Univerza na Primorskem

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Računalništvo in informatika, 2. stopnja

Marko Grgurovič

Najkrajše poti v krajšem času

Magistrsko delo

Koper, 2012

Mentor: prof. dr. Andrej Brodnik

Ključna dokumentacijska informacija

Ime in PRIIMEK: Marko GRGUROVIČ

Naslov magistrskega dela: Najkrajše poti v krajšem času

Kraj: Koper

Leto: 2012

Število listov: 49 Število slik: 6 Število tabel: 0
Število prilog: 0 Število strani prilog: 0
Število referenc: 28

Mentor: prof. dr. Andrej Brodnik

UDK: 004.02(043.2)

Ključne besede: najkrajše poti, teorija grafov, algoritmi, asimptotična analiza, kombinatorična optimizacija

Izvleček: Magistrsko delo obravnava problem najkrajših poti v grafih. Predstavljen je nov algoritem, ki rešuje problem najkrajših poti med vsemi pari vozlišč. Omenjeni algoritem boljše izrablja poljuben algoritem za reševanje problema najkrajših poti iz enega izvora. Dobljene asimptotične zahtevnosti so bistveno boljše od vseh trenutno znanih pristopov. Podamo tudi povezavo med težavnostjo problema najkrajših poti iz enega izvora in problemom urejenih najkrajših poti med vsemi pari vozlišč. Slednja povezava nudi bodisi olajšavo za nadaljne delo pri iskanju hitrejših algoritmov za problem najkrajših poti med vsemi pari vozlišč, bodisi način, kako lahko dokažemo spodnje meje za problem najkrajših poti iz enega izvora. Nenazadnje definiramo problem najkrajših poti med nekaterimi pari vozlišč in podamo algoritem, ki izvede manj osnovnih operacij kot algoritem Floyd-Warshall.

Key words documentation

Name and SURNAME: Marko GRGUROVIČ

Topic of Master's thesis: Shortest paths in shorter time

Place: Koper

Year: 2012

Number of pages: 49 Number of figures: 6 Number of tables: 0
Number of appendix: 0 Number of appendix pages: 0
Number of references: 28

Advisor: prof. dr. Andrej Brodnik

UDC: 004.02(043.2)

Keywords: shortest paths, graph theory, algorithms, asymptotic analysis, combinatorial optimization

Abstract: We study the shortest path problem in graphs. A new approach to solving the all-pairs shortest path problem is presented. The resulting algorithm makes better use of an arbitrary algorithm solving the single-source shortest path problem. This presents a significant improvement over the currently known approaches. We also identify a connection between the complexity of the single-source shortest path problem and a sorted variant of the all-pairs shortest path problem. This either suggests a simplification for future research on all-pairs shortest path algorithms or a way to prove a lower bound on the single-source shortest path problem. Last but not least, we define the some-pairs shortest path problem and give an algorithm that performs fewer basic operations than Floyd-Warshall.

Kazalo vsebine

Kazalo slik

Kazalo algoritmov

Seznam kratic

Kratica	**Pomen (angl.)**	**Pomen (sl.)**
APSP	*All-pairs shortest path*	Najkrajše poti med vsemi pari vozlišč
BFS	*Breadth-first search*	Iskanje v širino
DAG	*Directed acyclic graph*	Usmerjen acikličen graf
DFS	*Depth-first search*	Iskanje v globino
FIFO	*First in, first out*	Prvi noter, prvi ven
LIFO	*Last in, first out*	Zadnji noter, prvi ven
RAM	*Random access machine*	Stroj z naključnim dostopom
SAPSP	*Sorted all-pairs shortest path*	Urejene najkrajše poti med vsemi pari vozlišč
SCC	*Strongly connected component*	Krepko povezana komponenta
SSSP	*Single source shortest path*	Najkrajše poti iz enega vozlišča do vseh ostalih vozlišč

Slovarček

Slovarček vsebuje nekatere pojme, ki se pojavljajo v nalogi. Določene razlage so povzete po Islovarju[1].

Algoritem (angl. *algorithm*)	zaporedje pravil, operacij, ukazov, ki zagotavljajo rešitev problema v končnem številu korakov.
Dinamično programiranje (angl. *dynamic programming*)	postopek reševanja problemov, kjer učinkovito rešimo ponavljajoče se podprobleme.
Stroj z naključnim dostopom (angl. *random access machine*)	teoretični model računalnika, kjer lahko dostopamo do poljubnega naslova v pomnilniku v konstantem času.
Sklad (angl. *stack*)	linearni seznam, pri katerem je mogoče elemente dodajati ali odvzemati po načelu LIFO le na enem koncu seznama.
Vrsta (angl. *queue*)	linearni seznam, pri katerem se elementi po načelu FIFO dodajajo na enem koncu seznama, odvzemajo pa na nasprotnem.

[1] http://www.islovar.org/

Zahvala

Zahvalil bi se rad svojemu mentorju, prof. dr. Andreju Brodniku, ki je bil vedno pripravljen prisluhniti marsikaterim (smiselnim in tudi manj smiselnim) idejam tekom zadnjih dveh let. Rad bi se zahvalil tudi svoji mami, ki si je vzela čas in mi pomagala pri odpravi slovničnih napak prisotnih v tem delu.

1 UVOD

V magistrskem delu se ukvarjamo s problemom najkrajših poti v grafih. Gre za enega izmed najstarejših osnovnih problemov, s katerim se področje teoretičnega računalništva ukvarja že več kot pol stoletja. Problem ima razne aplikacije na probleme kot so: pošiljanje paketa po omrežju tako, da bo pot čim krajša, iskanje najkrajše poti v cestnem omrežju, iskanje razdalje med ljudmi v socialnih omrežjih, itd. Obravnavamo problema iskanja najkrajših poti iz enega izvora in iskanja najkrajših poti med vsemi pari vozlišč. Sprva predstavimo nekaj osnovnih pojmov o teoriji grafov in njihovi računalniški predstavitvi. Ogledamo si klasične pristope za reševanje problemov najkrajših poti in tudi nove prijeme, ki so se uveljavili v zadnjih letih. V nadaljevanju nato podamo nove algoritme, formalno dokažemo njihovo pravilnost in asimptotično analiziramo njihovo prostorsko in časovno zahtevnost.

Bistveni prispevki magistrskega dela k znanosti, razvrščeni po pomembnosti od najpomembnejših do manj pomembnih, so:

- Nov algoritem za reševanje problema najkrajših poti med vsemi pari vozlišč, ki boljše izrablja poljuben algoritem za reševanje problema najkrajših poti iz enega izvora. Dobljene asimptotične zahtevnosti so bistveno boljše od vseh trenutno znanih pristopov.

- Povezava med zahtevnostjo problema najkrajših poti iz enega izvora in problemom urejanja najkrajših poti med vsemi pari vozlišč. Ta povezava nudi bodisi olajšavo za nadaljne delo pri iskanju hitrejših algoritmov za problem najkrajših poti med vsemi pari vozlišč, bodisi način, kako lahko dokažemo spodnje meje za problem najkrajših poti iz enega izvora.

- Nov algoritem za reševanje problema sorodnega najkrajšim potem med vsemi pari vozlišč, kjer nas zanimajo samo poti med nekaterimi pari. Dobljeni algoritem izvede manj osnovnih operacij kot algoritem Floyd-Warshall.

2 OSNOVNI POJMI

V tem poglavju bomo definirali nekaj osnovnih pojmov, ki jih bomo potrebovali v nadaljevanju.

2.1 GRAF

Najprej definirajmo graf $G = (V, E)$, kjer množica V predstavlja množico vozlišč (angl. *vertices*) grafa G in množica E predstavlja množico povezav (angl. *edges*) grafa G. Opravka bomo imeli z digrafi, tj. grafi, kjer je vsaka povezava $(u, v) \in E$ usmerjena. Usmerjenost bomo izrazili z urejenim parom (u, v), kar pomeni, da je to povezava iz vozlišča u v vozlišče v. Digrafi, s katerimi se bomo ukvarjali, bodo uteženi, kar pomeni, da bo imela vsaka povezava $(u, v) \in E$ določeno težo, ki jo bomo izrazili kot funkcijo $\ell(u, v) : E \to \mathbb{R}$. V primeru, da povezava med dvema vozliščema u, v ne obstaja, bomo zahtevali, da funkcija $\ell(u, v)$ vrne vrednost ∞.

Zaradi enostavnejšega zapisa bomo definirali $n = |V|$ in $m = |E|$. Opravka bomo imeli z enostavnimi grafi, za katere bomo v nadaljevanju uporabljali besedo graf. V enostavnih grafih[1] ima vsak par vozlišč $v_1, v_2 \in V$ največ eno povezavo v usmerjenosti (v_1, v_2) ter največ eno povezavo v usmerjenosti (v_2, v_1). V enostavnih grafih lahko omejimo število povezav z $m \leq n^2 - n$, vendar pogosto želimo tudi prisotnost zank, zato bomo pisali $m \leq n^2$.

