Konačni automati kao model evaluacije za upite na grafovima

Dujić, Mateo

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:315108

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-09-11**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Mateo Dujić

KONAČNI AUTOMATI KAO MODEL EVALUACIJE ZA UPITE NA GRAFOVIMA

Diplomski rad

Voditelj rada:
prof.dr.sc. Domagoj Vrgoč

Zagreb, 2023

Ovaj diplomski rad obranjen je dana ____________________ pred ispitnim povjerenstvom u sastavu:

1. ____________________________, predsjednik
2. ____________________________, član
3. ____________________________, član

Povjerenstvo je rad ocijenilo ocjenom ________________.

Potpisi članova povjerenstva:

1. ____________________________
2. ____________________________
3. ____________________________

Roditeljima, sestri i Lauri.
Želim izraziti svoju iskrenu zahvalnost mentoru prof. dr. sc. Domagoju Vrgoču na neizmjernom strpljenju i neprocjenjivoj pomoći koju mi je pružio prilikom pisanja ovog rada.

Sadržaj

Sadržaj **iv**

Uvod **1**

1 Osnovne definicije **3**
1.1 Grafovske baze podataka . . . 3
1.2 Modeli grafovskih baza podataka . . . 5
1.3 Regularni upiti na putove u grafovskoj bazi podataka . . . 9

2 Algoritmi za izračunavanje `RPQ`ova **11**
2.1 Bazni slučaj: `restrictor` == `ŠETNJA` . . . 12
2.2 `restrictor` ∈ {`STAZA`, `ACIKLIČAN`, `JEDNOSTAVAN`} . . . 18

3 Implementacija **31**
3.1 Glushkovov algoritam . . . 32
3.2 `NKA` s jedinstvenim završnim stanjem . . . 33
3.3 Implementacija grafovske baze podataka . . . 34

4 Eksperimenti **39**
4.1 Umjetni primjer s eksponencijalno mnogo slučajeva . . . 40
4.2 Primjer iz stvarnog svijeta . . . 48

Bibliografija **53**

Uvod

U današnje vrijeme, velike količine podataka prikupljaju se i pohranjuju u različitim oblicima. Grafovske baze podataka istaknule su se kao jedan od načina za učinkovitu pohranu, upravljanje i analizu podataka sa složenom strukturom. U kontekstu grafovskih baza podataka, regularni upiti na putovima postali su jedni od ključnih upita za dobivanje relevantnih informacija iz grafova.

Cilj ovog diplomskog rada je implementacija ključnih algoritama regularnih upita na putove. Pritom je važno dokazati rezultate o složenosti za svakoga od njih.

Na kraju će se svi teoretski rezultati provjeriti u praksi nizom eksperimenata.

Rad će dati pregled primjene grafovskih baza podataka i algoritama za izračunavanje regularnih upita na putove. Biti će istaknuti različiti modeli baza, implementacija, kao i eksperimenti koji će pokazati uspješnost razvijenih algoritama. Diplomski rad pridonosi razumijevanju i razvoju grafovskih baza podataka u kontekstu regularnih upita na putove.

Poglavlje 1

Osnovne definicije

1.1 Grafovske baze podataka

Predmet proučavanja ovog rada neki su algoritmi koji se izvode nad podatcima spremljenima u strukturi koju zovemo grafovska baza podataka. Postoji puno različitih definicija za nju, no u ovom radu koristit će se sljedeća.

Definicija 1.1.1. *Neka je* `Vrhovi` *skup identifikatora vrhova i* `Bridovi` *skup identifikatora bridova, pri čemu je* $\texttt{Vrhovi} \cap \texttt{Bridovi} = \emptyset$*. Uz to, neka je* `Oznake` *skup oznaka.* ***Grafovska baza podataka*** *je uređena četvorka* (V, E, ρ, λ)*, pri čemu je:*

- $V \subset \texttt{Vrhovi}$ *konačan skup vrhova,*
- $E \subset \texttt{Bridovi}$ *konačan skup bridova,*
- $\rho\colon E \to V \times V$ *totalna funkcija. Ako je* $\rho(e) = (v_1, v_2)$*, to znači da je* e *usmjereni brid od* v_1 *do* v_2*,*
- $\lambda\colon E \to \texttt{Oznake}$ *totalna funkcija koja bridu pridružuje oznaku.*

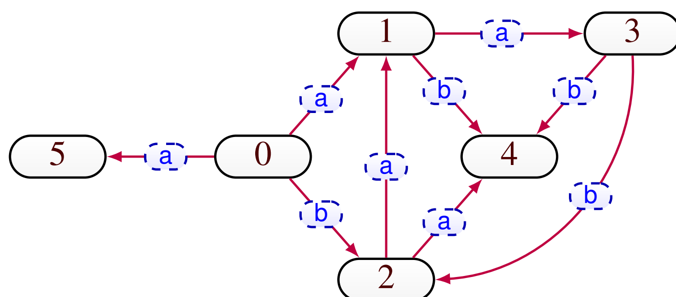
Na sljedećoj stranici nalazi se primjer baze koja će nam i poslije poslužiti.

Jezici za upite na grafovskim bazama podataka zasnivaju se (između ostalog) na traženju putova između dva čvora na grafu.

Definicija 1.1.2. ***Put*** *u grafovskoj bazi* $G = (V, E, \rho, \lambda)$ *je niz*

$$p = v_0 e_1 v_1 e_2 v_2 \dots e_n v_n$$

gdje je $n \geq 0$*,* $e_i \in E$*,* $\rho(e_i) = (v_{i-1}, v_i)$ *za svaki* $i = 1, \dots, n$*. Ako je* p *put u* G*, definiramo oznaku puta kao niz oznaka bridova* $\texttt{oznaka}(p) := \lambda(e_1) \dots \lambda(e_n)$*. Početak puta označavamo s* $\texttt{izvor}(p) := v_0$*, kraj puta s* $\texttt{cilj}(p) := v_n$ *i duljinu puta s* $l(p) := n$ *(broj bridova koje obilazi). Kažemo da je put* p*:*



Slika 1.1: Primjer jednostavne grafovske baze na kojoj ćemo izvršavati algoritme.

- *`ŠETNJA` za bilo koji p,*
- *`STAZA` ako p ne ponavlja bridove,*
- *`ACIKLIČAN` ako p ne ponavlja nijedan vrh,*
- *`JEDNOSTAVAN` ako p ne ponavlja vrhove, osim eventualno $\texttt{izvor}(p) = \texttt{cilj}(p)$.*

Dodatno, za dani skup puteva P nad grafovskom bazom G, kažem da je $p \in P$ `NAJKRAĆI` put u P, ako vrijedi $l(p) \leq l(p')$, za svaki $p' \in P$. Koristimo oznaku $\texttt{Putovi}(G)$ da bismo označili sve putove u G.

1.2 Modeli grafovskih baza podataka

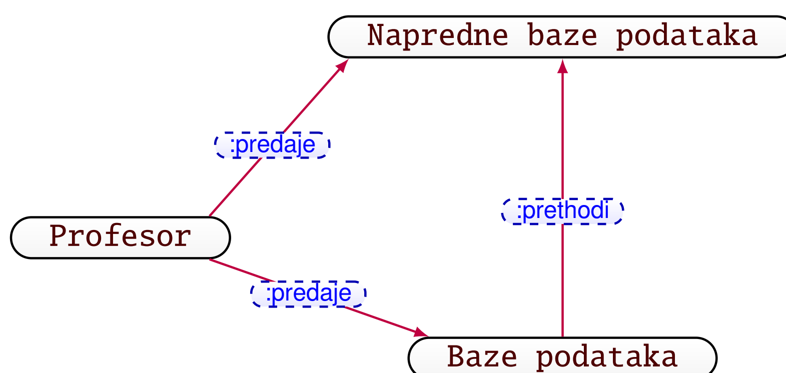
U ovom odjeljku prikazat ćemo dvije definicije grafova koje se često koriste (primjerice u [1]). Za obje definicije najprije dajemo motivaciju.

Promotrimo što najčešće čvorovi i bridovi predstavljaju u raznim slučajevima uporabe. Zamislimo da imamo neku domenu koja nam je zanimljiva. Objekte u njoj predstavljat ćemo čvorovima. Odnose između objekata prikazujemo bridovima. Međutim, što ako želimo da objekti mogu biti u različitim odnosima? Za to nam treba dodatna informacija te tako dolazimo do potrebe za definiranjem *grafova s označenim bridovima*.

Definicija 1.2.1. ***Graf s označenim bridovima*** *je par* (V, E) *gdje je*

1. $V \subset \texttt{Vrhovi}$ *konačan skup vrhova,*
2. $E \subseteq V \times \texttt{Oznake} \times V$.

Grafovi s označenim bridovima (kakav je primjerice na Slici 1.2) masovno se koriste u praksi, najpoznatiji primjer su podatci koji su povezani u globalnu mrežu - Resource Description Framework (skraćeno: `RDF`).



Slika 1.2: Primjer grafa s označenim bridovima.

Nadalje, što kada želimo prikazati dva objekta v_1, v_2 koji su povezani bridovima gdje svi imaju jednu te istu oznake e, ali svaka se veza (brid) odnosi na različite značajke od v_1? Dakle, želimo više bridova među vrhovima, ali na raspolaganju nam je samo oznaka brida. U slučaju prethodne definicije to nije moguće: E je skup pa on ne može sadržavati više od jednog objekta (v_1, e, v_2). U ovom slučaju, umjesto binarnih relacija, veze možemo prikazivati n-arnim relacijama, za $n > 2$. To znači da ćemo dozvoliti da i bridovi i vrhovi za svaku značajku/svojstvo (od njih konačno mnogo) mogu spremati podatak/vrijednost. Definiramo još izražajniju vrstu grafova: *grafove svojstava*.

Definicija 1.2.2. ***Graf svojstava*** *je uređena petorka* $(V, E, \rho, \lambda, \sigma)$*, pri čemu je:*

1. $V \subset \texttt{Vrhovi}$ *konačni skup vrhova,*

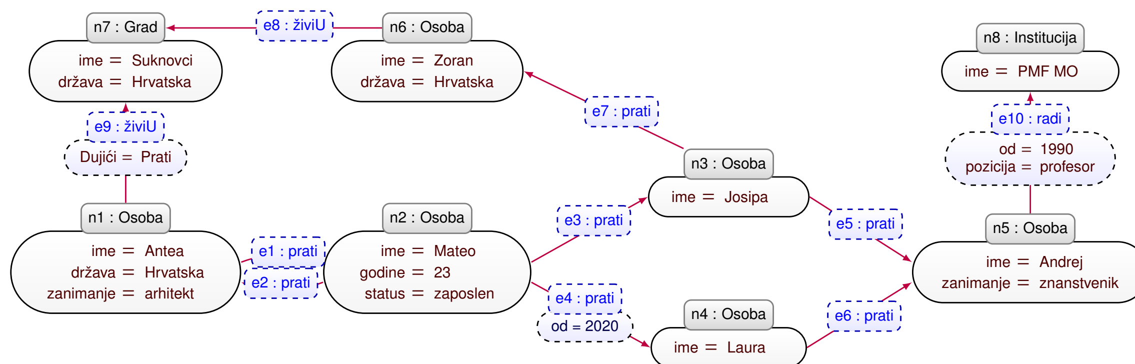
2. $E \subset \texttt{Bridovi}$ *konačni skup bridova t.d.* $V \cap E = \emptyset$*,*

3. $\rho \colon E \to (V \times V)$ *totalna funkcija,*

4. $\lambda \colon (V \cup E) \to \texttt{Oznake}$*,*

5. $\sigma \colon (V \cup E) \times \texttt{Svojstva} \to \texttt{Vrijednosti}$ *parcijalna funkcija pri čemu je* $\texttt{Svojstva}$ *konačan skup svojstava, a* $\texttt{Vrijednosti}$ *skup vrijednosti.* σ *zovemo funkcijom svojstava.*

Primjer baze u skladu s gornjom definicijom dan je na Slici 1.2: Ako bismo željeli



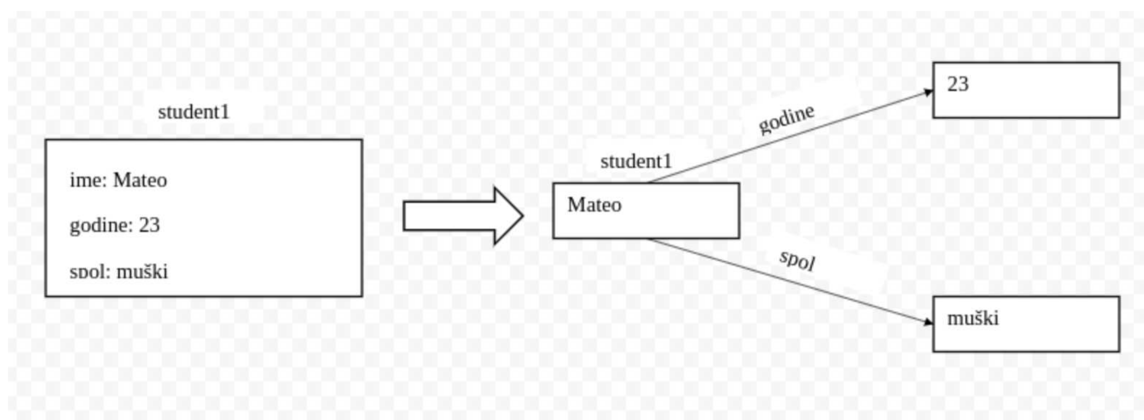
Slika 1.3: Primjer grafovske baze koja predstavlja društvenu mrežu

definirati slučaj u kojem omogućujemo više vrijednosti na bridovima ili vrhovima, kodomenu funkcije σ u točki 5. u Definiciji 1.2.2 (skup $\texttt{Vrijednosti}$) zamijenili bismo s $\mathcal{P}(\texttt{Vrijednosti})$. Trenutno najpoznatija grafovska baza podataka, Neo4J, zasniva se na ovom modelu grafa. Ona omogućuje spremanje najviše jedne vrijednosti u brid, više vrijednosti u čvor i najviše jednu vrijednost u svako svojstvo.

Uočimo da je početna definicija koju koristimo upravo druga navedena definicija bez da uzimamo u obzir podatke (prve četiri značajke, bez korištenja σ).

Simuliranje između različitih modela baza

U nastavku pokazujemo kako različiti modeli baza mogu jedni druge simulirati kao što je objašnjeno u [9]. Najprije pokazujemo kako model baze koja ima vrhove s više podataka svesti na model bazu koja ima samo jedan podatak u svakom bridu.



Slika 1.4: Svođenje jednog vrha s više vrijednosti na graf gdje svaki vrh ima jednu vrijednost.

Primjer 1.2.3. *Pretpostavimo da postoji baza s više podataka u nekom čvoru. Odaberimo jedan takav čvor oznake v i podacima prikazanima kao rječnik* $\{(x_i, y_i), 1 \leq i \leq n\}$*, gdje je* $n > 1$ *broj podataka. Tada transformaciju čvora vršimo na sljedeći način:*

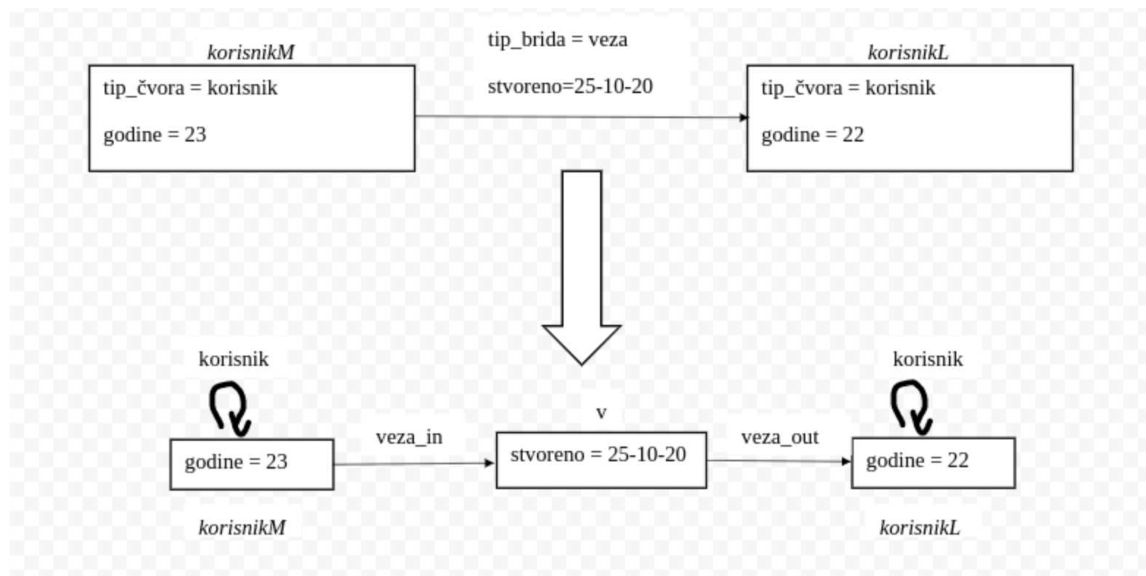
1. *za svaki* (x_i, y_i), $2 \leq i \leq n$*, dodamo u bazu čvor* v_i *s podatkom* y_i *te označeni brid* (v, x_i, v_i)*,*

2. *čvoru v ostavimo podatak* y_1*, a ostale pobrišemo.*

Dakle, baze s više podataka u čvorovima mogu se svesti na one s jednim podatkom. Pokažimo u idućem primjeru kako bazu s podacima u bridovima možemo svesti na bazu s bridovima bez podataka.

