Intelligence*, št. 174, zv. 12–13, 2010, str. 850–864.

[13] J. R. Ullmann, "An Algorithm for Subgraph Isomorphism", *Journal of the ACM*, št. 23, zv. 1, 1976, str. 31–42.

[14] S. Zampelli, "A constraint programming approcah to subgraph isomorphism", Doktorska disertacija, Université catholique de Louvain, Département d'Ingénierie Informatique, Belgija, 2008.