2.1.1 Stopnja vozlišča

V digrafih predstavlja vhodna stopnja (angl. *indegree*) nekega vozlišča $v \in V$ število vhodnih povezav v to vozlišče. Podobno tudi izhodna stopnja (angl. *outdegree*) nekega vozlišča $v \in V$ predstavlja število izhodnih povezav iz tega vozlišča. Pisali bomo $deg^+(v)$ za vhodno stopnjo vozlišča v in $deg^-(v)$ za izhodnjo stopnjo.

2.2 POTI V GRAFIH

V grafu $G = (V, E)$ definiramo pot π_{st} od nekega vozlišča $s \in V$ do drugega vozlišča $t \in V$ kot zaporedje prehojenih povezav $\pi_{st} = \{(s, u), (u, v), ..., (w, t)\}$. Dolžino poti označimo z $\ell(\pi_{st}) = \sum_{(u,v)\in\pi_{st}} \ell(u, v)$. Definirali bomo naslednje pojme:

- $D(v)$ predstavlja trenutno zgornjo oceno dolžine najkrajše poti od izvora do vozlišča $v \in V$. V kolikor želimo izvor izrecno podati (če to ni samoumevno), bomo pisali $D(s, v)$, pri čemer je $s \in V$ izvor.

[1] Za razliko od multigrafov, kjer je lahko povezav med vozlišči poljubno mnogo.

- $\delta(v)$ predstavlja trenutno spodnjo oceno dolžine poti od izvora do vozlišča v. Če želimo izvor izrecno podati, bomo pisali $\delta(s, v)$.

- $d(v)$ predstavlja dolžino najkrajše poti od izvora do vozlišča v. Če želimo izvor izrecno podati, bomo pisali $d(s, v)$. Naj bo Π_{st} množica vseh poti med vozliščima s in t. Dolžino najkrajše poti definiramo kot $d(s, v) = \min_{\pi_{sv} \in \Pi_{sv}} \ell(\pi_{sv})$, če obstaja pot iz vozlišča s do vozlišča v v grafu G in $d(s, v) = \infty$, če pot ne obstaja.

Pri tem bomo zahtevali, da velja naslednje:

$$\forall s, v \in V : \delta(s, v) \leq d(s, v) \leq D(s, v) \tag{2.1}$$

Zgornje in spodnje ocene dolžin poti ter njihovo uporabnost bomo natančneje opredelili kasneje. Definirali bomo tudi pojem premer (angl. *diameter*) grafa $DIAM(G)$, ki predstavlja dolžino najdaljše izmed vseh najkrajših poti v grafu G. Formalno zapišemo kot:

$$DIAM(G) = \max_{u,v \in V} d(u, v)$$

2.2.1 Sproščanje

Sedaj si bomo ogledali pojem zgornjih ocen dolžin najkrajših poti podrobneje. Zgornje ocene bi lahko poimenovali tudi trenutni približki najkrajših poti. Sproščanje je postopek, ki izboljšuje te približke, tj. jih zniža. Enostaven primer sproščanja je naslednji: imamo zgornji oceni do vozlišč u in v zapisani kot $D(u)$ in $D(v)$. Ker v grafu obstaja povezava (u, v), lahko poizkusimo znižati zgornjo oceno do vozlišča u tako, da $D(v) = min(D(v), D(u) + \ell(u, v))$. Razlog za uporabo funkcije min je enostavno ta, da ne želimo še dodatno povišati zgornje ocene $D(v)$. Sproščanje lahko torej kvečjemu zniža zgornjo oceno.

Vsi algoritmi, ki si jih bomo ogledali v naslednjih poglavjih, uporabljajo sproščanje. Če je sproščanje edini način spreminjanja zgornjih ocen razdalj v nekem algoritmu, potem velja lema 2.1.

Lema 2.1. (Sproščanje) $\forall v \in V : D(v) \geq d(v)$ *in kadar* $D(v) = d(v)$*, se od takrat dalje* $D(v)$ *ne spremeni.*

Dokaz. Če bi veljalo $D(v) < d(v)$, bi to pomenilo, da obstaja pot, katere dolžina je $D(v)$ in je krajša od najkrajše poti v grafu, kar je protislovje. Zaradi definicije sproščanja s funkcijo min se zgornja ocena ne more povečati. Torej, če $D(v) = d(v)$ in bi se vrednost $D(v)$ v nadaljevanju spremenila, bi se morala zgornja meja dodatno znižati. To bi pomenilo $D(v) < d(v)$, kar je ponovno v protislovju z definicijo najkrajših poti. □

2.2.2 Cikli

Cikel je tako zaporedje povezav, ki se začne in konča v istem vozlišču $v \in V$. Grafom, ki nimajo ciklov in so obenem usmerjeni, pravimo usmerjeni aciklični grafi (angl. *directed acyclic graph, DAG*).

V grafih, kjer uteži povezav zavzamejo tudi negativne vrednosti, lahko pride do ciklov negativnih dolžin. V takih grafih pojem najkrajših poti ni dobro definiran.

Razlog za to je, da lahko v neko pot vključimo cikel negativne dolžine in ga ponavljamo, kar sproti zmanjšuje dolžino poti. V primeru, da je cikel dosegljiv iz vsakega vozlišča, so dolžine vseh poti poljubno majhne in jih pogosto zapišemo kot $\forall v \in V : d(v) = -\infty$. Algoritmi, ki iščejo najkrajše poti v grafih, kjer so uteži lahko tudi negativne, imajo pogosto mehanizem, ki take cikle najde in jih izpiše. Zaradi težavnosti definicije pojma najkrajših poti bomo v teh primerih imeli v nadaljevanju opravka z grafi, ki ne vsebujejo negativnih ciklov.

2.2.3 Optimalnost podpoti

Naj bo π neka poljubna najkrajša pot, ki jo lahko zapišemo kot $\pi = \{v_1, v_2, ..., v_k\}$. Potem za dve poljubni števili i, j, tako, da velja $1 \leq i \leq j \leq k$, pravimo, da je $\pi_{ij} = \{v_i, v_{i+1}, ..., v_j\}$ podpot poti π.

Lema 2.2. (Optimalnost podpoti) *Vsaka podpot π_{ij} najkrajše poti π je najkrajša pot od vozlišča v_i do vozlišča v_j.*

Dokaz. Vzemimo najkrajšo pot π. Sedaj jo lahko razdelimo na tri dele za poljubni števili i, j, kot smo jih definirali zgoraj. Potem velja $\ell(\pi) = \ell(\pi_{1i}) + \ell(\pi_{ij}) + \ell(\pi_{jk})$ po definiciji dolžine poti. V kolikor π_{ij} ni najkrajša pot od vozlišča i do vozlišča j, potem obstaja neka pot π'_{ij}, za katero velja $\ell(\pi'_{ij}) < \ell(\pi_{ij})$. To pomeni, da lahko v poti π zamenjamo podpot π_{ij} s potjo π'_{ij} in po definiciji razdalj poti dobimo krajšo pot od najkrajše poti, kar je protislovje. □

2.2.4 Trikotniška neenakost

Ključnega pomena v nekaterih algoritmih bo pojem trikotniške neenakosti.

Lema 2.3. (Trikotniška neenakost) $\forall s, t, v \in V : d(s, v) + \ell(v, t) \geq d(s, t)$

Dokaz. Predpostavimo, da $d(s, v)+\ell(v, t) < d(s, t)$. Spomnimo, da smo $d(s, t)$ definirali kot *najkrajšo* pot med vozliščema s in t. Vendar bi jo v tem primeru lahko zamenjali s potjo $d(s, v) + \ell(v, t)$, ki je *krajša* od $d(s, t)$, kar je v protislovju z definicijo, saj je $d(s, t)$ najkrajša pot. □

Če v trikotniški neenakosti zamenjamo $d(\cdot)$ z $\delta(\cdot)$, dobimo trikotniško neenakost spodnjih ocen. Trikotniška neenakost spodnjih ocen je uporabna, če želimo povečevati spodnje ocene. Pri slednjem moramo tudi paziti, da ohranjamo zahtevo spodnjih ocen, tj. $\delta(v) \leq d(v)$. V ta namen lahko združimo pojem trikotniške neenakosti in sproščanja tako, da sprostimo vse povezave (u, v) kadar $\delta(v) = D(v)$ in od tu naprej ne povečujemo več $\delta(v)$. Tak način povečevanja spodnjih ocen stoji v ozadju trenutno asimptotično najhitrejših algoritmov za iskanje najkrajših poti v grafih, katere si bomo bolj podrobno ogledali v kasnejših poglavjih.