Primjer 1.2.4. *Pretpostavimo da postoji grafovska baza s barem jednim podatkom u bridovima. Tada čvorovi imaju dodatni podatak* `tip_čvora`*, a bridovi* `tip_brida`*. Neka su* v_1 *i* v_2 *čvorovi tipa t te* (v_1, e, v_2) *označeni brid s barem jednim podatkom p tipa s. Tada transformaciju u grafovsku bazu bez bridova koji imaju podatke vršimo na sljedeći način:*

1. *Izbacimo brid* (v_1, e, v_2)*,*



Slika 1.5: Pretvaranje baze s podacima u bridovima u bazu s bridovima bez podataka.

2. *Dodamo čvor s podatkom p (i, ako ih ima, svim ostalim podacima koji su se nalazili na bridu) koji označimo s v,*
3. *Dodamo bridove* (v_1, s_{ulaz}, v) *i* (v, s_{izlaz}, v_2)*,*
4. *Dodamo bridove petlje* (v_1, t, v_1) *i* (v_2, t, v_2)*.*

Ako je brid imao više podataka, onda novi čvor ima više podataka pa na njega primjenimo postupak iz prethodnog primjera.

Sada zaključujemo da se grafovska baza s više podataka u bridovima i čvorovima može svesti na jednostavniju: s čvorovima koji sadrže samo jedan podatak i bridovima koji sadrže samo oznaku.

1.3 Regularni upiti na putove u grafovskoj bazi podataka

Najpoznatiji deklarativni jezici za upite na grafovskim bazama podataka za `RDF` grafove su `SPARQL`, a za grafove svojstava `Cypher` te `Gremlin`. Svi jezici za upite u svojoj srži implementiraju sljedeće dvije operacije: *podudaranje uzoraka* (eng. match) i *navigacijski upit*. U ovome radu bavimo se navigacijskim upitima.

Putovi su osnovni objekti za navigaciju u grafovskoj bazi podataka. Najosnovniji tip navigacijskog upita je provjera postojanja puta između čvorova u grafu, bez obzira na oznake bridova. U praksi se često koriste upiti na putove s dodatnim ograničenjima, primjerice upravo s ograničenjima na oznake bridova. U gornjem primjeru za društvenu mrežu, sada možemo tražiti putove koji obilaze samo bridove s oznakama `prati` i tako traži pratitelje-od-pratitelja u dubinu.

Klasa navigacijskih upita koje ćemo proučavati zovu se **regularni upiti na putove** (eng. **regular path queries**, skraćeno `RPQ`).

Regularni upit na putove je izraz oblika

$$\texttt{selector? restrictor}\ (v, \texttt{regex}, ?x)$$

pri čemu je $v \in \texttt{Vrhovi}$, `regex` neki regularni izraz i $?x$ varijabla. Varijable `selector` i `restrictor` služe da bismo precizirali dodatna svojstva oko puteva koje vraćamo i zadaju se gramatikom:

$$\texttt{restrictor} : \texttt{ŠETNJA} \mid \texttt{STAZA} \mid \texttt{ACIKLIČAN} \mid \texttt{JEDNOSTAVAN}$$
$$\texttt{selector} : \texttt{NEKI} \mid \texttt{NEKI NAJKRAĆI} \mid \texttt{SVI NAJKRAĆI}$$

Dio $(v, \texttt{regex}, ?x)$ govori da želimo pronaći sve čvorove v' u grafu za koje postoji put od v do v', takav da je $\texttt{oznaka}(p)$ riječ u jeziku regularnog izraza `regex`. Oznaka `selector?` označava da se u (većini) upita `selector` može, ali i ne mora koristiti.

S obzirom da skup svih takvih putova može biti beskonačan, uloga `restrictor`a je da specificira koji putevi se smatraju valjanima, dok je uloga `selector`a filtriranje rezultata iz danog skupa valjanih putova. Nadalje, formalno definiramo semantiku od `RPQ`, najprije bez `selector`a, a zatim proširujemo.

Definicija 1.3.1. *Neka je G grafovska baza podataka i q neki* `RPQ` *oblika*

$$\texttt{restrictor}\ (v, \texttt{regex}, ?x)$$

Koristimo oznaku $\texttt{Putovi}(G, \texttt{restrictor})$ *da označimo sve putove valjane s obzirom na* `restrictor`*.* $\mathcal{L}(\texttt{regex})$ *je jezik svih riječi koje* `regex` *generira. Zatim definiramo semantiku od q nad G, označenu s* $[\![q]\!]_G$*:*

$$\begin{aligned}[\![\texttt{restrictor}\ (v, \texttt{regex}, ?x)]\!]_G = \{(p, v') \mid\ & p \in \texttt{Putovi}(G, \texttt{restrictor}, izvor) \\ & \texttt{izvor}(p) = v,\ \texttt{cilj}(p) = v', \\ & \texttt{oznaka}(p) \in \mathcal{L}(\texttt{regex})\}\end{aligned}$$

Primjer 1.3.2. *Primjerice,* $\texttt{Putovi}(G, \texttt{JEDNOSTAVAN})$ *je skup svih jednostavnih puteva, odnosno onih koji ne ponavljaju vrhove, osim eventualno izvor i cilj. Osim toga,* RPQ *"*$\texttt{JEDNOSTAVAN}$ $(v, \texttt{regex}, ?x)$*" vraća sve parove* (p, v') *takve da je* p *jednostavan put u grafu od* v *do* v' *i* $\texttt{oznaka}(p) \in \mathcal{L}(\texttt{regex})$*.*

Konačno, proširimo semantiku upita da koriste i `selector`e kako smo ih na početku uveli. Definiciju proširujemo po slučajevima.

Definicija 1.3.3. *Neka je* $q = \texttt{restrictor}(v, \texttt{regex}, ?x)$ RPQ *kao iz prethodne definicije. Definiramo po slučajevima:*

- $[\![\texttt{NEKI restrictor } (v, \texttt{regex}, ?x)]\!]_G$ *za svaki* v' *takav da postoji* $(p, v') \in [\![q]\!]_G$ *vraća* ***jedan par*** $(p, v') \in [\![q]\!]_G$*, biran nedeterministički*
- $[\![\texttt{NEKI NAJKRAĆI restrictor } (v, \texttt{regex}, ?x)]\!]_G$ *za svaki* v' *takav da postoji* $(p, v') \in [\![q]\!]_G$ *vraća* ***jedan par*** $(p, v') \in [\![q]\!]_G$*, gdje je* p ***neki put najmanje duljine*** *od svih putova* p' *za koje je* $(p', v') \in [\![q]\!]_G$
- $[\![\texttt{SVI NAJKRAĆI restrictor } (v, \texttt{regex}, ?x)]\!]_G$ *za svaki* v' *takav da postoji* $(p, v') \in [\![q]\!]_G$ *vraća* ***skup svih parova*** $(p, v') \in [\![q]\!]_G$*, gdje su* p ***svi putovi najmanje duljine*** *od svih putova* p' *za koje* $(p'v') \in [\![q]\!]_G$

Nad putovima možemo promatrati relaciju ekvivalenciju, pri čemu su dva puta u relaciji ako imaju isti `cilj` v'. Uočimo da je u tom slučaju semantika `NEKI` i `NEKI NAJKRAĆI` nedeterministička za pojedinu klasu ekvivalencije kada ona ima više od jednog elementa. Za $[\![q]\!]_G$ koristimo oznaku $|[\![q]\!]_G|$ da bismo označili **duljinu** svih putova u $[\![q]\!]_G$. To je upravo suma duljina svih putova u $[\![q]\!]_G$. S obzirom da stroj koji računa $[\![q]\!]_G$ sigurno mora zapisati barem jedan cijeli put, $|[\![q]\!]_G|$ bit će nam važna kad budemo izražavali složenost ovog algoritma.

Poglavlje 2

Algoritmi za izračunavanje RPQova

U nastavku se prisjetimo teorije konačnih automata. Najprije za regularni izraz `regex` s $\mathcal{L}(\texttt{regex})$ označimo jezik koji taj regularni izraz generira. Ponovimo definicije konačnih automata kao iz [12].

Definicija 2.0.1. ***Deterministički konačni automat*** *(skraćeno KA) nad abecedom Σ je idealizirani (matematički) stroj $\mathcal{M} := (Q, \Sigma, \delta, q_0, F)$ koji pored Σ ima:*

- ***konačan*** *skup Q čije elemente nazivamo stanjima,*
- *istaknuti element $q_0 \in Q$ koji nazivamo početnim stanjem,*
- *podskup $F \subseteq Q$ čije elemente nazivamo završnim stanjima,*
- *funkciju prijelaza $\delta\colon Q \times \Sigma \to Q$ često zadanu tablično.*

Kažemo da $\mathcal{M}$ prihvaća riječ $w = \alpha_1 \ldots \alpha_n \in \Sigma^*$ ako postoji niz stanja $r_0, r_1, \ldots, r_n \in Q$ takav da je $r_0 = q_0$, $\delta(r_{i-1}, \alpha_i) = r_i$ za sve $i \in \{1, \ldots, n\}$ (takav je jedinstven za zadane $\mathcal{M}$ i w), te $r_n \in F$.
Nedeterministički konačni automat *(skraćeno NKA) ima sve isto kao deterministički osim:*

- *abecedu $\Sigma_\epsilon := \Sigma \cup \{\epsilon\}$,*
- *umjesto funkcije prijelaza δ ima* ***relaciju prijelaza*** *$\delta \subseteq Q \times \Sigma_\epsilon \times Q$.*

Svaki `regex` može se pretvoriti u ekvivalentan NKA prostorne složenosti $O(|\texttt{regex}|)$. Postoje razne konstrukcije pretvorbe iz regularnog izraza u automat. Neke od njih su: Thompsonova, Glushkovova itd. Nadalje, za potrebe ovog rada pretpostavimo da nema ϵ-prijelaza. Iz tog razloga koristimo Glushkovov algoritam jer on, za razliku od Thompsonovog, vraća automate bez ϵ-prijelaza. NKA zovemo *nedvosmislenim* ako ima najviše jedan prihvaćajući prolaz za svaku riječ. Svaki KA je nedvosmislen, ali obrat ne vrijedi.

2.1 Bazni slučaj: `restrictor` == `ŠETNJA`

U baznome slučaju algoritma koje ćemo pokazati, izbor `restrictora` je `ŠETNJA`, odnosno svi RPQ-ovi tog oblika su:

$$\texttt{selector ŠETNJA } (v, \texttt{regex}, ?x).$$

Ovo je (jedini) slučaj u kojem ne piše `selector?`, već `selector`, što znači da je `selector` obavezan. To vrijedi jer je broj šetnji po grafu koji ima petlju neograničen pa ih moramo nekako ograničiti. U ostalim mogućnostima upita (s drugim `restrictorom`) nemamo ovaj problem pa za njih `selector` može i ne mora biti prisutan.

Izračunavanje RPQ-ova kombinira pretraživanje po grafu paralelno s pokretanjem automata. Ideja kombiniranja grafa i automata u literaturi se može naći pod imenom **produktna konstrukcija**. Sada slijedi jedan od ključnih pojmova koje ćemo koristiti, **produktni graf**.

Definicija 2.1.1. *Neka je dana grafovska baza podataka* $G = (V, E, \rho, \lambda)$ *i izraz oblika* $q = (v, \texttt{regex}, ?x)$. *Neka je* $(Q, \Sigma, \delta, q_0, F)$ `regex`*-u ekvivalentni* NKA.
Produktni graf $G_\times$ *je grafovska baza podataka* $G_\times = (V_\times, E_\times, \rho_\times, \lambda_\times)$, *pri čemu je:*

- $V_\times = V \times Q$
- $E_\times = \{(e, (q_1, a, q_2)) \in E \times \delta \mid \lambda(e) = a\}$
- $\rho_\times(e, d) = ((x, q_1), (y, q_2))$ *ako* : $d = (q_1, a, q_2), \lambda(e) = a, \rho(e) = (x, y)$
- $\lambda_\times((e, d)) = \lambda(e)$.

Intuitivno, produktni graf dobiven je kartezijevim produktom početnog grafa i automata za `regex`. Za neki $q = (v, \texttt{regex}, ?x)$, sada možemo pronaći sve v' dostižive iz v u originalnoj grafovskoj bazi G, ali po putu `oznaka` sigurno pripada u $\mathcal{L}(\texttt{regex})$ tako da u $G_\times$ tražimo sve vrhove $(v', q') \in V_\times$, pri čemu je $q' \in F$, dostižive bilo kojim putom iz (v, q_0). Uočimo ključno: u $G_\times$ ne moramo brinuti pripada li oznaka puta u $\mathcal{L}(\texttt{regex})$ jer je to osigurano produktnom konstrukcijom.

Dakle, svi takvi v' mogu se pronaći korištenjem jednostavnih algoritama za pretraživanje (BFS, DFS) po grafu $G_\times$ počevši s (v, q_0). Usputna korist ovog pristupa je da ne moramo u startu konstruirati cijeli $G_\times$, nego radije tijekom izvođenja algoritma idemo po susjedima i gradimo graf korak-po-korak.

`selector` == `NEKI(NAJKRAĆI)`?

Najprije ćemo pokazati algoritam kada je `selector` `NEKI` ili `NEKI NAJKRAĆI`, tj. za sve RPQ-ove oblika

$$q = \texttt{NEKI (NAJKRAĆI?) ŠETNJA } (v, \texttt{regex}, ?x).$$

Glavna je ideja već objašnjena: za grafovsku bazu podataka G i upit q kao gore, možemo provesti BFS ili DFS počevši od čvora (v, q_0) u produktnom grafu $G_\times$, izgrađenog od automata $\mathcal{A}$ za `regex` i G. U nastavku prikazujemo algoritam. Svi algoritmi preuzeti su iz [3] i [10].

Algoritam 1 Izračunavanje `RPQ` upita `NEKI (NAJKRAĆI)? ŠETNJA` $(v,$ `regex`$, ?x)$ na grafovskoj bazi $G = (V, E, \rho, \lambda)$

```
1: function NekaŠetnja(G, q)
2:     𝒜 ← konstruirajAutomat(regex)          ▷ q0 početno stanje, F konačna stanja
3:     inicijaliziraj red/stog Otvoreni
4:     inicijaliziraj rječnik Posjećeni
5:     inicijaliziraj skup DostignutoZavršno
6:     početnoStanje ← (v, q0, null, ⊥)
7:     Posjećeni.push(početnoStanje)
8:     Otvoreni.push(početnoStanje)
9:     if v ∈ V and q0 ∈ F then
10:        DostignutoZavršno.add(v)
11:        Rješenja.add(v)
12:    end if
13:    while Otvoreni ≠ ∅ do
14:        trenutni ← Otvoreni.pop()          ▷ trenutni = (n, q, brid, prethodni)
15:        for idući = (n', q', edge') ∈ Susjedi(trenutni, G, 𝒜) do
16:            if (n', q', ⋆, ⋆) ∉ Posjećeni then
17:                novoStanje ← (n', q', brid', trenutni)
18:                Posjećeni.push(novoStanje)
19:                Otvoreni.push(novoStanje)
20:                if q' ∈ F and n' ∉ DostignutoZavršno then
21:                    DostignutoZavršno.add(n')
22:                    put ← pronađiPut(trenutni).extend(n', edge')
23:                    Rješenja.add(put)
24:                end if
25:            end if
26:        end for
27:    end while
28:    return Rješenja
29: end function
30: function pronađiPut(stanje = (n, q, brid, prethodni))
31:    if prethodni == ⊥ then                                ▷ Početno stanje
32:        return [v]
```

```
33:     else                                          ▷ Rekurzivni "backtracking"
34:         return PRONAĐIPUT(prethodni).extend(n, brid)
35:     end if
36: end function
```

Možemo uočiti da je osnovni objekt koji promatramo te koji nije odmah jasan čemu služi tzv. **stanje traženja**. To je četvorka oblika $(n, q, brid, \texttt{prethodni})$ pri čemu je:

- n vrh iz G koji promatramo,
- q stanje iz $\mathcal{A}$ u kojem smo trenutno,
- *brid* je brid s kojim smo došli u n,
- `prethodni` je pokazivač na prethodno stanje traženja.

Intuitivno, (n, q) je stanje u $G_\times$, dok *brid* i `prethodni` služe za rekonstruiranje puta. Objasnimo i varijable koje koristimo u algoritmu:

- `Otvoreni` je red (u slučaju `BFS`-a) ili stog (u slučaju `DFS`-a) stanja traženja sa standardnim *push*() i *pop*() metodama.
- `Posjećeni` je rječnik stanja traženja koje smo već posjetili u obilasku, a on služi da ne završimo u beskonačnoj petlji. (n, q) je ključ da provjerimo je li neko stanje oblika $(n, q, brid, \texttt{prethodni})$ posjećeno. `prethodni` uvijek pokazuje na stanje koje je sigurno spremljeno u `Posjećeni`.
- `DostignutoZavršno` je skup vrhova (iz G) koje smo već vratili kao odgovor na upit pa nas više ne interesiraju (`NKA` ima više završnih stanja pa se može dogoditi da na više načina otkrijemo isti kraj puta).