Lema 2.4. (Povečevanje spodnjih ocen) *Sprva naj bodo vse spodnje ocene $\forall v \in V : \delta(v) = 0$. Naj obstaja nek algoritem, ki povečuje spodnje ocene upoštevajoč $\forall t \in V : \delta(v) + \ell(v, t) \geq \delta(t)$ in $\forall v \in V : \delta(v) \leq D(v)$ ter sprosti vsako povezavo $\forall u : (v, u) \in E$ kadar $\delta(v) = D(v)$. Tak algoritem ob sprostitvi vseh povezav izračuna najkrajše poti od nekega izvora $s \in V$ do vseh ostalih vozlišč $v \in V$ katerih dolžine so $D(v)$.*

Dokaz. Predpostavimo, da za vse $v \in V$ in nek $t \in V$ velja: $\delta(v) + \ell(v,t) \geq \delta(t)$ in $\delta(t) = D(t)$. V tem primeru velja $D(t) = d(t)$, saj so vse druge poti, ki vodijo v vozlišče t daljše. V kolikor to ne velja, potem obstaja nek $v \in V$ za katerega velja $\delta(v) + \ell(v,t) < \delta(t)$, vendar tako povečevanje spodnjih ocen ne upošteva zahtevane neenakosti, kar je v protislovju z definicijo. □

2.2.5 Problem iskanja najkrajših poti

Problem, s katerim se v magistrskem delu ukvarjamo, je iskanje najkrajših poti. Obravnavali bomo dve formulaciji tega problema. Prva je problem iskanja najkrajših poti iz enega izvora (angl. *Single source shortest path, SSSP*), kjer iščemo vrednosti $d(s,v)$ (in pogosto tudi poti) za vse $v \in V$, kjer je vozlišče $s \in V$ (izvor) izrecno podano v problemu. Druga formulacija je problem iskanja najkrajših poti med vsemi pari vozlišč (angl. *All-pairs shortest path, APSP*), kjer nas zanimajo vrednosti $d(u,v)$ za vse $u, v \in V$. Problema APSP in SSSP sta tesno povezana. V kolikor imamo algoritem, ki rešuje problem SSSP, ga lahko uporabimo za reševanje problema APSP. To storimo tako, da izvedemo n poizvedb tipa SSSP, kjer spreminjamo izvorno vozlišče s. V poglavju 3 si bomo ogledali vrsto algoritmov, ki učinkovito rešujejo ta dva problema.

2.3 POVEZANOST GRAFA

Za nek *neusmerjen* graf pravimo, da je povezan (angl. *connected*), če v njem obstaja pot med poljubnima vozliščima $u, v \in V$. Ker imamo opravka predvsem z digrafi, lahko definiramo dva sorodna pojma. Naj bo graf G' neusmerjena različica grafa G, ki jo dobimo tako, da vsako usmerjeno povezavo naredimo neusmerjeno. Digraf G je potem:

- šibko povezan (angl. *weakly connected*), če obstaja pot med poljubnima vozliščima $u, v \in V$ v grafu G'
- krepko povezan (angl. *strongly connected*), če obstaja pot med poljubnima vozliščima $u, v \in V$ v grafu G.

2.3.1 Krepko povezane komponente grafa

Kljub temu, da nek graf G ni nujno krepko povezan, lahko najdemo nek podgraf $\bar{G}$ grafa G, ki je krepko povezan. Enostavno je videti, da ima tudi krepko povezani podgraf dodatne podgrafe, ki so sami tudi krepko povezani. Iz tega razloga se bomo osredotočili na krepko povezane podgrafe, ki vsebujejo največje možno število vozlišč. Takim podgrafom pravimo krepko povezane komponente (angl. *strongly connected component, SCC*) grafa G.

Ena izmed uporabnih lastnosti krepko povezanih komponent je ta, da po skrčitvi vseh krepko povezanih komponent grafa G ostane acikličen usmerjen graf G'. Skrčitev neke krepko povezane komponente C definiramo kot odstranitev vseh vozlišč $v \in C$, vsebovanih v krepko povezani komponenti iz grafa G in dodatek novega vozlišča c, ki prevzame povezave odstranjenih vozlišč. V kolikor bi po tovrstni skrčitvi obstajala krepko povezana komponenta grafa G', bi morala le-ta tvoriti večjo krepko povezano komponento v izvornem grafu G, kar pa je v protislovju z definicijo krepko povezane komponente.

2.4 DREVESA

Praviloma se v teoriji grafov drevesa nanašajo na neusmerjene grafe. Tukaj in v nadaljevanju, razen v primeru, da izrecno zahtevamo drugače, obravnavamo drevesa kot usmerjene grafe. Za nek graf $G = (V, E)$ pravimo, da je drevo (angl. *tree*), če ob odstranitvi poljubne povezave $e \in E$ iz grafa G dobimo graf, ki ni šibko povezan. Vsako drevo je usmerjen acikličen graf, ni pa vsak usmerjen acikličen graf drevo. Definirali bomo naslednje pojme, ki se nanašajo na drevesa:

- Vozlišču, ki nima izhodnih povezav, pravimo list (angl. *leaf*).
- Vozlišču, ki nima vhodnih povezav, pravimo koren (angl. *root*).
- Če ima vozlišče v izhodno povezavo do vozlišča u, pravimo, da je v starš (angl. *parent*) vozlišča u in ekvivalentno, da je u otrok (angl. *child*) vozlišča v.
- Vozlišče v je prednik (angl. *ancestor*) vozlišča u, če obstaja pot od vozlišča v do vozlišča u. V tem primeru pravimo tudi, da je vozlišče u naslednik (angl. *successor*) vozlišča v.
- Vozlišče k je skupni prednik (angl. *common ancestor*) vozlišč u in v, če je prednik tako vozlišča u kot tudi vozlišča v.

3 ALGORITMI NA GRAFIH

3.1 PREDSTAVITEV GRAFOV

Kadar imamo opravka z algoritmi, je potrebno problem ustrezno predstaviti s podatki tako, da lahko z njimi učinkovito računamo. Najpogostejša načina predstavitve grafov sta t. i. matrika sosednosti in sosednostni seznam. Oba načina imata svoje prednosti in slabosti, katere si bomo v nadaljevanju tudi ogledali.

3.1.1 Matrika sosednosti

Matrika sosednosti (angl. *adjacency matrix*) M grafa G je matrika dimenzij $V \times V$, ki je definirana na sledeč način:

$$M_{ij} = \begin{cases} 1 & \text{če } (i,j) \in E; \\ 0 & \text{če } (i,j) \notin E. \end{cases} \tag{3.1}$$

Če imamo opravka z uteženim grafom, lahko namesto vrednosti 1 ali 0 hranimo vrednost $\ell(u,v)$, če povezava (u,v) obstaja, oziroma ∞, če ne obstaja:

$$M_{ij} = \begin{cases} \ell(i,j) & \text{če } (i,j) \in E; \\ \infty & \text{če } (i,j) \notin E. \end{cases} \tag{3.2}$$

Taki matriki M pravimo matrika razdalj (angl. *distance matrix*) grafa G. Prednost pri uporabi matrike sosednosti je zmožnost preverjanja in nastavljanja sosednosti dveh poljubnih vozlišč $u, v \in V$ v konstantnem času. Slabost je prostorska zahtevnost $\Theta(n^2)$, četudi je lahko sam graf zelo redek. Tudi sprehajanje po vseh sosednjih vozliščih nekega vozlišča $v \in V$ je časovno potratno, saj je za to, v najslabšem primeru, potrebno izvesti $O(n)$ operacij neglede na stopnjo vozlišča v.

3.1.2 Sosednostni seznam

Če imamo opravka z redkimi grafi, je velikokrat ugodnejše uporabiti sosednostni seznam (angl. *adjacency list*). Sosednostnih seznamov je dejansko več: eden za vsako vozlišče. Sosednostni seznam za neko vozlišče $v \in V$ definiramo kot seznam velikosti $deg^+(v)$, ki vsebuje vse take $u \in V$, za katere velja $(v,u) \in E$. Ker je teh seznamov n, je skupna velikost $\sum_{i=0}^{n} deg^+(i)$, kar znaša $\Theta(m)$. Tako kot pri matriki sosednosti, lahko v sezname vključimo tudi vrednost $\ell(v,u)$.

Prednost sosednostnih seznamov je manjša prostorska zahtevnost. Poleg tega nam sosednostni seznami omogočajo učinkovito sprehajanje po sosednjih vozliščih nekega vozlišča $v \in V$ in sicer v času $O(deg^+(v))$. Slabost je, da ne moremo preveriti sosednosti

dveh poljubnih vozlišč $u, v \in V$ v konstantnem času. Za slednje se moramo sprehoditi po celotnem seznamu vozlišča u in/ali vozlišča v.

Matrika sosednosti in sosednostni seznam nista nezdružljiva. V primeru, da moramo uporabiti matriko sosednosti lahko dodatno uporabimo tudi sosednostni seznam, saj se v tem primeru prostorska zahtevnost ne spremeni. To nam omogoča hitrejše sprehajanje po sosednjih vozliščih nekega vozlišča v kot, če bi uporabili samo matriko sosednosti.

3.2 ISKANJE V GLOBINO

Iskanje v globino [4] (angl. *depth-first search, DFS*) je preprost algoritem, ki omogoča obisk vseh vozlišč v grafu. Sam po sebi je uporaben za preverjanje povezanosti grafa. Uporablja se kot procedura v velikem številu algoritmov, od katerih si jih bomo tudi nekaj ogledali kasneje. DFS je prikazan v algoritmu 1.

Algoritem 1 Iskanje v globino.

```
procedure DFS(v, E)
    sklad := emptyStack
    v.obiskan := true
    sklad.push(v)
    repeat
        w := sklad.pop()
        for all u ∈ V : (w, u) ∈ E do
            if ¬u.obiskan then
                u.obiskan := true
                sklad.push(u)
            end if
        end for
    until sklad.empty()
end procedure
```

Prostorska zahtevnost algoritma DFS je $2n$, saj potrebujemo sklad velikosti n in po eno oznako za vsako obiskano vozlišče. Asimptotična časovna zahtevnost je $O(m+n)$, saj moramo v najslabšem primeru prečesati celoten graf, pri čemer se ne vračamo na že obiskana vozlišča, torej se nobena povezava ne ponovi.

3.3 ISKANJE V ŠIRINO

Iskanje v širino [4] (angl. *breadth-first search, BFS*) je preprost, DFS-ju soroden algoritem. Razlika med dvema algoritmoma je zgolj v uporabi vrste (BFS) in sklada (DFS). Tako kot DFS, se lahko tudi BFS uporablja za preverjanje povezanosti grafa in je pogosto uporabljan kot procedura v drugih algoritmih. Za razliko od DFS, se lahko BFS uporabi za iskanje najkrajših poti, kadar imamo opravka z neuteženimi grafi[1]. BFS je prikazan v algoritmu 2.

Tako časovna, kot tudi prostorska analiza algoritma BFS, je enaka algoritmu DFS, torej časovna zahtevnost $O(m+n)$ in prostorska zahtevnost $2n$.

[1] Oziroma, ekvivalentno, če imamo opravka z grafi, kjer so vse uteži enotske.

Algoritem 2 Iskanje v širino.

```
procedure BFS(v, E)
    vrsta := emptyQueue
    v.obiskan := true
    vrsta.enqueue(v)
    repeat
        w := vrsta.dequeue()
        for all u ∈ V : (w, u) ∈ E do
            if ¬u.obiskan then
                u.obiskan := true
                vrsta.enqueue(u)
            end if
        end for
    until vrsta.empty()
end procedure
```

3.4 TOPOLOŠKO UREJANJE

Topološka ureditev usmerjenega acikličnega grafa G je taka permutacija $\pi : V \to V$ vozlišč grafa, da velja: $\forall(u, v) \in E : \pi(u) < \pi(v)$, pri čemer je lahko tovrstnih permutacij π več, torej niso edinstvene. Topološko urejanje je možno le v primeru, da je graf acikličen. Obstaja množica algoritmov, ki tovrstno urejanje najdejo v času $O(m + n)$, npr. [16, 23], vendar si jih tukaj ne bomo podrobno ogledali.

3.5 ISKANJE KREPKO POVEZANIH KOMPONENT

Če želimo poiskati krepko povezane komponente grafa, lahko uporabimo naslednje algoritme: Tarjanov [22], Kosaraju-Sharir [21] in pa Cheriyan-Mehlhorn-Gabow [3,11]. Vsi od naštetih poiščejo krepko povezane komponente v času $O(m+n)$, torej linearnem v številu vozlišč in povezav. Tukaj si teh algoritmov ne bomo podrobneje ogledali.

3.6 ALGORITEM BELLMAN-FORD

Algoritem Bellman-Ford [1,8] rešuje problem SSSP. Prikazan je v algoritmu 3. Časovno in prostorsko zahtevnost je enostavno razbrati iz psevdokode. Časovna zahtevnost tako znaša $\Theta(n) + \Theta(mn) + \Theta(m) = \Theta(mn)$. Prostorska zahtevnost je enaka hrambi polja $d[]$, kar znaša natanko n.

Pravilnost algoritma Bellman-Ford nam zagotovi dokaz leme 3.1. Algoritem Bellman-Ford je asimptotično najhitrejši algoritem, ki rešuje splošen problem SSSP, tj. kjer lahko povezave zavzamejo tudi negativne vrednosti.

Lema 3.1. *Po izvedbi algoritma Bellman-Ford za nek izvor $s \in V$ v grafu, ki ne vsebuje ciklov negativne dolžine, hrani vsak element $d[t]$ polja $d[]$ dolžino najkrajše poti od izvora s do vozlišča t.*

Dokaz. Naj bo $\pi = \{s, v_1, ..., v_k\}$ najkrajša pot od vozlišča s do nekega vozlišča v_k. Ker so najkrajše poti aciklične, se lahko vsako vozlišče pojavi v π največ enkrat, torej

Algoritem 3 Algoritem Bellman-Ford.

```
procedure BELLMAN-FORD(V, E, d[], s)
    for all v ∈ V do
        d[v] := ∞
    end for
    d[s] := 0
    for i := 1 to n − 1 do
        for all (u, v) ∈ E do
            d[v] := min(d[v], d[u] + ℓ(u, v))
        end for
    end for
    for all (u, v) ∈ E do
        if d[u] + ℓ(u, v) < d[v] then
            return graf vsebuje negativni cikel
        end if
    end for
end procedure
```

$k \leq n-1$ in pot ima največ $n-1$ povezav. Ker algoritem izvede $n-1$ iteracij in ob vsaki iteraciji i sprosti vse povezave $e \in E$, je ena izmed njih vsebovana v najkrajši poti π na mestu i, kar pomeni, da $d[i] = d(s,i)$. Po izvedenih $n-1$ iteracijah torej velja $\forall v \in V : d[v] = d(s,v)$. □

3.7 ALGORITEM FLOYD-WARSHALL

Algoritem Floyd-Warshall [7, 28] je preprost algoritem, ki rešuje problem APSP. Prikazan je v algoritmu 4. Algoritem je izredno preprost, sestavljen iz treh *for* zank in ene operacije *min*. Enostavno je videti, da je časovna zahtevnost algoritma $O(n^3)$ oziroma natančneje $\Theta(n^3)$. Prostorska zahtevnost je n^2, saj algoritem uporablja matriko razdalj. Algoritem reši problem najkrajših poti z uporabo dinamičnega programiranja.

Algoritem 4 Algoritem Floyd-Warshall.

```
procedure FLOYD-WARSHALL(V, W)
    for all k ∈ V do
        for all i ∈ V do
            for all j ∈ V do
                W_ij := min(W_ij, W_ik + W_kj)
            end for
        end for
    end for
end procedure
```

Nekoliko težje je videti, kljub enostavnosti, zakaj algoritem deluje. Algoritem Floyd-Warshall temelji na naslednji lastnosti najkrajših poti. Vzemimo neko množico vozlišč $\{v_0, v_1, ..., v_k\}$. Potem za vsak par vozlišč $i, j \in V$ lahko definiramo neko pot π_{ij}^k od vozlišča i do j, ki je najkrajša pot upoštevajoč zgolj vmesna vozlišča $\{v_0, v_1, ..., v_k\}$. Enostavno je videti, da v kolikor $k = n$, potem je to najkrajša pot, tj. $\pi_{ij}^n = d(i,j)$.

Algoritem Floyd-Warshall izrabi naslednjo povezavo med potjo π_{ij}^k in potjo π_{ij}^{k-1}, tj. potjo, ki uporablja samo vozlišča $\{v_0, v_1, ..., v_{k-1}\}$:

1. Če vozlišče k ni uporabljeno v poti π_{ij}^k, potem velja $\pi_{ij}^k = \pi_{ij}^{k-1}$.

2. Če je vozlišče k uporabljeno v poti π_{ij}^k, potem lahko pot π_{ij}^k razbijemo na dve poti: π_{ik}^k in π_{kj}^{k-1}. Iz leme 2.2 (Optimalnost podpoti) vemo, da sta obe dobljeni poti najkrajši, če je pot π_{ij}^k najkrajša pot. Ker vemo, da se pot π_{ik}^k konča v vozlišču k, ne more vsebovati vozlišča k, kot vmesnega vozlišča, torej velja $\pi_{ik}^k = \pi_{ik}^{k-1}$.

Algoritem Floyd-Warshall ni torej nič drugega, kot direktna implementacija zgoraj naštetih lastnosti, zato je pravilnost preprosto razbrati kar iz algoritma.

3.8 DIJKSTROV ALGORITEM

Dijkstrov algoritem [5] rešuje problem SSSP kadar uteži povezav zavzamejo nenegativne vrednosti, tj. $\ell : E \to \mathbb{R}^+$. Prikazan je v algoritmu 5. Lemi 3.2 in 3.3 nam zagotovita pravilnost Dijkstrovega algoritma.

Algoritem 5 Dijkstrov algoritem.

```
procedure DIJKSTRA(V, E, d[], s)
    for all v ∈ V do
        d[v] := ∞
    end for
    d[s] := 0
    Q := V
    repeat
        u := arg min_{v∈Q} d[v]
        Q := Q \ {u}
        for all v ∈ V : (u, v) ∈ E do
            d[v] := min(d[v], d[u] + ℓ(u, v))
        end for
    until Q = ∅
end procedure
```

Lema 3.2. *Kadar v Dijkstrovem algoritmu vzamemo element v iz množice Q, je dolžina najkrajše poti vsebovana v $d[v]$.*

Dokaz. Ker so vrednosti vseh vozlišč, razen s, nastavljene na ∞, je s prvi element, ki ga vzamemo iz Q. Lema trivialno drži za prvo vozlišče s (izvor). Sedaj moramo dokazati, da drži tudi ob vsaki iteraciji. Naj bo vozlišče u prvo tako vozlišče, za katerega velja $d[u] \neq d(s, u)$. Vemo, da $s \neq u$, prav tako mora obstajati pot od vozlišča s do vozlišča u, sicer drugače $d[u] = d(s, u) = \infty$.