Nakon što inicijaliziramo potrebne varijable, započinjemo pretragu s `početnoStanje` := $(v, q_0, null, \perp)$ koje zatim spremamo u `Otvoreni` i `Posjećeni` u linijama 6-8. U linijama 9-11, ako je v vrh u grafu i q_0 završno stanje, spremamo v u `DostignutoZavršno` i `Rješenja`. To rješenje će biti put sastavljen od samo jednog vrha. Dalje slijedi petlja klasičnog algoritma pretraživanja (`BFS/DFS`) u linijama 13-27. Petlja se izvršava dok je `Otvoreni` neprazan. U pomoćnu varijablu `trenutni` sprema se stanje traženja s vrha stoga/početka reda. Stanje traženja je oblika $(n, q, brid, \texttt{prethodni})$. U ovom trenutku odvija se pretraživanje dijela produktnog grafa koje smo spominjali: za trenutno stanje traženja, treba gurnuti na stog/red sva nova stanja traženja. Dakle, za vrh $(n, q) \in G_\times$, pronalazimo sve njegove susjede (n', q'), a u stanja traženja spremamo bridove $brid'$ do njih te pokazivače na prethodna stanja traženja `trenutni` koja će poslužiti kada budemo rekonstruirali put. Pritom je važno da ne posjećujemo susjede (n', q') koji su već u `Posjećeni` jer bismo

inače potencijalno završili u ciklusu nekog puta. novoStanje koje tada dobivamo, spremamo u Posjećeni i Otvoreni. Možda je q tada i završno stanje: ako jest, a n' nije već prije dostignut, onda smo pronašli rješenje. n' spremamo u DostignutoZavršno, rekonstruiramo šetnju i spremamo ju u Rješenja. Po konstrukciji produktnog grafa, znamo da je oznaka te šetnje sigurno u $\mathcal{L}$(regex). Za kraj napomenimo jako važnu činjenicu zbog koje smo napisali da je ovo algoritam i za NEKI i NEKI NAJKRAĆI selector. Zašto smo cijelo vrijeme spominjali i BFS i DFS? Upravo zato što BFS istovremeno traži **najkraće** šetnje do svakog dostiživog čvora, a DFS bilo kakve.

Složenost algoritma sastoji se od dva aspekta:

1. obilazak svih vrhova po produktnom grafu: svaki vrh $(n, q) \in V \times Q$ može biti posjećen najviše jednom, a kardinalnost skupa stanja je $|V_\times| = |V \times Q| = |V| \cdot |Q| \leq |G| \cdot |\mathcal{A}| \in O(|\mathcal{A}| \cdot |G|)$,

2. ispisivanje svih puteva: dok dospijemo u završno stanje, rekonstruiranje puta p traje $|p|$ koraka - skup svih takvih putova je $[\![q]\!]_G$ pa se sjetimo definicije $|[\![q]\!]_G|$ i zaključujemo da je to jednako vremenu ispisivanja svih rezultata.

Dakle, ukupna složenost algoritma je $O(|\mathcal{A}| \cdot |G| + |[\![q]\!]_G|)$ pa vrijedi sljedeći rezultat:

Teorem 2.1.2. *Neka je G grafovska baza podataka, i q upit oblika*

$$\texttt{NEKI (NAJKRAĆI)? ŠETNJA}(v, \texttt{regex}, ?x)$$

Ako je $\mathcal{A}$ automat za regex*, onda Algoritam 1 ispravno računa $[\![q]\!]_G$ u vremenu*

$$O(|\mathcal{A}| \cdot |G| + |[\![q]\!]_G|).$$

selector == SVI NAJKRAĆI

Za restrictor == ŠETNJA, jedini preostali slučaj je selector == SVI NAJKRAĆI, odnosno svi upiti oblika

$$\texttt{SVI NAJKRAĆI ŠETNJA}\ (v, \texttt{regex}, ?x)$$

nad grafovskom bazom G. Da bismo riješili ovaj problem, potrebno je proširiti BFS verziju Algoritma 1 koja će podržati pronalaženje **svih** putova između para (v, v') čvorova umjesto jednog jedinog. Intuicija je jednostavna: za određivanje najkraćih putova nakon dostizanja v' iz v uz poklapanje na regexu po prvi put, BFS to radi koristeći najkraći put. Duljina tog puta može se upamtiti zajedno s v' i onda svaki sljedeći put kad posjetimo vrh v', rješenje će također biti šetnja čija je duljina jednaka spremljenoj. Algoritam prikazujemo u nastavku.

Algoritam 2 Izračunavanje RPQ upita SVI NAJKRAĆI ŠETNJA (v, regex, ?x) na grafovskoj bazi $G = (V, E, \rho, \lambda)$

```
1: function SveNajkraćeŠetnje(G, q)
2:     𝒜 ← konstruirajAutomat(regex)          ▹ q0 početno stanje, qF završno stanje
3:     inicijaliziraj red Otvoreni
4:     inicijaliziraj rječnik Posjećeni
5:     if v ∈ V then
6:         početnoStanje ← (v, q0, 0, ⊥)
7:         Posjećeni.push(početnoStanje)
8:         Otvoreni.push(početnoStanje)
9:     end if
10:    while Otvoreni ≠ ∅ do
11:        trenutni ← Otvoreni.pop()      ▹ trenutni = (n, q, dubina, listaPreth)
12:        if q == qF then
13:            trenutniPutevi ← pronađiSvePuteve(trenutni)
14:            Rješenja.add(trenutniPutevi)
15:        end if
16:        for idući = (n′, q′, edge′) ∈ Susjedi(trenutni, G, 𝒜) do
17:            if (n′, q′, ⋆, ⋆) ∈ Posjećeni then
18:                (n′, q′, dubina′, listaPreth’) ← Posjećeni.get(n′, q′)
19:                if dubina + 1 == dubina′ then
20:                    listaPreth’.add(⟨trenutni,brid’⟩)
21:                end if
22:            else
23:                inicijaliziraj listu listaPreth
24:                listaPreth.add(⟨trenutni,brid’⟩)
25:                novoStanje ← (n′, q′, dubina + 1, listaPreth)
26:                Posjećeni.push(novoStanje)
27:                Otvoreni.push(novoStanje)
28:            end if
29:        end for
30:    end while
31: end function
32:
33: function pronađiSvePuteve(stanje = (n, q, dubina, listaPreth))
34:    if listaPreth == ⊥ then                                  ▹ Početno stanje
35:        return [v]
36:    end if
```

```
37:     for prethodni = (prethodnoStanje, prethodniBrid) ∈ listaPreth do
38:         for prethodniPut ∈ PRONAĐISVEPUTEVE(prethodnoStanje) do
39:             putovi.add(prethodniPutevi.extend(n, prethodniBrid))
40:         end for
41:     end for
42: end function
```

Neke se stvari ponavljaju kao i u Algoritmu 1, stoga ćemo se koncentrirati na one ideje koje su nove. Ovog puta želimo sve najkraće puteve pa zato koristimo BFS, odnosno varijabla Otvoreni je red. Od automata $\overline{\mathcal{A}}$ koji se generira iz regexa dodatno zahtijevamo da je nedvosmislen i da ima jedinstveno završno stanje q_F. Glavna razlika ovog i prethodnog algoritma je kako modeliramo stanje traženja. Ovdje je stanje traženje oblika $(n, q, dubina,$ listaPreth$)$ gdje je:

- n vrh grafa G,
- q stanje od $\mathcal{A}$,
- *dubina* duljina (bilo kojeg) najkraćeg puta od n do v,
- listaPreth je lista pokazivača gdje svaki pokazuje na neko prethodno stanje traženja iz kojeg se najkraćim raspoloživim putem može u n.

Lista listaPreth je vezana lista koja počinje praznim vrhom i svaki element joj je par ⟨stanjeTraženja, brid⟩. Intuitivno, ona nam omogućuje konstruirati **sve** najkraće puteve koji dostižu vrh jer ih zasigurno može biti i više od jednog. stanjeTraženja je pokazivač na prethodno stanje traženja, a brid se koristi da bi se dospjelo do vrha iz prethodnog stanja traženja.

Navedimo još neke ključne razlike u odnosu na prethodni algoritam. Uočimo da smo u Algoritmu1 u liniji 16 odbacivali stanje u $G_\times$ ako je već bilo posjećeno. U Algoritmu 2 u liniji 17 ne odbacujemo već posjećeno stanje, već njegovoj pripadajućoj listi listaPreth dodajemo trenutno stanje i pripadajući brid ako također osiguravaju najkraći put. U slučaju da prvi put posjećujemo vrh (n, q), to znači da smo pronašli **prvi** najkraći put do njega pa inicijaliziramo pripadnu listu.

Složenost algoritma sastoji se od dva aspekta:

1. složenost BFS algoritma na $G_\times$ je $O(|\mathcal{A}| \cdot |G|)$. To vrijedi jer nakon što ih ponovno posjetimo, vrhovi u $G_\times$ ne dodaju se u red Otvoreni,
2. ispisivanje svih puteva: opet je to jednako $|[\![q]\!]_G|$, što je za razliku od Teorema 2.2.1 zasigurno dosta veći broj.

Također, uočimo da funkcija svaki put koji pronađe u G prođe točno jednom što osigurava točnost rješenja izračunavanja.

Dakle, ukupna složenost algoritma i u ovom je slučaju $O(|\mathcal{A}| \cdot |G| + |[\![q]\!]_G|)$ pa vrijedi sljedeći rezultat:

Teorem 2.1.3. *Neka je G grafovska baza podataka, i q upit oblika*

$$\mathtt{SVI\ NAJKRAĆI\ ŠETNJA}(v, \mathtt{regex}, ?x)$$

Ako je $\mathcal{A}$ automat za `regex` ***koji je nedvosmislen i ima jedinstveno završno stanje**, onda Algoritam 2 ispravno računa $[\![q]\!]_G$ u vremenu*

$$O(|\mathcal{A}| \cdot |G| + |[\![q]\!]_G|).$$

2.2 `restrictor` $\in$ {STAZA, ACIKLIČAN, JEDNOSTAVAN}

U ovoj sekciji promatramo ostale slučajeve, kada je `restrictor` $\neq$ `ŠETNJA`. Za danu grafovsku bazu G i vrhove v, v' u njoj te regularni izraz `regex`, definiramo sljedeći jezik:

$$\mathtt{NEKI\ JEDNOSTAVAN\ PUT} = \{\langle G, v, v', \mathtt{regex}\rangle \mid \text{postoji jednostavan put } p \text{ od } v \text{ do } v' \text{ u } \mathcal{G} \\ \text{takav da je } \mathtt{oznaka}(p) \in \mathcal{L}(\mathtt{regex}) \}$$

Iz [2] možemo vidjeti da vrijedi da je taj jezik `NP`-potpun. Definirajmo jezik u kojem unaprijed fiksiramo `regex`:

$$\mathtt{NEKI\ JEDNOSTAVAN\ PUT(regex)} = \{\langle G, v, v'\rangle \mid \text{postoji jednostavan put } p \text{ od } v \text{ do } v' \text{ u } \mathcal{G} \\ \text{takav da je } \mathtt{oznaka}(p) \in \mathcal{L}(\mathtt{regex}) \}$$

Pokazat ćemo da vrijedi sljedeći rezultat

Teorem 2.2.1. *Neka je Σ abeceda i $w \in \Sigma^*$* ***duljine barem** 2. Izračunavanje jezika*

$$\mathtt{NEKI\ JEDNOSTAVAN\ PUT}\,(w)^*$$

je NP-potpuno.

Da bismo dokazali prethodni teorem, potreban nam je jači rezultat koji je dokazan u sljedećem potpoglavlju.

PROBLEM HOMEOMORFIZMA USMJERENOG PODGRAFA

Započinjemo s definicijom jezika HOMEOMORFIZAM PODGRAFA koja je slična u [7]:

Definicija 2.2.2. *Neka je $H = (V_H, E_H)$ graf. Za proizvoljni graf G označimo s $\texttt{Putovi}(G) := \{p\colon p \text{ je put u } G\}$. Problem homeomorfizma podgrafa za fiksni graf H definiramo kao sljedeći jezik:*

$$\begin{aligned} \mathit{HOMEOMORFIZAM\ PODGRAFA}(H) :=\{\langle G, v, e\rangle \mid\ & G = (V, E) \text{ graf, } v\colon V_H \to V \text{ injekcija,} \\ & e\colon E_H \to \texttt{Putovi}(G) \text{ injekcija t.d. su} \\ & (\forall\ a, b \in E_H)\ e(a) \text{ i } e(b) \text{ međusobno} \\ & \text{disjunktni putovi (osim ili početak ili kraj} \\ & \text{ako ga dijele)}\} \end{aligned}$$

*Ako je H **usmjereni** graf, definiramo HOMEOMORFIZAM USMJERENOG PODGRAFA(H) := HOMEOMORFIZAM PODGRAFA(H) $\cap \{\langle G, v, e\rangle \mid G$ **usmjeren** graf$\}$*

U literaturi se može naći da je nekoliko netrivijalnih instanci HOMEOMORFIZAM PODGRAFA riješeno algoritmom polinomne vremenske složenosti.

Međutim, mi se bavimo usmjerenim podgrafovima za koje rezultat koji dokazujemo kaže da je problem **usmjerenog** podgrafa za **usmjerene** grafove NP-potpun, osim u trivijalnim slučajevima kada je H usmjereno stablo dubine 1 (bridovi ili ulaze u ili izlaze iz korijena) - svodi se na provjeru postojanja puta između dva čvora, što je polinomno. Nastavljamo s dokazivanjem problema **usmjerenog** podgrafa za **usmjerene** grafove NP-potpun, osim kada je H stablo dubine 1.

Do kraja dokaza potrebne su 3 leme iz kojih će slijediti dokaz.

Lema 2.2.3. *Neka je P podgraf od Q i HOMEOMORFIZAM PODGRAFA(P) NP-težak. Tada je i HOMEOMORFIZAM PODGRAFA(Q) NP-težak.*

Dokaz. Za $P = (V_P, E_P)$ usmjereni podgraf od usmjerenog grafa $Q = (V_Q, E_Q)$ želimo pokazati da vrijedi

$$\texttt{HOMEOMORFIZAM PODGRAFA}(P) \leq_p \texttt{HOMEOMORFIZAM PODGRAFA}(Q).$$

Dakle, neka su dani graf $G = (V, E)$, injekcija $g\colon V_P \to V$ i injekcija $m\colon E_H \to \texttt{Putovi}(G)$. Cilj je u polinomnom vremenu konstruirati graf $H = (V_H, E_H)$, injekciju $h\colon V_Q \to V_H$ i injekciju $n\colon E_Q \to \texttt{Putovi}(H)$ tako da vrijedi (HP je skraćena oznaka za HOMEOMORFIZAM PODGRAFA):

$$\langle G, g, m\rangle \in \mathrm{HP}(P) \iff \langle H, h, n\rangle \in \mathrm{HP}(Q)$$

Neka je $Q \backslash P$ graf koji se sastoji od bridova u Q koji nisu u P, zajedno sa svim susjednim vrhovima. Konstruiramo H na sljedeći način: dodamo u G kopiju od $Q \setminus P$, gdje vrh n od

$Q \backslash P$ koji je također vrh od P preslikan u $g(n)$ u G. g proširimo u h tako da na svim vrhovima iz P ostavimo iste vrijednosti, a sve vrhove iz $Q \setminus P$ preslikamo u odgovarajuće u $H \setminus G$ što je upravo jednako $Q \setminus P$. Proširimo preslikavanje m u preslikavanje n na sljedeći način: za sve bridove iz E_G ostavimo iste vrijednosti, a sve nove bridove a' iz $Q \setminus P$ preslikamo u skup koji sadrži samo put duljine 1 te njegov brid označimo s a'.

Dakle, po konstrukciji vrijedi smjer $\implies$. Dokažimo obratni smjer.

Obratnu tvrdnju dokazujemo indukcijom po broju bridova u $Q \setminus P$. Ako je $Q \setminus P$ prazan, tvrdnja očito vrijedi jer nema novih bridova koji bi se trebali preslikati u nove skupove međusobno disjunktnih skupova.

Pretpostavimo da je $Q \setminus P$ neprazan i $\langle H, h, n \rangle \in HP(Q)$.

Ako je za svaki brid a u P $n(a)$ skup disjunktnih puteva u G (po definiciji je u H, ne mora biti u G), onda tvrdnja odmah vrijedi. Dakle, pretpostavimo da za neki brid p iz P vrijedi da $n(p)$ sadrži put koji nije u G. Jedini dodavani putevi su oni duljine 1 pa $n(p)$ sadrži **put koji prolazi samo jednim bridom** q' koji je u H i nije u G. S obzirom da slika bilo kojeg brida a iz $Q \setminus P$ s barem jednim krajem koji nije u P može kao svoju sliku imati **jedino** odgovarajući put s jednim bridom a' u H (ili paralelan mu brid), oba kraja od q' moraju biti u G. Da bi oba kraja od q' bila u G, ali ne i sam brid q', morao je postojati brid q iz $Q \setminus P$ koji je prisilio q' da bude dodan u H. Brid q' mora biti cijela slika od p, jer inače homeomorfizam h ne bi preslikavao u međusobno disjunktne puteve; dakle, p i q su paralelni u Q. Preslikavanje h' formirano **međusobnim mijenjanjem vrijednosti** $h(p)$ i $h(q)$ je homeomorfizam iz Q u H. Restrikcijom domene od h' na $Q \setminus \{q\}$ pokazali smo da je $Q \setminus \{q\}$ homeomorfan podgrafu od $H \setminus \{q'\}$ pa po pretpostavci indukcije vrijedi da je P homeomorfan podgrafu od G. □

Dokaz je isti za P i Q usmjerene grafove.

Lema 2.2.4. *Promotrimo podgraf na Slici 2.1. Pretpostavimo da postoje dva međusobno disjunktna puta koji prolaze kroz podgraf - jedan izlazi iz* A *i drugi ulazi u* B*. Tada put koji izlazi iz* A *je morao početi u* C *i put koji prolazi kroz* B *morao je izaći u* D*. Nadalje, postoji točno jedan dodatni put kroz podgraf i on je neki od sljedeća dva:*

$$8 \to 9 \to 10 \to 4 \to 11 \quad \textit{ili} \quad 8' \to 9' \to 10' \to 4' \to 11',$$

ovisno o putu koji izlazi iz A*.*

Dokaz. Nazovimo put koji izlazi iz A, A-put. On mora koristiti brid 1 ili 1′. Podgraf je simetričan pa pretpostavimo da koristi brid 1. Dakle, također mora koristiti brid 2.

Put koji ulazi u B zovemo B-put i on ne može koristiti brid 6, dakle mora koristiti brid 6′ i brid 2′. Ne može koristiti brid 1′ pa mora po bridu 7′ u bridu 9. A-put ne može po 6, pa mora po 3 i 4. Ne može koristiti brid 10 pa mora po bridu 5 i ući kroz C. B-put ne može koristiti brid 10 pa mora koristiti brid 12 i izaći u D. Put $8 \to 9 \to 10 \to 4 \to 11$ je u

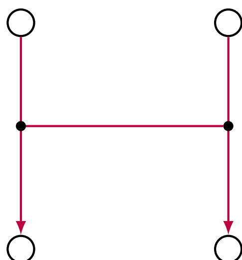
Slika 2.1: Podgraf "sklopka" s označenim bridovima

ovom slučaju blokiran i $8' \to 9' \to 10' \to 4' \to 11'$ je slobodan. Uočimo da ako put ulazi u $8'$, mora izaći u $11'$ s obzirom da su $3'$ i $12'$ blokirani. Slično, ako put izlazi u $11'$ mora ući u $8'$. □

Podgraf na Slici 2.1 zove se "sklopka". Možemo nanizati proizvoljno mnogo sklopki i opet primijeniti lemu tako da spojimo bridove `C` i `D` od jedne sklopke s bridovima `A` i `B` druge i tako dalje redom. Ideja sklopke je sljedeća: ako vizualiziramo graf kao na Slici 2.2 s četiri vrha i dva brida, onda, kako je u ovom problemu moguće koristiti samo jedan od putova $8 \to 9 \to 10 \to 4 \to 11$ ili $8' \to 9' \to 10' \to 4' \to 11'$, tako se i u tom grafu može putovati po vertikalnim bridovima, dok okomita linija simbolizira da se ne može po oba istodobno.

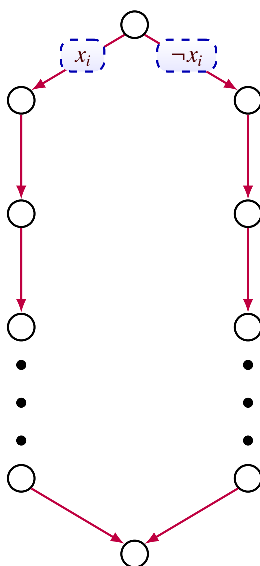
Dalje slijedi posljednja potrebna lema.

Lema 2.2.5. *Neka je usmjereni graf P sadržan od dva disjunktna usmjerena brida i četiri susjedna vrha. Tada je jezik* `HOMEOMORFIZAM USMJERENOG PODGRAFA`(P) *NP-težak.*



Slika 2.2: Shema sklopke

Dokaz. Svest ćemo poznati $3 - \texttt{CNF}$ problem na problem homeomorfizma podgrafa za zadani P iz iskaza. Fiksirajmo formulu F s varijablama $x_1 \ldots x_k$ i klauzulama $t_1 \ldots t_l$. Konstruiramo graf G_F na sljedeći način:



Slika 2.3: Podgrafovi za svaku varijablu x_i

Za svaku varijablu x_i napravimo podgraf kao sa Slike 2.3.

Definiramo da visina stupca bude jednaka ukupnom broju pojavljivanja literala x_i i $\neg x_i$. S obzirom da je k literala, tako nanižemo i spojimo bridom tih k podgrafova: najdonji vrh podgrafa za x_i s najgornjim za x_{i+1} i tako dalje.

Osim toga u graf dodamo vrhove $n_0, \ldots, n_l$ koji odgovaraju klauzulama $t_1, \ldots, t_k$ pri čemu iz svakog n_i izlaze 3 brida i ulaze u n_{i+1}, za svaki i. Također je tu i brid od najdonjeg vrha zadnjeg podgrafa za x_k do n_0.

Sada za svaki literal y koji se pojavljuje u klauzuli t_i zamijenimo jedan od bridova između n_{i-1} i n_i i jedan brid u stupcu pridružen y sa sklopkom.

Konačno, pokrijmo eventualne vrhove sklopki koji nemaju i izlazni i ulazni brid. Takva su četiri pa još dodamo vrhove označene s `W, X, Y` i `Z`. Brid iz Y identificiramo s ulazom `B` u prvu sklopku, brid s izlaza `D` od posljednje sklopke spojimo s najgornjim vrhom podgrafa od x_1 i još dodamo brid od n_l do `Z`. `C` ulaz od zadnje sklopke spojen je brid koji ide od `W` do njega i `A` izlaz od prve sklopke spojimo s bridom da ide u `X`. Primjer je pokazan na Slici 2.4 i označimo ovakav graf s G_F.

Tvrdimo da postoje dva međusobno disjunktna puta od `W` do `X` i od `Y` do `Z` u G_F ako i samo ako je fomula F ispunjiva, tj. postoji interpretacija I tako da je $I(F) = 1$. Pretpostavimo da je F ispunjiva. Tada put od Y do Z može proći jedino kroz stupac s $\neg y$ ako i samo ako je $I(y) = 1$. Tada s obzirom da je u svakoj klauzuli t_i barem jedan literal zadovoljen, uvijek će postojati barem jedan put sklopke koristan za ići od n_{i-1} do n_i. Put od W do X onda prolazi po svim sklopkama po onom putu po kojem drugi put nije prošao. Obratno, ako postoje dva međusobno disjunktna puta od `W` do `X` i od `Y` do `Z` u G_F, onda moraju prolaziti kroz sklopke kao u Lemi 2.2.4. Dakle, put od `Y` do `Z` mora proći kroz sve podgrafove od $x_i$eva i kroz vrhove n_0 do n_l. Tako definiramo i interpretaciju: postavimo da je $I(y) = 1$ ako i samo ako put od Y do Z koristi stupac pridružen $\neg y$. Konstrukcija ovog grafa je izračunljiva u polinomnom vremenu, pa je time dokaz gotov. □

Primijenimo sve leme na krajnji cilj i glavni rezultat.

Teorem 2.2.6. *Neka je P bilo koji usmjereni graf (osim trivijalnih slučajeva). Tada je* `HOMEOMORFIZAM USMJERENOG PODGRAFA`(P) *NP-potpun.*

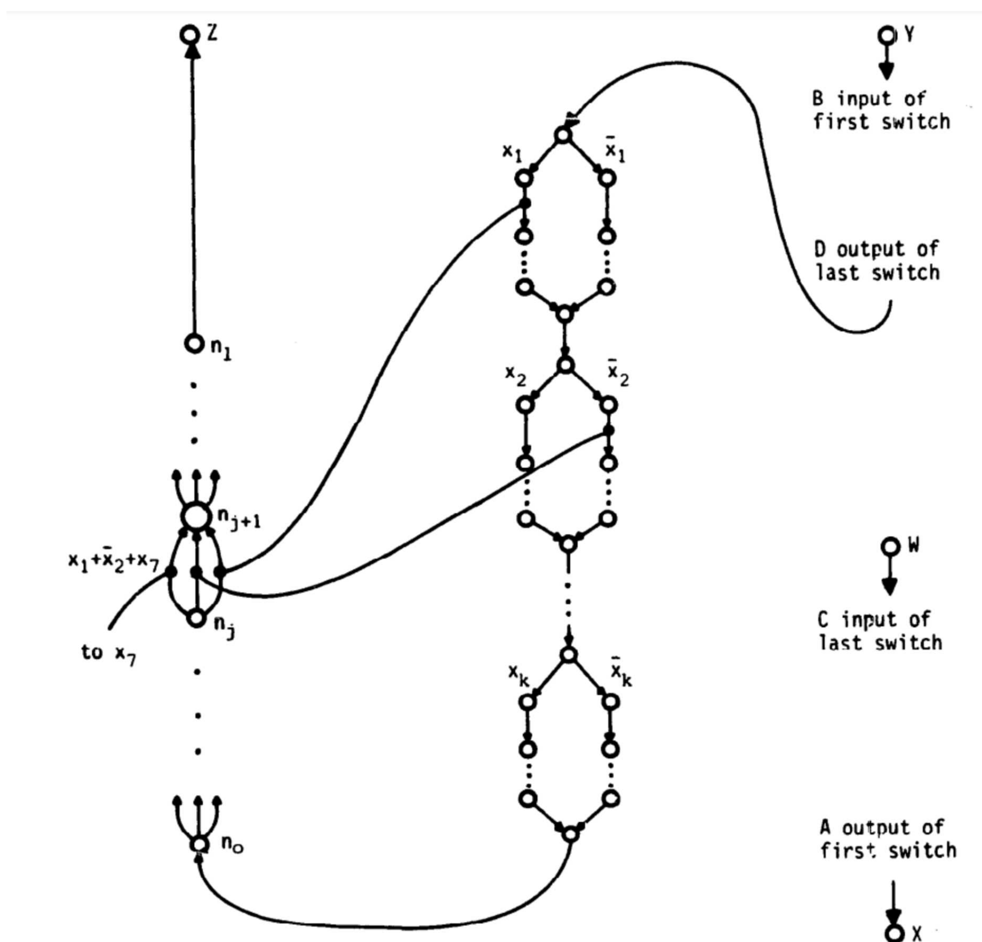
Dokaz. Jezik je u `NP` jer za $\langle G, v, e\rangle \in$ `HOMEOMORFIZAM USMJERENOG PODGRAFA`(P) jer se u polinomno vremena provjeri jesu li v i e injekcije s traženim svojstvima. Svi slučajevi usmjerenih grafova (osim trivijalnih) su ako graf sadrži neki od sljedećih podgrafova:

1. dva disjunktna brida, jedan ili oba mogu biti petlja,

2. put od dva brida koji posjećuje tri različita vrha ili

3. ciklus duljine 2.

Ako pokažemo da je problem `NP`-potpun za svakog od prethodna 3 slučaja, onda primjenom Leme 2.2.3 slijedit će tvrdnja teorema. Lema 2.2.5 je riješila prvi slučaj kada nema petlji. Ako bi bilo petlji, onda bismo identificirali u grafu G_F W s X i/ili Y s Z pa bismo opet mogli primijeniti istu konstrukciju. Za drugi slučaj, identificiranje u G_F X s Y rješava problem i konačno u trećem slučaju u G_F identificiramo parove vrhova `W, Z` i `X, Y` pa možemo lemu primijeniti i u ovom slučaju. □



Slika 2.4: Primjer izgleda grafa G_F iz Leme 2.2.5, izvađeno iz [4]

JEDNOSTAVNI PARNI PUTEVI

Primjenimo sada Teorem 2.2.6 u slučaju kada je P usmjereni put duljine 2, odnosno $P = (\{u, v, w\}, \{(u, v), (v, w)\}$, kao što je pokazano u [6]. Neka je $\langle G, h, e\rangle \in$ HOMEOMORFIZAM USMJERENOG PODGRAFA(P) (odnosno, skraćeno $\langle G, h, e\rangle \in$ HUP(P)), proizvoljan. S obzirom da je h injekcija, to znači da vrijedi

$$\langle G, h, e\rangle \in \texttt{HUP}(P) \iff \text{postoji jednostavni put od } h(u) \text{ do } h(w) \text{ preko } h(v)$$

Motivirani ekvivalencijom definiramo jezik:

$$\texttt{PUT PREKO VRHA} := \{\langle G, s, t, m\rangle \mid G \text{ usmjereni graf u kojem postoji jednostavni put od } s \text{ do } t \text{ preko } m\}.$$

Dakle, dokazali smo HOMEOMORFIZAM USMJERENOG PODGRAFA$(p) \leq_p$ PUT PREKO VRHA.

Po definiciji: PUT PREKO VRHA je NP-potpun.

Definiramo još jedan problem:

$$\texttt{JEDNOSTAVNI PARNI PUTEVI} := \{\langle G, s, t\rangle \mid G \text{ usmjereni graf, od } s \text{ do } t \text{ postoji jednostavni put parne duljine}\}$$

Dokažimo da je i on NP-potpun svođenjem s jezika PUT PREKO VRHA.

Teorem 2.2.7. *JEDNOSTAVNI PARNI PUTEVI je NP-potpun jezik.*

Dokaz. Želimo svesti u polinomnom vremenu PUT PREKO VRHA na JEDNOSTAVNI PARNI PUTEVI. Neka je tako $\langle G, s, t, m\rangle \in$ PUT PREKO VRHA. Iz $G = (V, E)$ konstruiramo novi usmjereni graf $G' = (V', E')$:

$$\begin{aligned} V' &= (V \setminus \{m\}) \times \{1, 2\}) \cup \{m\}, \\ E' &= \{((u, 2), (v, 1)) \mid (u, v) \in E\} \cup \{((u, 2), m) \mid (u, m) \in E\} \\ &\quad \cup \{(m, (u, 1)) \mid (m, u) \in E\} \cup \{((u, 1), (u, 2)) \mid u \in V \setminus \{m\}\} \end{aligned}$$

Promotrimo graf G' kada iz njega izbacimo sve bridove kojima je jedan od vrhova m. Svi bridovi u tome slučaju su između skupova $\{(u, 2) \mid u \in V\}$ i $\{(u, 1) \mid u \in V\}$ pa je to zapravo bipartitni graf. Dakle, svaki put koji ide iz jednog skupa i završava u drugom je neparne duljine, **osim ako** ne prođe preko m. Dakle, vrijedi ekvivalencija:

$$\text{postoji put parne duljine od } (s, 1) \text{ do } (t, 2) \text{ u } G' \iff \text{postoji put od } s \text{ do } t \text{ preko } m \text{ u } G$$

Time je dokaz gotov. □

Sada imamo sve potrebno za dokazati Teorem 2.2.1

Dokaz. Pokažimo najprije da je jezik iz NP.

Neka je $\langle G, v, w\rangle \in$ NEKI JEDNOSTAVNI PUTEVI(w^*). Rezultat dokazujemo primjenom Teorema o certifikatu koji je dokazan u [11]. Uzmimo da je certifikat put p koji počinje u v i završava u w. Sada je samo potrebno u $O(|p|)$ provjeriti je li to zaista put od v do v'. Dakle, po Teoremu o certifikatu, jezik je iz NP.

Neka je Σ konačna abeceda i w riječ iz Σ^* duljine barem 2. Pokažimo da je jezik NEKI JEDNOSTAVAN PUT $((w)^*)$ NP-potpun. Uzmimo u obzir najjednostavniji slučaj, kada je $\Sigma = \{0\}$ i $w = 00$. Tada regex $(00)^*$ generira jezik svih riječi parne duljine sastavljenih samo od 0, odnosno RPQ vraća sve **jednostavne** puteve čiji je izvor v parne duljine. Za graf $G = (V, E)$, $v \in V$, označimo jezik svih puteva parne duljine:

$$\texttt{PARNI PUTEVI(v)} := \{\langle p, v'\rangle \mid v' \in V,\ |p| \text{ paran i jednostavan put od } v \text{ do } v'\}$$

Dakle, pokazali smo da vrijedi

$$\texttt{PARNI PUTEVI}(v) \leq_p \texttt{NEKI JEDNOSTAVAN PUT}((00)^*)$$

pa je time dokaz gotov. □

Stoga, čak i ako fiksiramo točno određene `regexe`, opet ne možemo izbjeći eksponencijalnu složenost.

Dakle, "najbolji" poznat algoritam za traženje takvih putova je ispitivanje svih mogućih putova u produktnom grafu. Međutim, grafovi iz stvarnog svijeta nisu toliko komplicirani kako ćemo pokazati u eksperimentima.

Najprije ćemo pokazati algoritam u slučaju `selector` == (`SVI NAJKRAĆI`)?.

`selector` == (`SVI NAJKRAĆI`)?

U ovom odjeljku promatramo algoritam kada je `selector` == (`SVI NAJKRAĆI`)? i `restrictor` ∈ {`STAZA`, `ACIKLIČAN`, `JEDNOSTAVAN`}.