Naj bo $\pi = \{s, v_0, v_1, ..., v_k, v_{k+1}, ..., u\}$ najkrajša pot, ki povezuje s in u. Naj velja tudi $\forall v_i : i < k$, da $v_i \in V \setminus Q$ ter $\forall v_j : j \geq k$, da $v_j \in Q$. Trdimo, da $\forall v_i : i < k$ velja $d[v_i] = d(s, v_i)$. Ker smo v predpostavki zahtevali, da je u prvo tako vozlišče, za katerega velja $d[u] \neq d(s, u)$, je zgornja trditev pravilna, saj so bila vsa

vozlišča v_i odstranjena iz množice Q. Iz definicije najkrajše poti π vemo, da obstaja povezava (v_{k-1}, v_k), ki zaradi optimalnosti podpoti tvori najkrajšo pot med v_{k-1} in v_k. Po predpostavki vemo, da $d[v_{k-1}] = d(s, v_{k-1})$. Ker algoritem uporablja sproščanje, to pomeni, da smo sprostili vse izhodne povezave takih vozlišč v_i. Iz tega sledi, da $d[v_k] = d(s, v_k)$.

Sedaj trdimo, da je pot $\pi' = \{s, v_0, v_1, ..., v_k\}$ kvečjemu krajša od poti π. Ker pot π vsebuje dodatne povezave in ker imamo opravka z nenegativno uteženimi grafi, to trivialno drži. Ker vozlišče v_k ni bilo odstranjeno iz množice Q pred vozliščem u to pomeni $d[v_k] = d[u]$. Torej $d[u] = d(s, v_k) = d(s, u)$ kar je protislovje saj je bila predpostavka, da $d[u] \neq d(s, u)$. Lema torej drži. □

Lema 3.3. *Po izvedbi Dijkstrovega algoritma v grafu z nenegativnimi utežmi imamo* $\forall v \in V : d[v] = d(s, v)$.

Dokaz. Algoritem se konča kadar $Q = \emptyset$. Po lemi 3.2 velja $\forall v \in V \setminus Q : d[v] = d(s, v)$, torej velja $\forall v \in V : d[v] = d(s, v)$. □

Prostorska zahtevnost Dijkstrovega algoritma je $O(n)$, saj moramo hraniti polje dolžin najkrajših poti $d[]$ in množico vozlišč Q, velikosti $O(n)$. Časovna zahtevnost algoritma je $O(nT_{min}+m)$, kjer je T_{min} čas, ki je potreben za operacijo $\arg\min_{v \in Q} d[v]$. To lahko izvedemo z naivnim pregledom vseh vrednosti $d[v]$, kar znaša $O(n)$. Z uporabo naivnega iskanja najmanjšega elementa je časovna zahtevnost algoritma $O(n^2)$.

Boljše izvedbe zgoraj omenjene operacije so možne z uporabo vrst s prednostjo. Najboljši čas dobimo z uporabo Fibonaccijevih kopic [9], katere zmanjšajo časovno zahtevnost algoritma na $O(n \lg n + m)$. Tukaj in v nadaljevanju pišemo $\lg n$, kjer mislimo na dvojiški logaritem $\log_2 n$. Dijkstrov algoritem temelji na učinkoviti implementaciji vrste s prednostjo. Posledično je problem izboljšave časovne zahtevnosti Dijkstrovega algoritma ekvivalenten problemu izboljšave časovne zahtevnosti algoritmov urejanja [25]. V kolikor imamo opravka s celoštevilskimi utežmi, lahko uporabimo celoštevilske vrste s prednostjo [26], ki zmanjšajo časovno zahtevnost na $O(n \lg \lg n + m)$.

3.9 JOHNSONOV ALGORITEM

Johnsonov algoritem [15] rešuje problem APSP. Gre za preprosto združitev dveh algoritmov, ki smo jih že spoznali: algoritem Bellman-Ford in Dijkstrov algoritem. Kot že vemo, lahko Dijkstrov algoritem uporabimo za reševanje problema APSP, če velja: $\ell : E \to \mathbb{R}^+$. Johnsonov algoritem deluje tudi na grafih, kjer so uteži negativne, vendar ne vsebujejo ciklov negativne dolžine. Algoritem deluje tako, da najprej pokliče algoritem Bellman-Ford in na podlagi dobljenih poti spremeni vse uteži v grafu v pozitivne. Nato pokliče Dijkstrov algoritem n krat, ki reši problem APSP. Prikazan je v algoritmu 6.

Postopek spremembe uteži je edina novost, ki jo uvede Johnsonov algoritem, vendar je izredno pomembna, saj gre za splošno redukcijo iz problema poljubno uteženih grafov na problem nenegativno uteženih grafov. Postopek redukcije najprej uvede novo vozlišče i ter doda povezave $\forall v : (i, v)$ pri čemer $\ell(i, v) = 0$. Nato pokliče algoritem Bellman-Ford z izvorom $s = i$. Dobljene najkrajše poti oblike $\forall v \in V : d(i, v)$ nato uporabi tako, da spremeni utež posamezne povezave $\forall (u, v) \in E : \ell(u, v) = \ell(u, v) + d(i, u) - d(i, v)$. Dokaz leme 3.4 nam zagotovi, da so tako dobljene uteži nenegativne.

Algoritem 6 Johnsonov algoritem.

```
1: procedure JOHNSON(V, E, W)
2:     V' := V ∪ {i}
3:     E' := E
4:     for all v ∈ V do
5:         E' := E' ∪ {(i, v)}
6:         ℓ(i, v) := 0
7:     end for
8:     d[] := new_array
9:     BELLMAN-FORD(V', E', d, i)
10:    for all (u, v) ∈ E do
11:        ℓ(u, v) := ℓ(u, v) + d[u] − d[v]
12:    end for
13:    for all v ∈ V do
14:        d[] := new_array
15:        DIJKSTRA(V, E, d, v)
16:        W_v := d
17:    end for
18: end procedure
```

Lema 3.4. *Po izvedenem postopku spremembe uteži v Johnsonovem algoritmu so vse uteži nenegativne in se najkrajše poti v grafu, ki ne vsebuje ciklov negativne dolžine, ne spremenijo.*

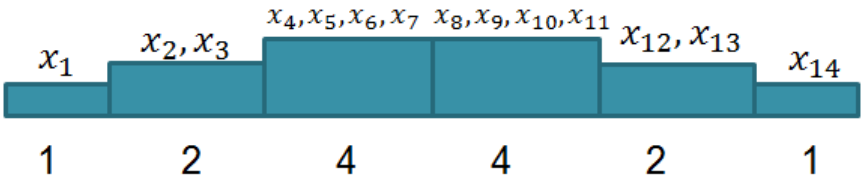
Dokaz. Najprej bomo dokazali, da se najkrajše poti v grafu po izvedenem postopku (vrstice $10 - 12$ v algoritmu 6) ne spremenijo. Vzemimo neko pot π med vozlišči s, t. Naj bo $\pi = \{(s, p_0), (p_0, p_1), ..., (p_k, t)\}$. Definirajmo $h(v) = d(i, v)$ zaradi lažjega zapisa. Naj bo $\ell(\pi)$ dolžina poti pred postopkom spremembe uteži in $\bar{\ell}(\pi)$ dolžina iste poti po postopku spremembe uteži. Pred postopkom je dolžina poti $\ell(\pi) = \ell(s, p_0) + \ell(p_0, p_1) + \ell(p_1, p_2)... + \ell(p_k, p_t)$. Torej je po izvedenem postopku dolžina iste poti $\bar{\ell}(\pi) = \ell(s, p_0) + h(s) - h(p_0) + \ell(p_0, p_1) + h(p_0) - h(p_1) + ... + \ell(p_k, t) + h(p_k) - h(t)$. Členi se izničijo in ostane nam $\bar{\ell}(\pi) = \ell(\pi) + h(s) - h(t)$. Torej vsaka pot s, t ima dodano enako količino $h(s) - h(t)$. Posledično, če je pot najkrajša pred postopkom spremembe uteži ostane najkrajša tudi po postopku spremembe uteži.

Sedaj moramo še dokazati, da so vse dobljene uteži nenegativne. Ker smo $h(u)$ in $h(v)$ definirali kot najkrajše poti $d(i, u)$ ter $d(i, v)$, lahko uporabimo trikotniško neenakost: $h(u) + \ell(u, v) \geq h(v)$. Člene postavimo na levo stran in dobimo: $\ell(u, v) + h(u) - h(v) \geq 0$. Dobljene uteži so torej nenegativne. □

3.10 SPLIT-FINDMIN

Tukaj bomo opisali podatkovno strukturo Split-findmin [10, 19], ki ima osrednjo vlogo v nekaterih novejših algoritmih za iskanje najkrajših poti [14, 18, 20, 24]. Podatkovno strukturo definiramo z naslednjim naborom operacij:

Initialize$(e_1, e_2, ..., e_n)$. Inicializira seznam $\mathcal{S} := \{(e_1, e_2, ..., e_n)\}$ kjer $\kappa(e_i) := \infty$ za vse $1 \leq i \leq n$. Potem definiramo $S(e_i)$ kot zaporedje v $\mathcal{S}$, ki vsebuje element e_i.



Slika 3.1: Primer bitonične organizacije blokov nad štirinajstimi elementi. Elementi, vsebovani v posameznem bloku, so našteti nad blokom. Velikosti blokov so označene pod bloki.

Split(e_i). Naj bo $S(e_i) = (e_j, ..., e_{i-1}, e_i, ..., e_k)$. Potem nastavimo $\mathcal{S} := (\mathcal{S} \setminus S(e_i)) \cup \{(e_j, ..., e_{i-1}), (e_i, ..., e_k)\}$.

Findmin(e). Vrne $\min_{f \in S(e)}\{\kappa(f)\}$.

Decreasekey(e, w). Nastavi $\kappa(e) := min(\kappa(e), w)$.

V nadaljevanju si bomo ogledali, kako lahko te operacije implementiramo čim bolj učinkovito.

3.10.1 Preprosta izvedba

Najprej si bomo ogledali preprosto izvedbo strukture, kot jo opisuje lema 3.7. Potrebovali bomo še naslednji definiciji.

Definicija 3.5. (Cikličen premik zaporedja) *Cikličen premik nekega zaporedja je novo zaporedje, kjer elemente starega zaporedja premaknemo za eno mesto v desno tako, da zadnji element starega zaporedja postane prvi element novega zaporedja.*

Definicija 3.6. (Bitonično zaporedje) *Bitonično zaporedje je zaporedje, kjer elementi najprej naraščajo po velikosti, nato padajo. Zaporedje $x_1, x_2, ..., x_n$ je bitonično, če in samo, če obstaja tak k, da za nek cikličen premik zaporedja velja:*

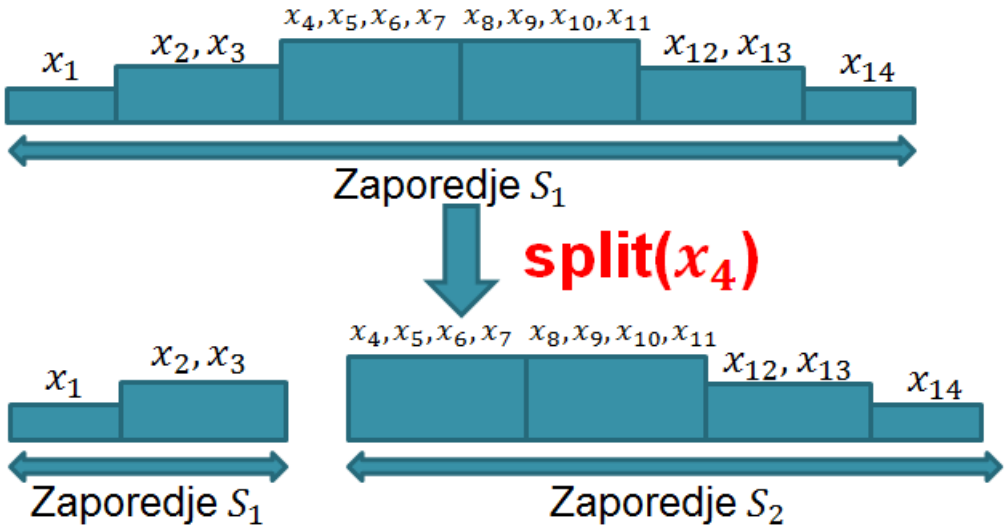
$$x_1 \leq x_2 \leq ... \leq x_k \geq ... \geq x_{n-1} \geq x_n$$

Lema 3.7. *Obstaja podatkovna struktura Split-findmin, ki podpira decreasekey operacije v $O(1)$ času in treh primerjavah. Vse ostale operacije potrebujejo $O(n \lg n)$ časa in manj kot $3n \lg n - 2n$ primerjav.*

Vsako zaporedje elementov bo razdeljeno na množico zaporednih blokov, ki vsebujejo elemente. Velikosti blokov (število vsebovanih elementov) bodo potence števila dve. Ker je vsako zaporedje sestavljeno iz več blokov, bomo zahtevali, da je vsako tako zaporedje blokov urejeno v bitoničnem vrstnem redu. Od leve proti desni bodo velikosti blokov najprej naraščale, nato pa padale, pri čemer bosta dva največja bloka lahko enake velikosti. Primer take organizacije je podan na sliki 3.1.

Vsak blok bo imel kazalec, ki bo kazal na najmanjši vsebovani element. Podobno bo tudi vsako zaporedje vsebovalo kazalec na najmanjši element, vsebovan v tem zaporedju. Vsak element bo vseboval kazalec na blok, v katerem je vsebovan. Podobno bo tudi vsak blok vseboval kazalec na zaporedje, v katerem je vsebovan. Sedaj bomo opisali izvedbo operacij nad podatkovno strukturo.

Decreasekey(e, w). Najprej preverimo, če je nov ključ manjši od trenutnega (prva primerjava). Če je nov ključ manjši, pogledamo v katerem bloku je vsebovan element. Nato preverimo, če je potrebno zamenjati kazalec na najmanjši element v bloku (druga



Slika 3.2: Primer izvedbe split operacije, kjer se bloki ne spremenijo.

primerjava). Če je element e postal najmanjši element v bloku, moramo še preveriti, ali je potrebno zamenjati kazalec na najmanjši element v zaporedju (tretja primerjava). Za izvedbo operacije smo potrebovali tri primerjave. Čas je konstanten, saj imamo kazalce iz posameznih elementov na blok, v katerem so vsebovani ter iz bloka na zaporedje, v katerem je vsebovan blok.

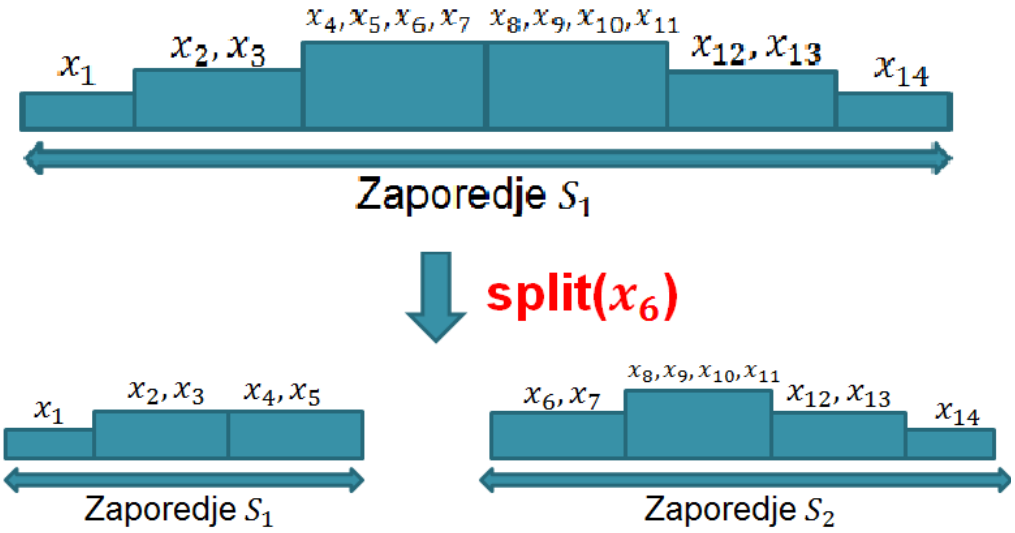
Findmin(e). Začnemo pri elementu e. Gremo do bloka, ki vsebuje e preko kazalca, nato gremo do zaporedja, ki vsebuje blok. Po definiciji, zaporedje že hrani kazalec na minimalni element, ki ga enostavno vrnemo kot rezultat. Operacijo smo izvedli v konstantnem času z uporabo kazalcev ter za to nismo potrebovali nobene primerjave.

Split(e_i). Denimo, da želimo izvesti split nad elementom e_i, ki leži v bloku b. Če je e_i prvi element bloka b, potem lahko enostavno razbijemo zaporedje na dva dela, kjer bloki ostanejo isti, zamenjamo jim le pripadnost zaporedju. Primer takega razbitja je prikazan na sliki 3.2.

Sedaj je potrebno še posodobiti kazalce. Opazimo naslednje: ker se bloki niso spremenili, je potrebno le nastaviti kazalce iz blokov na zaporedja, ki jih vsebujejo ter kazalce zaporedij na najmanjše vsebovane elemente. Eno izmed zaporedij očitno ohrani najmanjši element, torej ni potrebno zamenjati kazalca za tisto zaporedje. Poiskati moramo še nov minimalni element dobljenega zaporedja. To storimo tako, da se sprehodimo po blokih, ki so vsebovani v novem zaporedju ter z uporabo kazalca na najmanjši element v bloku, poiščemo najmanjši element v vsebovanih blokih.

V splošnem je problem težji. Kadar izvedemo split nad elementom e_i, ki leži v nekem bloku b, vendar ni prvi element tega bloka, je potrebno razbiti blok b na manjše bloke. Denimo, da $b = (e_j, ..., e_{i-1}, e_i, ..., e_k)$. Sprehodili se bomo po $(e_j, ..., e_{i-1})$ od *leve* proti desni in ga razbili na največ $\lceil \lg((i-1)-j+1) \rceil = \lceil \lg(i-j) \rceil$ blokov, katerih velikosti bodo padajoče potence števila dve. Potem se bomo sprehodili po $(e_i, ..., e_k)$ od *desne* proti levi in blok razdelili na $\lceil \lg(k-i+1) \rceil$ blokov, katerih velikosti bodo tudi padajoče potence števila dve. Primer takega razbitja je prikazan na sliki 3.3.