Algoritam 3 Izračunavanje RPQ upita (`SVI NAJKRAĆI`)? `restrictor` (v, `regex`, ?x) na graf. bazi $G = (V, E, \rho, \lambda)$ pri čemu je `restrictor` ∈ {`STAZA`, `ACIKLIČAN`, `JEDNOSTAVAN`}

```
 1: function SviRestringiraniPutovi(G, q)
 2:     𝒜 ← konstruirajAutomat(regex)        ▷ q0 početno stanje, F skup završnih stanja
 3:     inicijaliziraj red/stog Otvoreni
 4:     inicijaliziraj rječnik Posjećeni
 5:     inicijaliziraj skup DostignutoZavršno
 6:     početnoStanje ← (v, q0, 0, null, ⊥)
 7:     Posjećeni.push(početnoStanje)
 8:     Otvoreni.push(početnoStanje)
 9:     if v ∈ V and q0 ∈ F then
10:         DostignutoZavršno.add(v)
11:         Rješenja.add(⟨v, 0⟩)
12:     end if
13:     while Otvoreni ≠ ∅ do
14:         trenutni ← Otvoreni.pop()  ▷ trenutni = (n, q, dubina, brid, prethodni)
15:         for idući = (n′, q′, edge′) ∈ Susjedi(trenutni, G, 𝒜) do
16:             if valjanPrijelaz(trenutni, idući, restrictor) then
17:                 novi ← (n′, q′, dubina + 1, brid′, trenutni)
18:                 Posjećeni.push(novi)
19:                 Otvoreni.push(novi)
20:                 if q′ ∈ F then
```

```
21:                 put ← PRONAĐIPUT(novi)
22:                 if !(SVI NAJKRAĆI) then
23:                     Rješenja.add(put)
24:                 else if n′ ∈ DostignutoZavršno then
25:                     DostignutoZavršno.add(⟨n′, dubina + 1⟩)
26:                     Rješenja.add(put)
27:                 else
28:                     optimalna ← DostignutoZavršno.get(n′).dubina
29:                     if dubina + 1 == optimalna then
30:                         Rješenja.add(put)
31:                     end if
32:                 end if
33:             end if
34:         end if
35:     end for
36:     end while
37:     return Rješenja
38: end function
39:
40: function VALJANPRIJELAZ(stanje, idući, restrictor)
41:     s ← stanje
42:     while s ≠ ⊥ do
43:         if restrictor == ACIKLIČAN then
44:             if s.n == idući.n then
45:                 return False
46:             end if
47:         else if restrictor == JEDNOSTAVAN then
48:             if s.n == idući.n and s.prethodni ≠ ⊥ then
49:                 return False
50:             end if
51:         else if restrictor == STAZA then
52:             if s.brid == idući.brid then
53:                 return False
54:             end if
55:         end if
56:         s ← s.prethodni
57:     end while
58:     return True
59: end function
```

Kao i uvijek, algoritam iz `regex`a najprije konstruira automat $\mathcal{A}$ s početnim stanjem q_0. Intuitivno, on nabraja sve puteve u produktnom grafu i odbacuje one koji pritom ne zadovoljavaju `restrictor`. Da bismo imali ispravnost algoritma, opet zahtijevamo da $\mathcal{A}$ bude nedvosmislen.

Stanje traženja u ovome slučaju je hibrid stanja traženja Algoritma 1 i 2: petorka $(n, q, \mathit{dubina}, \mathit{brid}, \texttt{prethodni})$ pri čemu je:

- n vrh iz G koji promatramo,
- q stanje iz $\mathcal{A}$ u kojem smo trenutno,
- *brid* je brid s kojim smo došli u n,
- *dubina* duljina (bilo kojeg) najkraćeg puta od n do v,
- `prethodni` je pokazivač na prethodno stanje traženja.

Na početku inicijaliziramo varijable:

- `Otvoreni`: kao i do sada, red za stanja traženja u slučaju `BFS`a kada je `selector` == `SVI NAJKRAĆI`, stog u slučaju `DFS` kada `selector`*a* nema.
- `Posjećeni`: skup već posjećenih stanja traženja - u prethodnim algoritmima služio je za odbacivanje rješenja, a ovdje ne služi ničemu jer se odbacivanje provjerava pomoćnom funkcijom. Ipak, nije izbačen jer je dosta koristan za otkrivanje pogrešaka u implementaciji.
- `DostignutoZavršno`: u Algoritmu 1 ovo je bio skup, a ovdje će biti **rječnik** parova (n, dubina) jer čim prvi put dođemo do n, to će zbog `BFS`a biti i najkraći put pa je ovo mjesto gdje ćemo spremiti najkraću dubinu. U slučaju `DFS`a, linija 22 će biti ispunjena pa ova varijabla nema nikakvu ulogu.

Pomoćna funkcija u linijama 40-59 rekurzivno provjerava hoće li dodavanje svakog novog vrha zadovoljiti `restrictor`. Krene od kraja i završi na početku, pritom svaki vrh/brid uspoređujući s potencijalnim novim vrhom/bridom te ga ovisno o (ne)poklapanju prihvati ili odbaci. Napomenimo da se to odnosi na put u običnom grafu G, ne u produktnom $G_\times$.

Ako je prijelaz valjan, onda konstruiramo novo stanje traženja koje gurnemo u `Posjećeni` i `Otvoreni`. Ako je stanje automata u stanju traženja bilo završno, pronašli smo novi put koji kao i u Algoritmu 1 rekonstruiramo iz stanja traženja `novo`. Međutim, još nismo sigurni spremamo li ga u rješenja:

1. ako ne zahtijevamo `selector` == `SVI NAJKRAĆI`, onda spremamo,

2. inače ako smo prvi put dostigli završno stanje s tim vrhom iz G, onda je to sigurno najkraći put pa ga spremamo. Usput spremamo i u `DostignutoZavršno`.
3. inače ako smo već bili s tim vrhom u završnom stanju, onda moramo da je dubina optimalna. Ako jest, spremamo rješenje i u tom slučaju.

Što se složenosti tiče, opet zahtijevamo da je automat $\mathcal{A}$ nedvosmislen jer inače bismo ista rješenja brojili više puta. Iako uvjet zaustavljanja nije provjeravan preko skupa `Posjećeni`, zasigurno je konačno putova koji mogu biti valjan za svaki od `restrictor`a pa se algoritam sigurno zaustavlja. Nažalost, zbog toga u najgorem slučaju moramo proći sve moguće puteve u produktnom grafu.

Složenost u ovom slučaju sastoji se od:

1. obilaženja svih mogućih putova - svaki je duljine najviše $|V| + 1$ jer se (po uvjetima restriktora) vrhovi iz G ne ponavljaju (osim eventualno prvog i zadnjeg u slučaju `restrictor = ACIKLIČAN`) te je svaki vrh u $G_\times$ koji se posjeti na tom putu jedan od njih $|V| \cdot |Q|$. Ovdje je ključna nedvosmislenost, inače bi se identični putovi mogli pronaći što je svakako. Stoga, svi mogući putovi obilaze se u $O(|(\mathcal{A}| \cdot |G|)^|G|)$
2. kao i uvijek do sad, očito još dodamo $|[\![q]\!]_G|$.

Stoga vrijedi sljedeći rezultat.

Teorem 2.2.8. *Neka je G grafovska baza podataka, i q upit oblika*

$$(\mathit{SVI\ NAJKRAĆI})?\ \mathtt{restrictor}(v, \mathtt{regex}, ?x)$$

pri čemu je `restrictor` $\in$ {*STAZA*, *JEDNOSTAVAN*, *ACIKLIČAN*}*. Ako je* $\mathcal{A}$ *automat za* `regex` *koji je nedvosmislen, onda Algoritam 3 ispravno računa* $[\![q]\!]_G$ *u vremenu*

$$O\left((|\mathcal{A}| \cdot |G|)^{|G|} + |[\![q]\!]_G|\right).$$

`selector` == `NEKI (NAJKRAĆI)?`

Jedino nam je preostao slučaj kada je `selector` `NEKI` ili `NEKI NAJKRAĆI`. Srećom, to zahtjeva minimalnu promjenu Algoritma 3. S obzirom da tražimo **neki** put, više nije potrebno pamtiti dubinu pa tako `DostignutoZavršno` može biti običan skup, a ne rječnik. Tada je linije 21-29 dovoljno zamijeniti dodavanjem vrha n' u `DostignutoZavršno` ako n' nije već u tom skupu i pritom u `Rješenja` dodati trenutni otkriveni put. Dakle, kada pronađemo neki put do n', onda više ne spremamo kao rješenja putove koji dolaze do n'. Standarno već, `selector` je `NEKI NAJKRAĆI` u slučaju `BFS`a, a `NEKI` u slučaju `DFS`a. Primijetimo da ovdje više nije nužno da automat bude nedvosmislen jer vraćamo samo **neki** put. Isto kao u prethodnom slučaju, u najgorem slučaju svi obilazimo sve puteve u produktnom grafu pa, i ovdje, vrijedi sličan rezultat složenosti:

Teorem 2.2.9. *Neka je G grafovska baza podataka, i q upit oblika*

$$\mathit{NEKI}\ (\mathit{NAJKRAĆI})?\ \texttt{restrictor}(v, \texttt{regex}, ?x)$$

pri čemu je $\texttt{restrictor} \in \{\mathit{STAZA}, \mathit{JEDNOSTAVAN}, \mathit{ACIKLIČAN}\}$*. Tada možemo računati* $[\![q]\!]_G$ *u vremenu*

$$O\left((|\mathcal{A}| \cdot |G|)^{|G|} + |[\![q]\!]_G|\right).$$

Poglavlje 3

Implementacija

Navedeni algoritmi implementirani su u programskom jeziku Python. Ispod prikazujemo strukturu repozitorija koji je implementiran za ovu upotrebu. Repozitorij je javno dostupan na `https://github.com/mdujic/diplomski` te ga se može korištenjem sustava za verzioniranje `Github` kopirati naredbom

```
git clone https://github.com/mdujic/diplomski.git
```

```
/rpq_evaluation/
├── graph_database/
│   ├── core/
│   │   ├── __init__.py
│   │   └── trie.py
│   │       └── class Trie
│   └── graph.py
│       ├── class Edge
│       ├── class EdgeTrie
│       └── class Graph
├── regex_transform/
│   ├── __init__.py
│   └── automaton.py
│       └── class Automaton
├── tests/
│   └── ...
└── __init__.py
```

S obzirom da smo u svim algoritmima zahtijevali da ne postoje ϵ-prijelazi, pretvaranje iz regularnog izraza u `NKA` provedeno je Glushkovovim algoritmom koji to, za razliku od

Thompsonova algoritma, osigurava. Za to je korištena vanjska biblioteka `FAdo`. Opišimo ideju algoritma i promotrimo primjenu kakva je u navedenoj biblioteki. Sve operacije pretvorbe `regexa` u `NKA` odvijaju se u klasi `Automaton`.

3.1 Glushkovov algoritam

Za dani regularni izraz e, Glushkovov algoritam konsturira nedeterministički automat koji prihvaća $\mathcal{L}(e)$. Konstrukcija se odvija u 4 koraka, koji odmah pokazujemo na primjeru za $e = (a(ab)^*)^* + (ba)^*$:

1. Izraz "lineariziramo": svako slovo abecede koje se pojavljuje u izrazu preimenujemo, tako da se svako slovo pojavljuje najviše jednom u novom izrazu e'. Označimo s A staru abecedu, a B novu: $B = \{a_1, a_2, b_3, b_4, a_5\}$. Novi e' može izgledati ovako:

$$e' = (a_1(a_2b_3)^*)^* + (b_4a_5)^*$$

2. Neformalno definiramo i odmah računamo $P(e')$, $D(e')$ i $F(e')$:

$$\begin{aligned} P(e') &= \{\texttt{prva slova riječi iz } L(e')\} = \{a_1, b_4\} \\ D(e') &= \{\texttt{zadnja slova riječi iz } L(e')\} = \{a_1, b_3, a_5\} \\ F(e') &= \{\texttt{susjedni dvoslovi u } L(e')\} \\ &= \{a_1a_2, a_1a_1, a_2b_3, b_3a_1, b_3a_2, b_4a_5.a_5b_4\} \\ \Lambda(e') &= \{\epsilon\} \cap L(e') = \{\epsilon\} \end{aligned}$$

3. Konstruiramo automat od:

   - početnog stanja (ne nalazi se u B) kojeg označimo `Početno`,
   - skupom stanja $\{$`Početno`$\} \cup B$,
   - relacijom prijelaza $\{(x, y, y) \mid xy \in F(e')\} \cup \{($`Početno`$, y, y \mid y \in P(e')\}$,
   - skupom završnih stanja $D \cup \{$`Početno`$\}$ (napomena: početno stanje je uključeno jer $\Lambda(e') = \{\epsilon\}$, inače samo D)

4. Uklonimo "linearizaciju": za svako slovo $t \in B$ svako $(*, t, *)$ iz relacije prijelaza zamijenimo odgovarajućim $(*, s, *)$ pri čemu je linearizacijom s bio postao t. Iz teorije automata: prethodni automat prepoznavao je tzv. *lokalni jezik* $(PB^* \cap B^*D) \setminus B^*(B^2 \setminus F)B^*)$ pa vrijedi da automat nakon postupka reverznog "linearizaciji" prepoznaje početni jezik za koji smo tražili automat, $\mathcal{L}(e)$.

Izvedimo sada algoritam iz biblioteke FAdo. Pretpostavljamo da radimo na operacijskom sustavu Linux koji ima instaliran Python i pip. Najprije instaliramo FAdo biblioteku korištenjem naredbe `pip install fado`. Čitanjem dokumentacije na [8], vidimo da klasa NFA u biblioteci FAdo ima sljedeće varijable:

- States (lista) - skup stanja,
- sigma (skup) - abeceda,
- Initial (skup) - skup indeksa početnih stanja,
- Final (skup) - skup indeksa završnih stanja,
- delta (rječnik) - relacija prijelaza.

Učitavanjem regularnog izraza naredbama

```
from FAdo.fa import *
from FAdo.reex import *
from FAdo.fio import *

r = str2regexp("(a(ab)*)* + (ba)*")
nfa = r.nfaGlushkov()
```

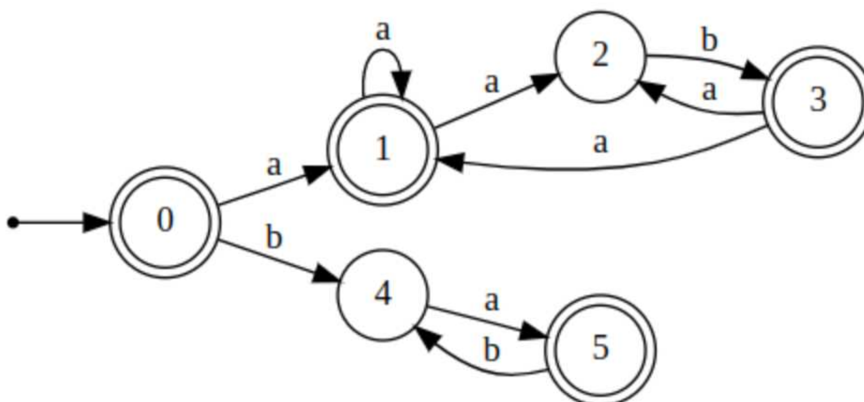
ispisuje se novonastali objekt

```
NFA((
['Initial', 'a', '2', 'b', '4', '5'],
['b', 'a'],
['Initial'],
['Initial', 'a', 'b', '5'],
"[('a', 'a', 'a'), ('a', 'a', '2'), ('2', 'b', 'b'),
('b', 'a', 'a'), ('b', 'a', '2'), ('Initial', 'b', '4'),
('Initial', 'a', 'a'), ('4', 'a', '5'), ('5', 'b', '4')]"))
```

a to se onda može vizualizirati naredbom `nfa.display()` gdje dobivamo sljedeći izlaz:

3.2 NKA s jedinstvenim završnim stanjem

U jednom od algoritama zahtijeva se da automat bude bez ϵ-prijelaza i s jedinstvenim završnim stanjem. Ekvivalentan automat možemo konstruirati na sljedeći način: za dani NKA bez ϵ-prijelaza



Slika 3.1: NKA dobiven Glushkovovim algoritmom

1. dodamo novo stanje f,
2. za svaki prijelaz (s, a, t) pri čemu je t završno stanje, u relaciju prijelaza dodamo (s, a, f),
3. skup završnih stanja proglasimo jednakim $\{f\}$.

Ako to izvedemo na automatu sa Slike 3.1, novi automat će izgledati kao na Slici 3.2.

Sada smo riješili sve probleme vezane za konstrukciju automata iz regularnog izraza pa je vrijeme da objasnimo kako konstruirati jezgru: **grafovsku bazu podataka**.