S takim sprehodom (od leve proti desni in nato obratno) smo zagotovili, da je zaporedje blokov bitonično znotraj posameznega zaporedja, ki si bloke lasti. Tako kot pri enostavnejšem primeru, je potrebno še posodobiti kazalce in minimalne elemente.



Slika 3.3: Primer izvedbe split operacije, kjer se bloki spremenijo.

Sedaj si bomo ogledali koliko časa in primerjav potrebujemo za izvedbo tovrstnih posodobitev. Opazimo, da je posamezen element vključen v največ $\lceil \lg(n+1) \rceil$[2] različnih blokih skozi življenjsko dobo strukture, saj ob vsaki split operaciji kvečjemu zmanjšamo velikost bloka. Primerjave, ki jih je potrebno izvesti za vsako split operacijo, lahko razdelimo na dva tipa: (a) primerjave, ki poiščejo minimalni element novih blokov ter (b) primerjave, ki poiščejo minimalni element novega zaporedja.

Število primerjav in čas potreben za posodobitve tipa (a) je vezano na dejstvo, da je vsak element del največ $\lceil \lg(n+1) \rceil$ različnih blokov. Vsak element torej »plača« konstanten čas dela za vsak blok, v katerem je vsebovan. Ker je vseh elementov n, to znese ravno $O(n \lg n)$ časa za posodobitve tipa (a) skozi življenjsko dobo strukture. Potrebujemo še število potrebnih primerjav. Tukaj moramo biti zelo natančni, saj bodo konstantni faktorji in aditivne konstante pomembne v nadaljevanju. Očitno je, da kadar blok razpade na en sam element, ni potrebno izvesti nobene primerjave – edini element je tudi najmanjši. Vemo tudi, da je takih blokov največ n. Podobno, če blok razpade na dva elementa, je potrebna samo ena primerjava. Takih blokov je $n/2$. Za blok, ki vsebuje štiri elemente so potrebne tri primerjave, takih blokov je $n/4$, in tako naprej. To lahko zapišemo formalno kot vsoto in dobimo:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor (2^i - 1) \leq n\lfloor \lg n \rfloor - n + 1$$

Število primerjav tipa (a) je torej največ $n\lfloor \lg n \rfloor - n + 1$.

Sedaj je potrebno še prešteti čas in primerjave potrebne za posodobitve tipa (b), ki se nanašajo na posodabljanje zaporedij. Spomnimo, da je vsako zaporedje sestavljeno iz bitoničnega zaporedja blokov, velikosti katerih so potence števila dve. Iz tega sledi, da je vsako zaporedje sestavljeno iz največ $2\lfloor \lg n \rfloor$ blokov. To pomeni, da bi morali ob vsaki split operaciji nad zaporedjem izvesti največ $2\lfloor \lg n \rfloor - 1$ primerjav, da lahko poiščemo nov minimalni element. Vseh split operacij je lahko največ n skozi celotno življenjsko

[2] V primeru, da $n = 1$ potem $\lg n = 0$, vendar očitno mora biti element vsebovan v vsaj enem bloku, torej $\lg(n+1)$.

dobo strukture, saj je potem vsako zaporedje sestavljeno iz enega samega elementa. Potrebnih primerjav tipa (b) skozi življenjsko dobo strukture je potem $2n\lfloor \lg n \rfloor - n$. Očitno je, da je čas $O(n \lg n)$, saj je iskanje najmanjšega elementa enostaven sprehod po seznamu vsebovanih blokov. Opisana podatkovna struktura izpolnjuje zahteve leme 3.7 in je posledično dokaz njene veljavnosti.

3.10.2 Rekurzivna izvedba

Najprej opazimo, da je do sedaj opisana podatkovna struktura že optimalna za primer, kadar je število decreasekey operacij omejeno z $\Omega(n \lg n)$, saj je potem skupen čas linearen. Sedaj bomo pokazali, kako lahko prej-omenjeno strukturo implementiramo na rekurziven način, ki omogoča vedno hitrejše split operacije, vendar pa tudi dražje decreasekey operacije. V nadaljevanju bomo nekoliko izrabljali asimptotično notacijo velikega O in pisali $O(i+1)$, čeprav gre za konstanto. Potrebovali bomo Ackermannovo funkcijo in njena inverza [19]:

Definicija 3.8. (Ackermannova funkcija $A(m,n)$)

$$\begin{array}{rcll} A(i+1,j+1) & = & A(i+1,j) \cdot A(i, A(i+1,j)) & \text{če } i,j \geq 1 \\ A(i,1) & = & 2 & \text{če } i > 1 \\ A(1,j) & = & 2^j & \text{če } j \geq 1 \end{array}$$

Definicija 3.9. (Inverz Ackermannove funkcije $\alpha(m,n)$)

$$\alpha(m,n) = \min\{i : A(i, \left\lceil \frac{2n+m}{n} \right\rceil) > n\}$$

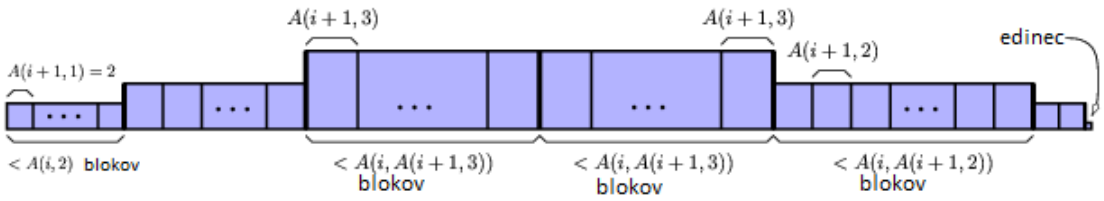
Definicija 3.10. (Stolpični inverz Ackermannove funkcije $\lambda_i(n)$) *Inverz Ackermannove funkcije za točno določen stolpec (tj. določeno vrednost prvega parametra). Zapišemo kot:*

$$\lambda_i(n) = \min\{j : A(i,j) > n\}$$

Lema 3.11. *Denimo, da obstaja Split-findmin struktura $\mathcal{SF}_i$, ki potrebuje $O(i)$ časa in $2i+1$ primerjav za operacijo decreasekey ter $O(in\lambda_i(n))$ časa in $3in\lambda_i(n)$ primerjav za vse ostale operacije. Potem obstaja Split-findmin struktura $\mathcal{SF}_{i+1}$, ki potrebuje $O(i+1)$ časa in $2i+3$ primerjav za operacijo decreasekey ter $O((i+1)n\lambda_{i+1}(n))$ časa in $3(i+1)n\lambda_{i+1}(n)$ primerjav za vse ostale operacije.*

Podatkovna struktura $\mathcal{SF}_{i+1}$ bo vsako zaporedje dolžine n' smatrala kot zaporedje največ $2(\lambda_{i+1}(n')-1)$ *platojev* (tj. najmanjše tako število j, da velja $A(i+1,j) > n'$) in največ dveh *edincev*. Plato je skupina blokov enake velikosti (kot nekakšen superblok), pri čemer je število in velikost blokov v platoju definirano z *nivojem* platoja. Plato nivoja j ima manj kot $A(i+1,j+1)/A(i+1,j) = A(i, A(i+1,j))$ blokov, katerih velikosti so natanko $A(i+1,j)$. V vsakem zaporedju so nivoji platojev urejeni v bitoničnem vrstnem redu, kjer za nek nivo j obstajata največ dva platoja s tovrstnim nivojem. Primer strukture je prikazan na sliki 3.4.

Inicializacijo počnemo tako, da struktura $\mathcal{SF}_{i+1}$ razdeli začetno zaporedje na največ $\lambda_{i+1}(n) - 1$ platojev in enega edinca. Vsakemu platoju dodelimo kazalec, ki kaže na zaporedje, v katerem je vsebovan. Struktura $\mathcal{SF}_i$ skrbi za posamezne platoje, kot ločene instance problema Split-findmin, kjer definiramo bloke kot elemente novega



Slika 3.4: Primer strukture s šestimi platoji in enim edincem. Povzeto po [19].

problema. Ključ elementa (bloka) v tem primeru definiramo kot ključ najmanjšega elementa vsebovanega v bloku.

Zaporedja in bloki vsebujejo kazalce na svoje najmanjše elemente. Sedaj si oglejmo, kako vpeljati željene operacije nad podatkovno strukturo.

Findmin(e). Tako kot prej, imamo kazalce na najmanjši element zaporedja, ki ga enostavno vrnemo. Ne potrebujemo nobene primerjave in zgolj $O(1)$ časa.

Decreasekey(e, w). Najprej preverimo, če je nov ključ manjši, kar doprinese eno primerjavo. Nato preverimo še, če je to nov minimalni element zaporedja ter kazalec zaporedja ustrezno popravimo. Če je element e edinec, smo zaključili, sicer je e vsebovan v nekem platoju p, ali bolj natančno v nekem bloku b znotraj platoja p. Sedaj pokličemo decreasekey(b, w) tako, kot je implementiran v $\mathcal{SF}_i$. Spomnimo, da struktura $\mathcal{SF}_i$ smatra bloke kot elemente novega problema Split-findmin. Če podatkovna struktura $\mathcal{SF}_i$ potrebuje $2i+1$ primerjav in $O(i)$ časa, potem $\mathcal{SF}_{i+1}$ potrebuje $2i+3$ primerjav ter $O(i+1)$ časa.