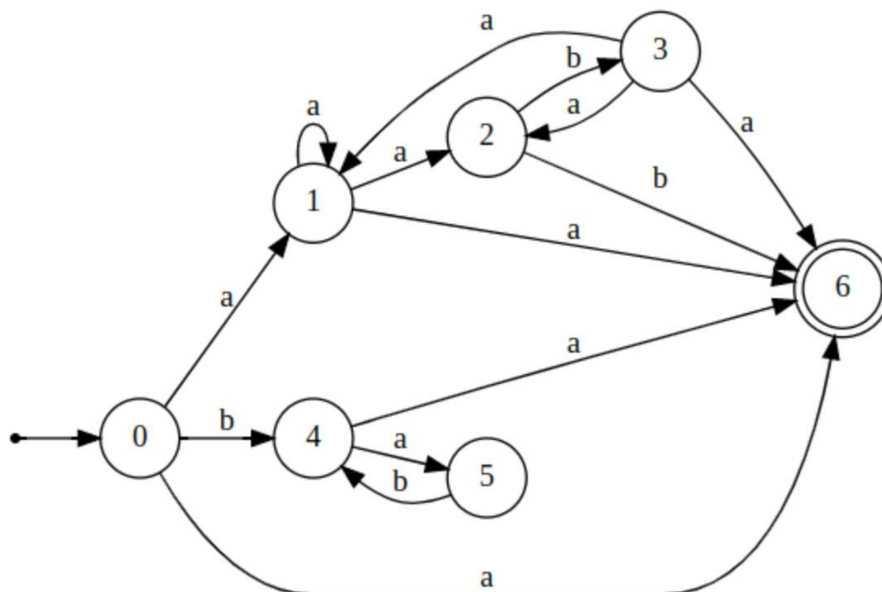
3.3 Implementacija grafovske baze podataka

Jezgra i osnovna struktura je klasa `Trie`, poznata i pod imenom **prefiksno stablo**. Ona se najčešće koristi za efikasnu pohranu i pretraživanje `string`ova. Upotreba se odnosi na pretraživanje, indeksiranje i funkcije automatskog dovršavanja.

Glavna prednost korištenja `Trie`ja je da omogućava brzo pretraživanje i dohvaćanje temeljeno na prefiksu. Smanjuje prostor pretraživanja tako da samo pretražuje grane stabla koje su potrebne.

U našem slučaju koristimo dvorazinski `Trie` koji pretpostavlja leksikografski poredak elemenata koje unosimo. Elementi su upravo bridovi. Važno je odmah istaknuti da za svaku oznaku brida formiramo zaseban `Trie`, tako da se u jednome objektu tog tipa nalaze samo bridovi koji dijele zajedničku oznaku.

Objasnimo obje razine:



Slika 3.2: `NKA` s jedinstvenim završnim stanjem dobiven opisanim postupkom na automat sa Slike 3.2.

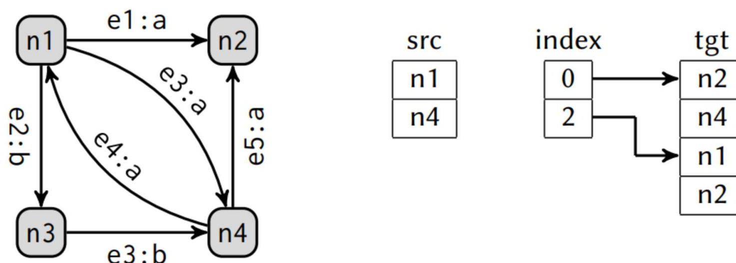
1. Razina 0 (razina prefiksa)

   - `prefix_lvl` je vektor koji sadrži prefikse - u ovom slučaju izvore bridova,
   - elementi se ne ponavljaju,
   - prefiksi se spremaju leksikografski da bi se traženje po njima vršilo binarnim, a ne sekvencijalnim traženjem.

2. Razina 1 (razina podataka)

   - `data_lvl` je vektor koji sadrži pridružene vrijednosti prefiksima - u ovome slučaju to su odredišta bridova,
   - `offsets` je vektor koji prati pomake unutar niza `data_lvl` za svaki prefiks i omogućuje efikasno pretraživanje svih bridova s istim vrhom,
   - raspon između `offsets[i]` i `offsets[i+1]` u `data_lvl` odgovara broju bridova kojima je izvor i.

Ista ideja implementacije `Trie`ja može se vidjeti u [3] pod imenom **Compressed Sparse Row** koju prikazujemo na Slici 3.3.



Slika 3.3: Primjer grafovske baze slijeva i pripadajući Compressed Sparse Row zdesna, izvađeno iz [3]. Vidimo da su u polju `src` indeksi svih vrhova iz kojih izviru bridovi s oznakom `'a'`, a to su `n1` i `n4`.

Time smo objasnili kako pohranjujemo bridove na najnižoj razini. Na višoj razini, svaki brid spremamo u objekt klase `Edge` koji ima varijable `source`, `target`, `label` i `id`.

Klasa `EdgeTrie` je razina apstrakcije iznad `Trie` u kojoj rješavamo problem spremanja oznaka. Naime, ona ima rječnik `trie_map` kojem ključeve predstavljaju oznake bridova, a vrijednosti su objekti klase `Trie` koji spremaju sve bridove s istom oznakom. Dakle, koliko god je različitih oznaka bridova, toliko je ključeva u rječniku.

Za kraj, klasa `Graph` je najviši sloj apstrakcije pomoću koje korisnik unosi graf i u kojem su implementirani svi RPQovi.

Pokažimo kako primjer sa Slike 1.1 bude spremljen u upravo objašnjenim klasama. To možemo napraviti primjerice da u datoteci `rpq_evaluation/tests/test_main.py` na liniji 18 otkomentiramo naredbu `breakpoint()`. Zatim pokrenemo testove naredbom
`pytest rpq_evaluation/tests/test_main.py`
Kada dostigne do točke prekida, imat ćemo dostupno konzolu u kojoj ćemo moći ispivati naredbe kao u nastavku.

Ispisujemo najprije `graph.edge_trie.trie_map` da bismo vidjeli kako su spremljene oznake:

```
(Pdb) graph.edge_trie.trie_map
{
    'a': <rpq_evaluation.graph_database.core.trie.Trie
    object at 0x7f361658d990>,
    'b': <rpq_evaluation.graph_database.core.trie.Trie
    object at 0x7f36163fa310>
}
```

Kao što vidimo, ključevi su sve oznake koje se pojavljuje u bazi, a vrijednosti su klase `Trie` koje sadržavaju sve bridove koje dijele istu oznaku. Ispišimo `Trie` i razmake za `'a'`.

```
(Pdb) graph.edge_trie.trie_map['a'].print()
Trie:

L0: 0,1,2,3,
L1: *1,5*3*1*2*

(Pdb) graph.edge_trie.trie_map['a'].print_offsets()
Offsets:

0: (0,2)
1: (2,3)
2: (3,4)
3: (4,5)
```

Dakle, na prvoj razini se nalazi svi prefiksi (izvori bridova), a na drugoj su pripadajuće vrijednosti (odredišta bridova). Na drugoj razini vrijednosti pridružene različitim prefiksima razdvojene su znakom `'*'`. Vidimo, primjerice, da iz vrha 0 izlaze dva brida s oznakom `'a'`, što znači da je i njihov razmak u drugoj razini jednak dva (stoga $(0, 2)$). Iz ostalih vrhova na prvoj razini izlazi samo jedan brid s oznakom `'a'`. Slično je i za oznaku `'b'`.

```
(Pdb) graph.edge_trie.trie_map['b'].print()
Trie:

L0: 0,1,2,3,
L1: *2*4*4*4*

(Pdb) graph.edge_trie.trie_map['b'].print_offsets()
Offsets:

0: (0,1)
1: (1,2)
2: (2,3)
3: (3,4)
```

Poglavlje 4

Eksperimenti

Eksperimenti koji ćemo izvoditi biti će ili na sintetičkim podacima koji će služiti kao "stres"-test, ili nad stvarnim podacima da vidimo kako se zapravo algoritmi ponašaju u situacijama koje se odvijaju u stvarnom svijetu.

Eksperimenti će se izvoditi se na operacijskom sustavu `Ubuntu 22.04` na računalu s 32 GB radne memorije i 13. generacijom Intelovog $i7$ procesora. Nalaze se u mapi `/rpq_evaluation/tests/`.

Konkretno, to su `Python` skripte `/rpq_evaluation/tests/test_diamond.py` u kojoj se nalaze eksperimenti nad sintetičkim podatcima te skripta `/rpq_evaluation/tests/test_facebook.py` koja služi eksperimentiranju nad pojednostavljenim grafom međusobnih prijateljstava određene skupine ljudi s društvene mreže `Facebook`.

Obje skripte imaju tri tipa testova uz koje ćemo odmah reći koje slučajeve upita pokrivaju:

1. `def test_any_walk(...)`,
   - `NEKI (NAJKRAĆI)? ŠETNJA`$(v,$ `regex`$, ?x)$,
2. `def test_all_shortest_walk(...)`,
   - `SVI NAJKRAĆI ŠETNJA`$(v,$ `regex`$, ?x)$,
3. `def test_restricted(...)`,
   - `(selector)? restrictor` $(v,$ `regex`$, ?x)$ tako da je `restrictor` $\neq$ `ŠETNJA` i `selector` proizvoljan.

Skripte se pokreću `Python`ovim okvirom `pytest` koji "olakšava pisanje malih, čitljivih testova i može se skalirati da podržava kompleksno funkcionalno testiranje aplikacija i biblioteka" (kao što stoji u [5]).

Osim što testovi mogu biti ručno pokretani, oni se mogu automatizirati, što je mogućnost koju sustav za verzioniranje `Github` pruža. Naime, datoteka `.github/workflows/python-app.yml` koju smo definirali služi da prilikom svakog slanja s lokalnog računala na udaljeni repozitorij (odnosno komande `git push`, testovi bivaju automatski pokrenuti te ako svi nisu zadovoljeni, korisniku se šalje mail s porukom na račun s kojim je registriran na `Github`. Time se osiguravamo da kod koji se piše prolazi neku vrstu provjere ispravnosti te korisnik ne mora stalno pokretati program da vidi je li on ispravno radi. Naime, dovoljno je napisati dovoljno općenite testove koji će pokretanjem samo jedne komande vratiti poruku jesu li svi zadovoljeni i može li korisnik neometano nastaviti raditi.

U testovima koji su napisani, omogućeno je istovremeno spremanje rezultata u datoteke i ispisivanje na komandnu liniju. Ipak, vremenski je zahtjevno pokretati svaki test više puta pa se i to može automatizirati. Naime, datoteka `/rpq_evaluation/testing.sh` je `bash` skripta koja služi upravo tome: `pytest` naredbu za svaku od navedenih skripti pokreće koliko god puta želimo (zadano 100 puta). Svako izvođenje neke od skripti traje $40 - 80$ sekundi pa je tako očekivano vrijeme eksperimenata $1 - 2$ sata po skripti, što znači sveukupno $2 - 4$ sata. Naravno, to je u slučaju računala nad kojim se vrše eksperimenti. Eksperimenti u našem slučaju su vršeni zasebno za sintetički i zasebno za realni skup podataka. Izvođenje 100 puta testova trajalo je:

- `test_diamond.py`: 5966.18 sek $\approx 99,43$ min $\approx 1,66$ sati,
- `test_facebook.py`: 6860.24 sek $\approx 114,33$ min $\approx 1,91$ sati,

što znači da je ukupno trajanje testova trajalo otprilike 3 i pol sata.

U nastavku prikazujemo sve `csv` tablice koji prilikom eksperimentiranja nastanu. To zgodno prikazujemo u `NerdTree`ju uređivača teksta `NeoVim` na Slici 4.

Napomenino, prije nego što prijeđemo na analizu rezultata, kako svaka funkcija ima implementirano završetak rada kada broj rezultata dosegne vrijednost varijable `limit` ili kada vrijeme izvršavanja dosegne vrijednost varijable `timeout`.

4.1 Umjetni primjer s eksponencijalno mnogo slučajeva

Promotrimo primjer baze sa Slike 4.2. Za nju vrijedi da su svakom $3k$-tom elementu pridružena dva susjeda, $3k + 1$ i $3k + 2$, a svaki $3k + 1$ i $3k + 2$ ima samo jednog susjeda $3k + 3$. S obzirom da izgledom asocira na dijamant, nadalje ovaj primjer zovemo "primjer dijamant". Za takvu bazu s ukupno $n = 3k + 1$ (u primjeru sa Slike 4.2 vrijedi $n = 13$,) vrhova lako je izračunati da vrijedi

$$|\texttt{SVI NAJKRAĆI ŠETNJA}\ (0, \mathtt{a}^*, ?x)| \in 2^{O(n)}.$$

```
nvim . 59x55
" Press ? for help

.. (up a dir)
/home/mateo/repos/diplomski/
▼ rpq_evaluation/
  ▶ __pycache__/
  ▶ graph_database/
  ▶ regex_transform/
  ▼ tests/
    ▶ __pycache__/
    ▼ test_diamond_results/
        all_shortest_walk_46_100000_60.csv
        all_shortest_walk_46_1_60.csv
        any_shortest_walk_1500_False_100000_60.csv
        any_shortest_walk_1500_False_1_60.csv
        any_shortest_walk_1500_True_100000_60.csv
        any_shortest_walk_1500_True_1_60.csv
        restricted_acyclic__46_100000_60.csv
        restricted_acyclic__46_10_60.csv
        restricted_acyclic_all_shortest_46_100000_60.csv
        restricted_acyclic_all_shortest_46_10_60.csv
        restricted_acyclic_any_46_100000_60.csv
        restricted_acyclic_any_46_10_60.csv
        restricted_acyclic_any_shortest_46_100000_60.csv
        restricted_acyclic_any_shortest_46_10_60.csv
        restricted_simple__46_100000_60.csv
        restricted_simple__46_10_60.csv
        restricted_simple_all_shortest_46_100000_60.csv
        restricted_simple_all_shortest_46_10_60.csv
        restricted_simple_any_46_100000_60.csv
        restricted_simple_any_46_10_60.csv
        restricted_simple_any_shortest_46_100000_60.csv
        restricted_simple_any_shortest_46_10_60.csv
        restricted_trail__46_100000_60.csv
        restricted_trail__46_10_60.csv
        restricted_trail_all_shortest_46_100000_60.csv
        restricted_trail_all_shortest_46_10_60.csv
        restricted_trail_any_46_100000_60.csv
        restricted_trail_any_46_10_60.csv
        restricted_trail_any_shortest_46_100000_60.csv
        restricted_trail_any_shortest_46_10_60.csv
    ▶ test_facebook_results/
      __init__.py
      facebook.csv
      test_diamond.py
      test_facebook.py
      test_main.py
      test_wikiRfA.py
      wikiRfA.csv
    __init__.py
  testing.sh*
```

```
    ▶ __pycache__/
    ▶ test_diamond_results/
    ▼ test_facebook_results/
        all_shortest_walk_0_100000_60.csv
        all_shortest_walk_0_10_60.csv
        all_shortest_walk_1123_100000_60.csv
        all_shortest_walk_1123_10_60.csv
        all_shortest_walk_1543_100000_60.csv
        all_shortest_walk_1543_10_60.csv
        all_shortest_walk_3754_100000_60.csv
        all_shortest_walk_3754_10_60.csv
        any_shortest_walk_0_False_100000_60.csv
        any_shortest_walk_0_False_10_60.csv
        any_shortest_walk_0_True_100000_60.csv
        any_shortest_walk_0_True_10_60.csv
        any_shortest_walk_1123_False_100000_60.csv
        any_shortest_walk_1123_False_10_60.csv
        any_shortest_walk_1123_True_100000_60.csv
        any_shortest_walk_1123_True_10_60.csv
        any_shortest_walk_1542_False_10_60.csv
        any_shortest_walk_1543_False_100000_60.csv
        any_shortest_walk_1543_True_100000_60.csv
        any_shortest_walk_1543_True_10_60.csv
        any_shortest_walk_3754_False_100000_60.csv
        any_shortest_walk_3754_False_10_60.csv
        any_shortest_walk_3754_True_100000_60.csv
        any_shortest_walk_3754_True_10_60.csv
        restricted_acyclic__100000_1.csv
        restricted_acyclic__10_60.csv
        restricted_acyclic_all_shortest_100000_1.csv
        restricted_acyclic_all_shortest_10_60.csv
        restricted_acyclic_any_100000_1.csv
        restricted_acyclic_any_10_60.csv
        restricted_acyclic_any_shortest_100000_1.csv
        restricted_acyclic_any_shortest_10_60.csv
        restricted_simple__100000_1.csv
        restricted_simple__10_60.csv
        restricted_simple_all_shortest_100000_1.csv
        restricted_simple_all_shortest_10_60.csv
        restricted_simple_any_100000_1.csv
        restricted_simple_any_10_60.csv
        restricted_simple_any_shortest_100000_1.csv
        restricted_simple_any_shortest_10_60.csv
        restricted_trail__100000_1.csv
        restricted_trail__10_60.csv
        restricted_trail_all_shortest_100000_1.csv
        restricted_trail_all_shortest_10_60.csv
        restricted_trail_any_100000_1.csv
        restricted_trail_any_10_60.csv
        restricted_trail_any_shortest_100000_1.csv
        restricted_trail_any_shortest_10_60.csv
```

Slika 4.1: Rezultati eksperimenata, spremni za obradu.

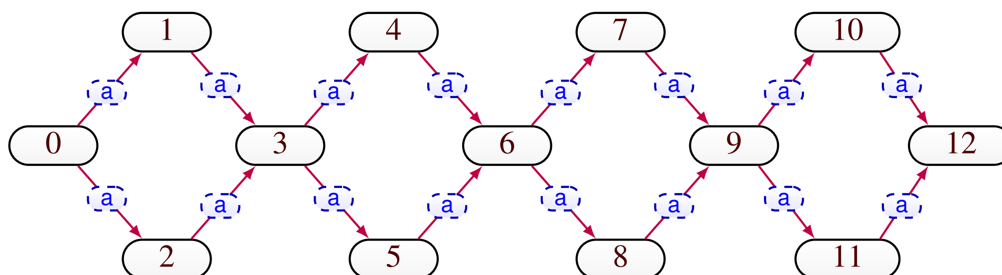
Isto tako, očito je

$$|\texttt{NEKI (NAJKRAĆI)? ŠETNJA}\,(0, \texttt{a}^*, ?x)| \in O(n).$$

Jedna varijanta testova koju ćemo izvršavati je na upravo opisanoj bazi izračunati upit s regularnim izrazom $\texttt{a}^*$.

Druga varijanta testova biti će na alternativnoj bazi koja se od one sa Slike 4.2 razlikuje samo u oznakama bridova koji ulaze u posljednji vrh. Njih ćemo zamijeniti iz `'a'` u `'b'`. Na njima ćemo izvršavati upit s regularnim izrazom $\texttt{a}^*b$ i selektori koji će biti na raspolaganju su `NEKI` i `NEKI NAJKRAĆI`. Dakako, rezultat izračunavanja takvih upita je 1 upit za broj čvorova > 1.

Uočimo da za svaki put u grafu vrijedi da nema ponavljanja ni vrhova ni bridova pa su svi putovi koji su `ŠETNJA`, ujedno i `STAZA`, `ACIKLIČAN` i `JEDNOSTAVAN`. Međutim, nemojmo se odmah zaletiti, to ne znači da je vrijeme izvršavanja isto! Kako smo opisali po rezultatima složenosti, jedino upiti za `restrictor = ŠETNJA` nemaju eksponencijalnu složenost. To ćemo vidjeti i po rezultatima eksperimenata.



Slika 4.2: Grafovska baza s eksponencijalno mnogo putova

Iako smo ključne značajke eksperimenata koje možemo očekivati već istaknuli, u nastavku prikazujemo potvrdu tih tvrdnji i u praksi. Analize `csv` tablica napravljene su u mapi `/rpq_evaluation/tests/visualization/` te pripadajući grafovi s rezultatima nalaze se u mapama `/rpq_evaluation/tests/visualization/diamond_visualized/` i u `/rpq_evaluation/tests/visualization/facebook_visualized/` pa ćemo neke zanimljive rezultate koje smo tamo vizualizirali i ovdje prikazati.

Ako je varijabla neprekidna (npr. vrijeme), rezultate prikazujemo s istaknutom prosječnom vrijednosti i standardnom devijacijom oko nje.

Testovi s `def test_any_walk(...)`

Na Slici 4.3 možemo vidjeti rezultate testiranja. Iznad svakog grafa se nalazi iz koje je on tablice potekao. Nakon imena `all_shortest_paths` slijede opisi varijabli koji su korišteni prilikom testiranja. Eksperiment provodimo tako da postupno povećavamo broj čvorova u grafu i pritom testiramo vrijeme izvršavanja i broj rješenja. Vrijeme izvršavanja preciznije testiramo postavljanjem malog `limita`, konkretno 10, a broj rješenja ispitujemo testovima u kojima je `limit`= 10000.

U svakom testu postavljene su 4 značajke:

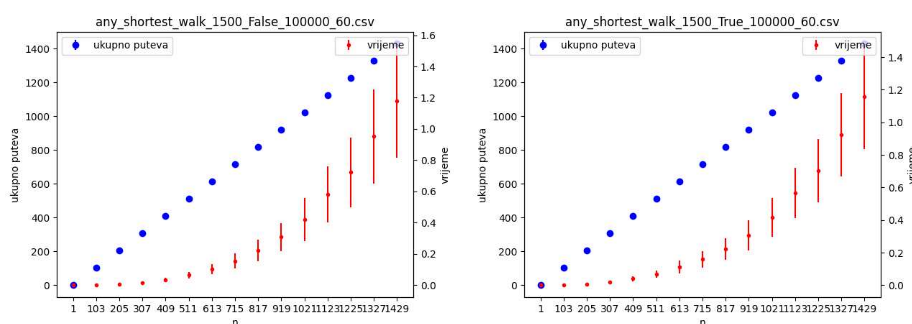
- broj `max_n` - najveći broj čvorova u grafu,
- logička vrijednost `shortest` - je li provodimo `BFS` ili `DFS` u algoritmu,
- najveći mogući broj rješenja `limit`,

- najduže moguće vrijeme izvršavanja `timeout`.

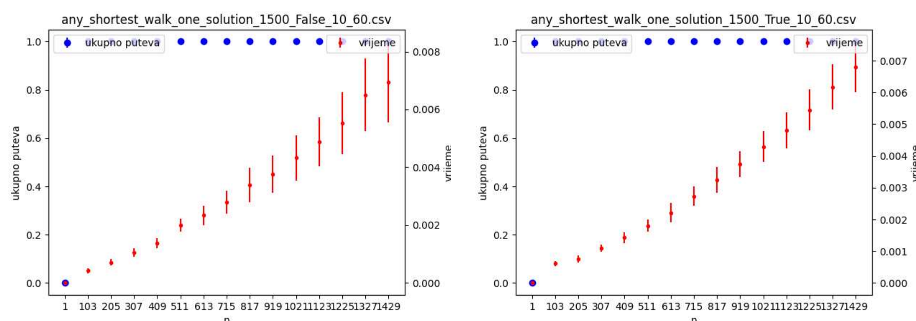
Dakle, imena grafova su shodno tome oblika
`any_shortest_walk_{max_n}_{shortest}_{limit}_{timeout}.csv` što predstavlja iz koje tablice su podaci. Napomenimo da u ovome grafu, zbog linearne složenosti, u svakom koraku broj čvorova u grafu n povećavamo za 102 te je ukupan broj putova u njemu jednak n.

S rezultata možemo vidjeti da je dohvaćanje malog broja rezultata tim sporije što je veći broj čvorova u grafu. S druge strane, broj puteva za `limit` od 10000 je linearan, kao što smo i očekivali.

Osim testova u kojem tražimo sve moguće putove po primjeru "dijamant", možemo tražiti i **samo jedan** najdulji put. To se postiže ako za regularni izraz postavimo `a`*`b`, a bridove koji ulaze u vrh s najvećim bridom označimo s `'b'` (sve ostale s `'a'`). Rezultate i u tom slučaju možemo vidjeti na Slici 4.4.



Slika 4.3: Rezultati za primjer "dijamant" za `def test_any_walk` s reg. izrazom a^*.

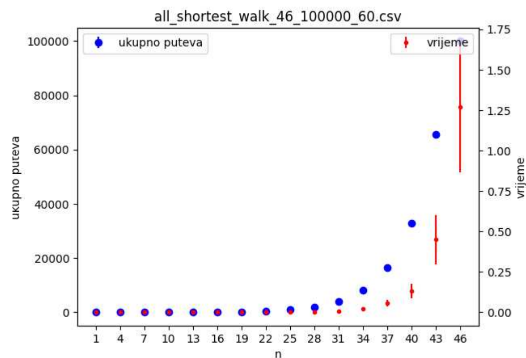


Slika 4.4: Rezultati za "dijamant" u kojem tražimo 1 put pomoću funkcije `test_any_walk` i reg. izraza a^*b na bazi s novim oznakama bridova koji ulaze u zadnji vrh.

Zaključak: za regularni izraz a^*, ukupno puteva je linearno. Vrijeme raste brže od linearnog, a što je veći broj vrhova, to više varira vrijeme izvođenja.

Testovi s `def all_shortest_walk(...)`

Slično kao u prethodnim testovima, imena grafova su oblika `all_shortest_walk_{max_n}_{limit}_{timeout}.csv`. Kao što smo objasnili gore, broj putova je eksponencijalan pa u ovom slučaju n u svakom koraku povećavamo za 3 te i na Slici 4.5 vidimo potvrdu u praksi teoretskog zaključka Za mali broj upita vrijeme izvođenja je otprilike identično dok je ukupan broj rješenja, potvrđeno i u praksi, eksponencijalan.



Slika 4.5: Rezultati testiranja primjera "dijamant" za `def test_all_shortest_walk`.

Zaključak: zorno se vidi da ukupan broj puteva raste eksponencijalna, a vrijeme povećavanjem broja čvorova sve više varira.

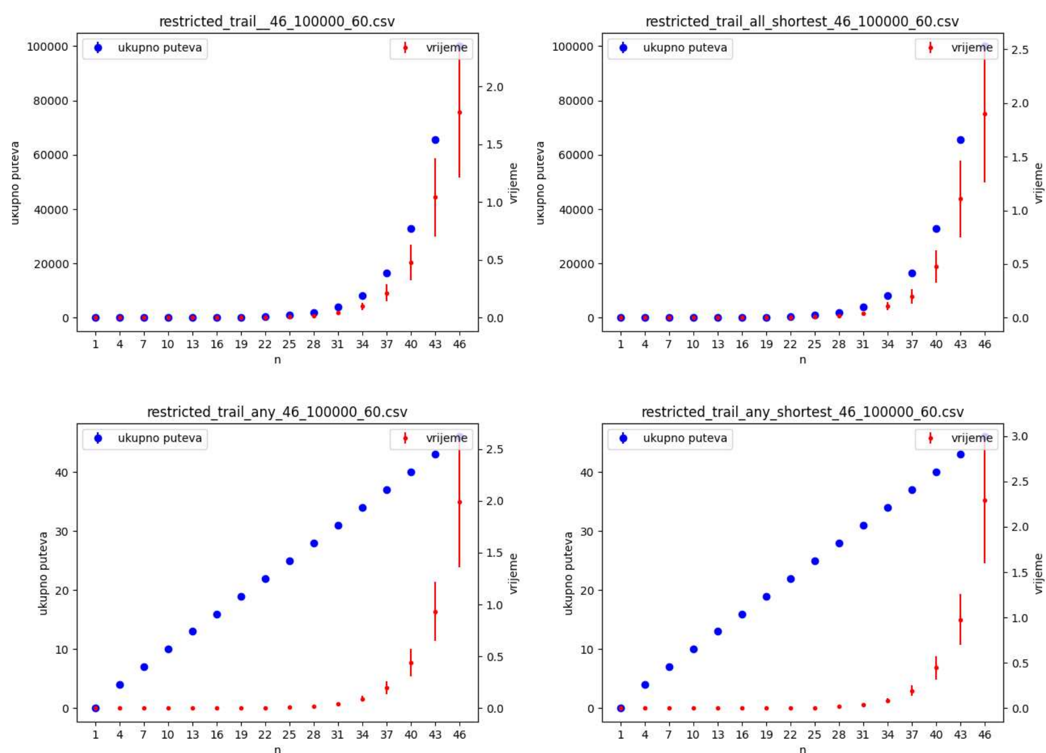
Testovi s `def restricted_walk(...)`

Kako smo već objasnili, `def restricted_walk` ispituje sve moguće puteve u grafu pa je eksponencijalne složenosti. U svim slučajevima smo n povećavali za 3 te radili s `limit`om od 10^5 i najvećim brojem čvorova 46. Rezultati izvođenja za svaki od `restrictor`a koji nisu `ŠETNJA` mogu se vidjeti na Slikama 4.6, 4.8 i 4.11. Osim toga, kao i u testovima za `def any_walk`, testovi za jedan put nalaze se na 4.7, 4.9 i 4.10.

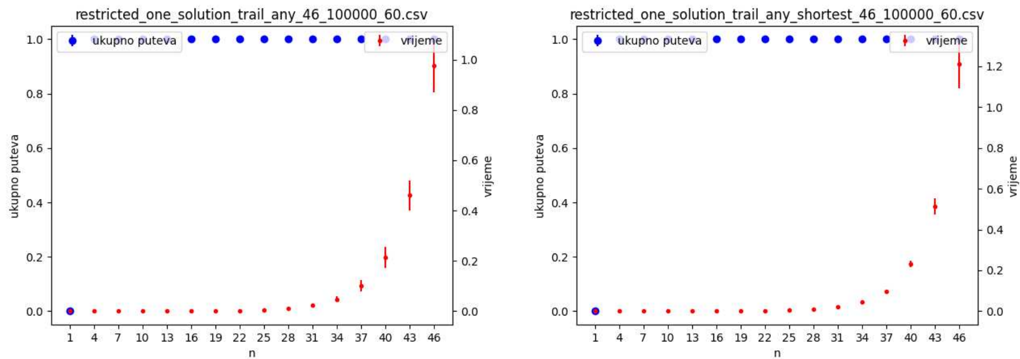
Vidimo da odabir `restrictor`a bitno ne mijenja izvođenje programa. Za male vrijednosti vrijeme izvršavanja ne mijenja se ovisno o broj čvorova u grafu. S druge strane, odabir `selector`a bitno mijenja vrijeme izvođenja. Oni `selector`i koji traže neki put, a takvi su `NEKI` i `NEKI NAJKRAĆI` izvršavaju se u linearnom vremenu. Drugi pak, koji traže

sve putove, a takav je SVI NAJKRAĆI ili slučaj kada testiramo bez selectora, izvršavaju se u eksponencijalnom vremenu.

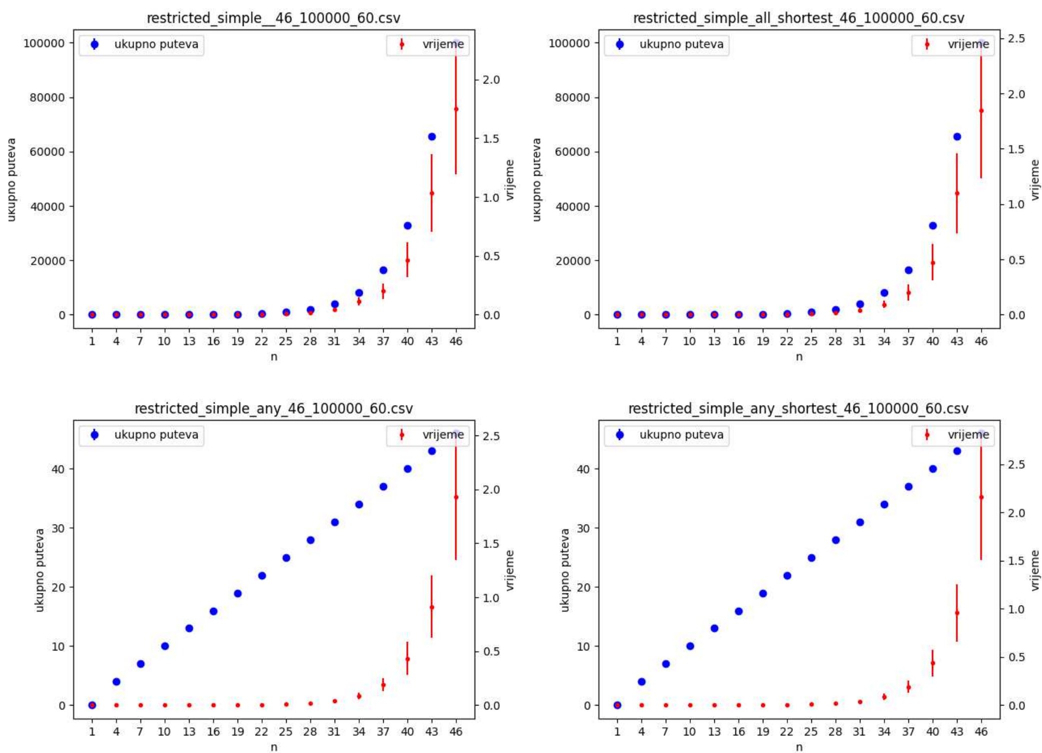
Imena grafova su u obliku test_diamond_results/restricted_{restrictor}_{selector}_{max_n}_{limit}_{timeout}.csv pri čemu su između ostalih već objašnjenih, restrictor i selector nizovi znakova (eng. string) koji definiraju koji restriktor i selektor se koristi.



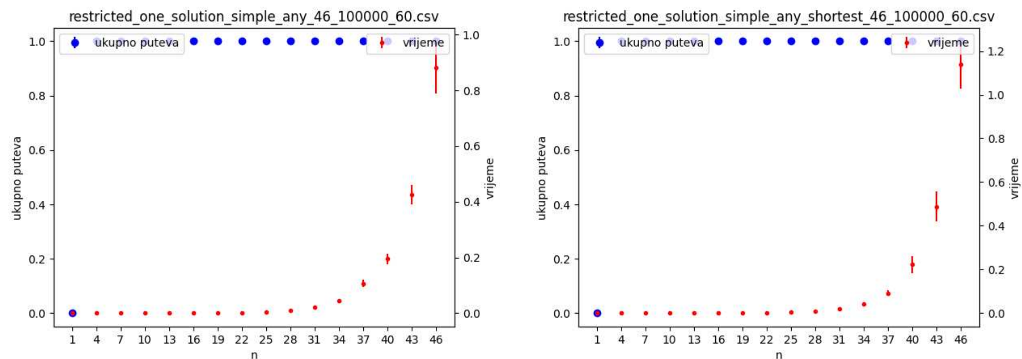
Slika 4.6: Rezultati testiranja primjera "dijamant" za def test_restricted pri čemu je restrictor = STAZA.



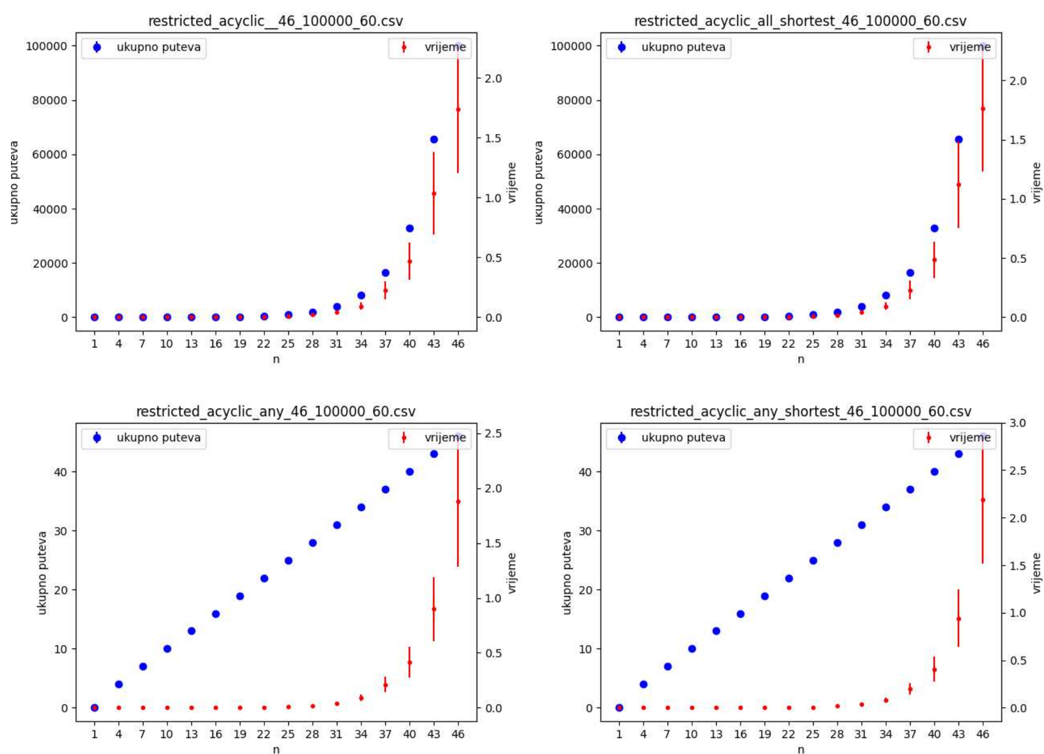
Slika 4.7: Rezultati testiranja primjera "dijamant" za `def test_restricted` pri čemu je `restrictor` = `STAZA` kada vraćamo jedan put.



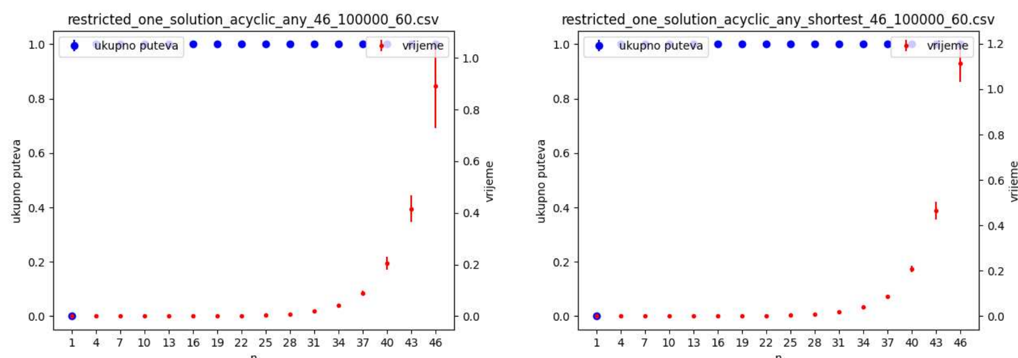
Slika 4.8: Rezultati testiranja primjera "dijamant" za `def test_restricted` pri čemu je `restrictor` = `JEDNOSTAVAN`.



Slika 4.9: Rezultati testiranja primjera "dijamant" za `def test_restricted` pri čemu je `restrictor` = `JEDNOSTAVAN` kada vraćamo jedan put.



Slika 4.10: Rezultati testiranja primjera "dijamant" za `def test_restricted` pri čemu je `restrictor` = `ACIKLIČAN`.



Slika 4.11: Rezultati testiranja primjera "dijamant" za `def test_restricted` pri čemu je `restrictor` = `ACIKLIČAN` kada vraćamo jedan put.

Zaključak: algoritmi imaju eksponencijalni broj ukupnih putova bez selektora i za selektor `all_shortest`. Za ostale restrictore ukupan broj putova je linearan. Traženje jednog rješenja dosta je sporije nego traženjem rješenja s prve dvije funkcije.

4.2 Primjer iz stvarnog svijeta

Primjer iz stvarnog svijeta pojednostavljeni je graf međusobnih prijateljstava određene skupine ljudi s društvene mreže `Facebook`. Za razliku od sintetičkog grafa, gdje smo unaprije znali što možemo očekivati od rezultata, inherentnu strukturu ovog grafa još ne poznajemo te će se svi zaključci donisiti *a posteriori*. Ono što znamo je da je ovaj graf neusmjeren i ima:

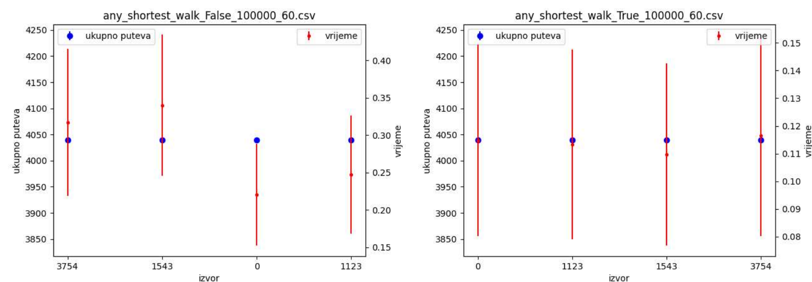
- 4039 vrhova,
- 88234 neusmjerenih bridova, koji su reprezentirani pomoću dvostruko toliko, 176468 usmjerenih bridova,
- sve bridove s istom oznakom `'a'`.

Kao i u primjeru dijamant, jedino je smisleno testirati s regularnim izrazom $\mathtt{a}^*$ jer su sve oznake iste.

Za razliku od prethodnih vizualizacija, u ovom grafu je broj bridova uvijek fiksan pa ne možemo ispitivati složenost, ali varijabilan ostaje početak puta. Tako ćemo testirati funkcije `test_any_walk` i `all_shortest_walk`. Posljednju funkciju testiramo samo za početak koji je vrh 0.

Testovi s `def test_any_walk(...)`

Testovi u ovom slučaju prikazani su na Slici 4.12. Možemo zaključiti da je graf potpuno povezan jer se iz svakog vrhova uspije doći do svakog drugog. Od svih rezultata možemo zaključiti da je sve čvorove najbrže posjetiti iz čvora 0, kada se pokreće `DFS`, inače je podjednako.

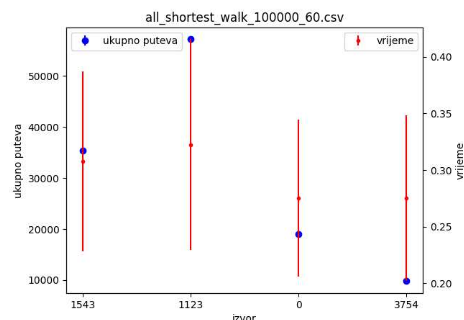


Slika 4.12: Rezultati testiranja primjera "Facebook" za `def test_any_walk`.

Zaključak: možemo vidjeti da je graf potpuno povezan i da iz svakog vrha možemo doći u sve ostale.

Testovi s `def all_shortest_walk(...)`

Rezultati testova u ovom slučaju prikazani su na Slici 4.13. Iako je graf potpuno povezan, to ne znači da mora biti jednako svih najkraćih putova iz različitih vrhova. Konkretno, vidimo da je najmanje rezultata iz vrha 3754, a najviše iz 1123. Jasno, otkrivanje manje puteva zahtijeva i manje vremena pa 0 i 3754 koji imaju najmanje rezultata, također zahtijevaju u prosjeku manje vremena.

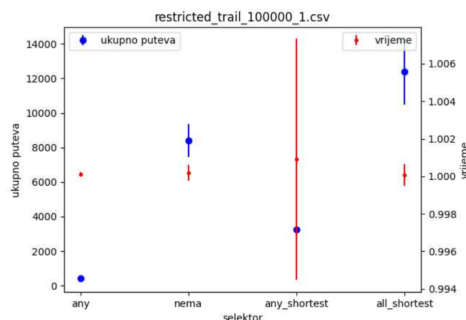


Slika 4.13: Rezultati testiranja primjera "Facebook" za `def test_all_shortest_walk`.

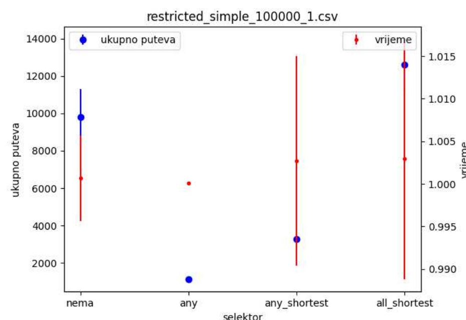
Zaključak: iako se do svakog vrha može doći, nije jednako najkraćih puteva u grafu od svako vrha.

Testovi s `def restricted_walk(...)`

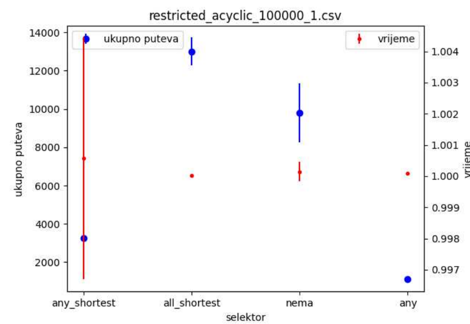
Rezultati testova u ovim slučajevima mogu se vidjeti na Slikama 4.14, 4.15 i 4.16. Ovo su možda i najzanimljiviji rezultati. Naime, zbog eksponencijalne složenosti koje funkcija `test_restricted` ima, prilikom eksperimentiranja uviđamo da vrijeme izvršavanja traje predugo, a većina putova se otkriju tijekom prve sekunde. Stoga, je za `limit = 100000` stavljeno `timeout = 1`. Stoga, ovo su prvi rezultati u kojima ukupan broj puteva nije fiksan rezultat.



Slika 4.14: Rezultati testiranja primjera "Facebook" za `def test_restricted` i `restrictor` = `STAZA`.



Slika 4.15: Rezultati testiranja primjera "Facebook" za `def test_restricted` i `restrictor` = `JEDNOSTAVAN`.



Slika 4.16: Rezultati testiranja primjera "Facebook" za `def test_restricted` i `restrictor` = `STAZA`.

Zaključak: većina vrijednosti nađe se relativno brzo, ali treba jako dugo čekati da se ovaj algoritam izvede.

Bibliografija

[1] Angles, Renzo, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter i Domagoj Vrgoc: *Foundations of Modern Query Languages for Graph Databases*, 2017.

[2] Barceló Baeza, Pablo: *Querying Graph Databases*. U *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, stranica 175–188, New York, NY, USA, 2013. Association for Computing Machinery, ISBN 9781450320665. `https://doi.org/10.1145/2463664.2465216`.

[3] Farías, Benjamín, Carlos Rojas i Domagoj Vrgoč: *Evaluating Regular Path Queries in GQL and SQL/PGQ: How Far Can The Classical Algorithms Take Us?*, 2023.

[4] Fortune, Steven, John Hopcroft i James Wyllie: *The directed subgraph homeomorphism problem*. Theoretical Computer Science, 10(2):111–121, 1980, ISSN 0304-3975. `https://www.sciencedirect.com/science/article/pii/0304397580900092`.

[5] Krekel, Holger: *pytest Documentation Release 8.0*. `https://docs.pytest.org/_/downloads/en/latest/pdf/`, 2023. Pristupljeno: 2023-07-01.

[6] Lapaugh, Andrea S. i Christos H. Papadimitriou: *The even-path problem for graphs and digraphs*. Networks, 14(4):507–513, 1984. `https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230140403`.

[7] Lapaugh, Andrea S. i Ronald L. Rivest: *The subgraph homeomorphism problem*. Journal of Computer and System Sciences, 20(2):133–149, 1980, ISSN 0022-0000. `https://www.sciencedirect.com/science/article/pii/0022000080900574`.

[8] Reis, Rogério i Nelma Moreira: *FAdo Documentation Release 2.1.2*. `https://www.dcc.fc.up.pt/~rvr/FAdo.pdf`, 2023. Pristupljeno: 2023-07-01.

[9] Vrgoč, Domagoj: *Querying graphs with data*. Disertacija, University of Edinburgh, UK, 2014. `https://hdl.handle.net/1842/8953`.

[10] Vrgoč, Domagoj: *Evaluating regular path queries under the all-shortest paths semantics*, 2023.

[11] Vuković, Mladen: *Složenost algoritama*. https://www.pmf.unizg.hr/_download/repository/SA-skripta-2019.pdf, 2019. Pristupljeno: 2023-07-01.

[12] Čačić, Vedran i Marko Horvat: *Interpretacija programa — predavanja*. https://web.math.pmf.unizg.hr/~veky/ip/IP%20slajdovi.pdf, 2019. Pristupljeno: 2023-07-01.

Sažetak

U ovom diplomskom radu istražuje se primjena grafovskih baza podataka u kontekstu regularnih upita na putove. Prvo su dane osnovne definicije vezane uz grafovske baze podataka, kao i pregled različitih modela tih baza te potom simuliranje između različitih modela baza

Nadalje, rad se fokusira na algoritme za izračunavanje regularnih upita na putove i pruža pregled u glavne važne vrste putova (šetnja, staza, jednostavan i acikličan), kao i slučajeve kada su odabrani svi najkraći, neki ili neki najkraći putevi. Izvedene su složenosti algoritama te dokazan rezultat o `NP`-potpunosti jezika `NEKI JEDNOSTAVNI PUT`.

Algoritmi su implementirani u programskom jeziku Python i ona je javno dostupna na https://github.com/mdujic/diplomski. U sklopu implementacije primjenjen je Glushkovovov algoritma i opisan postupak izgradnje nedeterminističkog konačnog automata s jedinstvenim završnim stanjem. Osim toga, prikazana je i implementacija grafovske baze poodataka sa strukturom podataka Trie u svojoj jezgri.

U sklopu eksperimenata, izvršeni su testovi na umjetnim primjerima s eksponencijalnim brojem slučajeva, kao i na primjerima iz stvarnog svijeta. Ovi testovi služe za provjeru efektivnosti i skalabilnosti razvijenih algoritama.

U zaključku, ovaj rad pruža pregled primjene grafovskih baza podataka i algoritama za izračunavanje regularnih upita na putove. Istraženi su različiti modeli baza, implementacija, kao i izvedeni eksperimenti koji su pokazali uspješnost razvijenih algoritama. Ovaj rad doprinosi razumijevanju i razvoju grafovskih baza podataka u kontekstu regularnih upita na putove.

Summary

This thesis explores the application of graph databases in the context of regular path queries. First, the basic definitions related to graph databases are given, as well as an overview of different models of these databases, followed by simulation between different database models.

Furthermore, the thesis focuses on algorithms for calculating regular path queries and provides an overview of the main important types of paths (walk, trail, simple, and acyclic), as well as cases when all shortest, any, or any shortest paths are selected. The complexities of the algorithms are derived, and a result of the NP-completeness of the language ANY SIMPLE PATH is proven. The algorithms were implemented in the Python programming language and the code is publicly available at https://github.com/mdujic/diplomski. As part of the implementation, Glushkov's algorithm was applied, and the process of constructing a nondeterministic finite automaton with a unique final state was described. Additionally, an implementation of a graph database using a Trie data structure is presented. In the experiments, tests were performed on artificial examples with an exponential number of cases, as well as on real-world examples. These tests serve to verify the effectiveness and scalability of the developed algorithms.

In conclusion, this thesis provides an overview of the application of graph databases and algorithms for calculating regular path queries. Different database models, implementations, and conducted experiments that demonstrated the success of the developed algorithms were explored. This work contributes to the understanding and development of graph databases in the context of regular path queries.

Životopis

Rođen sam 2. 12. 1999. u Kninu. Završio sam Osnovnu školu Domovinske zahvalnosti u Kninu i III. gimnaziju - Split. Tijekom osnovne i srednje škole sudjelojem na natjecanjima iz matematike, fizike, kemije i geografije na državnoj razini te dalje upisujem preddiplomski studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu. Tijekom cijelog studija aktivno se bavim volontiranjem u Udruzi Mladi nadareni matematičari "Marin Getaldić" gdje u jednom mandatu izvršavam i službu tajnika udruge. U akademskoj godini 2021./2022., zajedno s Laurom Horvat, dobivam rektorovu nagradu za individualni znanstveni rad, a u 2022./2023. dobivam Nagradu za najuspješnije studente završnih godina svih preddiplomskih, diplomskih i integriranog studija za uspjehe na Diplomskom studiju Računarstvo i Matematika. Tijekom studija bio sam u 2 navrata demonstrator na kolegiju Računarski praktikum 1 te sam više godina demonstrator za snimanje predavanja i čuvanje u praktikumima na Matematičkom odsjeku. Tijekom fakultetskog obrazovanja skupljao sam radno iskustvo na ljetnim praksama u tvrtkama Visage Technologies i Memgraph te trenutno radim u tvrtki CIAL Dun & Bradstreet.