Split(e_i). V primeru, da želimo izvesti split nad nekim elementom e_k, ki je prvi element bloka b, ki je hkrati prvi blok nekega platoja p, potem enostavno premestimo plato p v drugo zaporedje. Platojev in blokov v tem primeru ni potrebno razbijati – ta primer je podoben tistemu pri enostavni izvedbi, kjer smo izvedli split operacijo nad prvim elementom bloka. Seveda je v splošnem primer težji. Denimo, da želimo izvesti split operacijo nad elementom e_k vsebovanem v bloku b, ki pa je vsebovan v platoju p nivoja j. Uporabimo split operacijo strukture $\mathcal{SF}_i$ nad e_{k-1} in e_{k+1} (elementa, ki ležita levo in desno od e_k v bloku b). Nato razdelimo e_{k-1} in e_{k+1} na bloke in platoje (katerih nivoji so nujno manjši od j), tako kot pri inicializaciji. Podobno kot pri preprosti izvedbi, je potrebno paziti na bitonični vrstni red nove delitve, zato inicializacijo izvajamo od leve proti desni za e_{k-1} in od desne proti levi za e_{k+1}. Eno izmed zaporedij ohrani najmanjši element. Za drugo zaporedje ga poiščemo tako, da poiščemo najmanjši element izmed platojev z uporabo Findmin operacije strukture $\mathcal{SF}_i$ ter najmanjšega izmed največ dveh edincev. Število primerjav in čas potreben za podatkovno strukturo $\mathcal{SF}_{i+1}$ analiziramo na naslednji način:

- **Primerjave tipa (A)** To so primerjave, ki so potrebne za iskanje minimalnih elementov blokov. Skozi življenjsko dobo strukture, vsak element pripada k največ $\lambda_{i+1}(n) - 1$ blokom. Število primerjav tipa (A) je potem dano z:

$$\sum_{j\geq 1}^{\lambda_{i+1}(n)-1} (n - \frac{n}{A(i+1,j)})$$

  Ker velja $A(i+1,1) = 2$ lahko omejimo število primerjav tipa (A) z:

$$\sum_{j\geq 1}^{\lambda_{i+1}(n)-1} (n - \frac{n}{A(i+1,j)}) \leq n(\lambda_{i+1}(n) - \frac{3}{2}).$$

- **Primerjave tipa (B)** To so primerjave, ki so potrebne za iskanje minimalnih elementov zaporedij. Ker je vsako zaporedje sestavljeno iz največ $2(\lambda_{i+1}(n) - 1)$ platojev in dveh edincev, je število teh primerjav enostavno $n(2\lambda_{i+1}(n) - 1)$.
- **Primerjave tipa (C)** To so primerjave, ki jih izvede podatkovna struktura $\mathcal{SF}_i$. Najprej opazimo, da se vsak blok velikosti $A(i, j + 1)$ pojavi v strukturi $\mathcal{SF}_i$, ki ima skupno manj kot $A(i + 1, j + 1)/A(i + 1, j) = A(i, A(i + 1, j))$ elementov. Število primerjav tipa (C) je torej:

$$\sum_{1 \le j \le \lambda_{i+1}(n)} \frac{3in\lambda_i(A(i, A(i+1, j)) - 1)}{A(i+1, j)} = \\ = 3in(\lambda_{i+1}(n) - 1).$$

Če sedaj seštejemo vse zgoraj omenjene primerjave skupaj, dobimo manj kot $3(i + 1)n\lambda_{i+1}(n)$ primerjav, kar dokazuje lemo 3.11.

Izrek 3.12. *Obstaja podatkovna struktura $\mathcal{SF}_\alpha$, ki teče v času $O((m + n)\alpha(m, n))$.*

Dokaz. Lemi 3.7 in 3.11 skupaj dokažeta, da $SF_{\alpha(m,n)}$ teče v času $O(\alpha(m, n)n\lambda_{\alpha(m,n)}(n)+ m\alpha(m, n))$, ki pa je $O((m + n)\alpha(m, n))$ saj $\lambda_{\alpha(m,n)}(n) = O(1 + \frac{m}{n})$. □

3.11 HIERARHIČNI ALGORITMI

Opisali bomo nove prijeme, ki se uporabljajo pri reševanju problema najkrajših poti v grafih. Algoritmi, ki spadajo v razred hierarhičnih pristopov reševanja, predstavljajo odmik od Dijkstrovega algoritma. Razlika med Dijkstrovim algoritmom in algoritmi, ki delujejo po principu hierarhije, je predvsem ta, da slednji ne nujno obiskujejo vozlišča v naraščajočem vrstnem redu. V nadaljevanju se bomo omejili na digrafe s celoštevilčnimi utežmi. Hagerupov algoritem [14], ki ga bomo tukaj opisali, se lahko ustrezno dopolni tudi tako, da deluje na realnih številih. Kako narediti slednje, je prikazano v [18]. V primeru, da želimo poiskati najkrajše poti v neusmerjenih grafih s celoštevilčnimi ali realnimi utežmi, obstajajo učinkoviti hierarhični algoritmi, ki so bili tudi prvi te vrste [20, 24], vendar jih tukaj ne bomo opisali.

V tem poglavju bo predstavljen algoritem, ki reši problem APSP v času $O(n^2 \lg \lg n)$. Pri opisu Dijkstrovega algoritma smo že poudarili, da obstaja celoštevilčna vrsta s prednostjo, ki dovoli tudi Dijkstrovemu algoritmu rešitev APSP v času $O(n^2 \lg \lg n)$. Hagerupov algoritem je bil objavljen leta 2000, omenjena celoštevilčna vrsta pa šele leta 2003. Poleg samega izvajalnega časa, je pomemben tudi postopek, ki ga ubere Hagerupov algoritem, saj se bistveno razlikuje od Dijkstrovega.

Pravilnost algoritma bomo dokazali z uporabo trikotniške neenakosti. Zagotovili bomo, da vse razdalje upoštevajo trikotniško neenakost in so torej pravilne. Enostavno je formulirati algoritem podoben Dijkstrovem, ki postopoma povečuje vse spodnje ocene razdalj $\delta(v) = \delta(v) + T$ in obišče vozlišče v kadar $\delta(v) = D(v)$ ter sprosti vse povezave $(v, u) \in E$. Če imamo opravka s celoštevilčnimi utežmi, tj. $\ell : E \to \mathbb{Z}^+$ in $T = 1$, je algoritem pravilen, saj zadosti trikotniški neenakosti ob trivialnem pogoju, da $\forall(u, v) \in E : \ell(u, v) \ge 1$ [3]. Seveda to ne velja v primeru, da $\ell : E \to \mathbb{R}^+$, vendar je možno izbrati $T = \min_{(u,v)\in E} \ell(u, v)$ in je ob istem pogoju algoritem tudi pravilen.

[3] Sicer je možno, da $\ell(u, v) = 0$, vendar lahko v tem primeru problem enostavno rešimo ob obisku vozlišča u tako, da takoj obiščemo vsa taka neobiskana vozlišča v.

V kolikor bi želeli tak algoritem realizirati, bi nastopil problem pri povečevanju spodnjih ocen, saj je to zelo potraten postopek. Ideja, ki jo izrabljajo hierarhični algoritmi je, da ni potrebno povečevati vseh spodnjih ocen enakomerno. Izbira univerzalnega T, kot najkrajše povezave v grafu, nam zagotavlja, da je najkrajša pot med dvema *poljubnima* vozliščema u in v vsaj T. V kolikor vemo, da je možno graf razdeliti na dve disjunktni skupini vozlišč V_1 in V_2, in če dodatno vemo, da vsaka pot iz $v_1 \in V_1$ do $v_2 \in V_2$ in pa iz v_2 do v_1 prečka povezavo, ki je dolžine vsaj T_k, potem se lahko spodnji oceni teh dveh skupin vozlišč razlikujeta med seboj za vsaj T_k. Ker želimo ta koncept izrabiti čim bolje, ga lahko definiramo na nek način rekurzivno tako, da tudi V_1 in V_2 razdelimo na podmnožice vozlišč in postopek ponavljamo. V nadaljevanju bomo podali način, kako to lahko počnemo učinkovito, kar je osredje vseh hierarhičnih algoritmov.

3.11.1 Preprosta gradnja hierarhije

Definirali bomo hierarhijo, ki predstavlja točno take relacije med skupinami vozlišč, kot smo jih opisali. Ker imamo opravka z utežmi, ki so cela števila, lahko utež vsake povezave $(u, v) \in E$ podamo s številom $i \in \mathbb{Z}^+$, ki mu pravimo *nivo* povezave, tako, da $2^{i-1} \leq \ell(u, v) < 2^i$, kjer v primeru $\ell(u, v) = 0$ definiramo $i = 0$. Naj bo potem G_i tak podgraf grafa G, da so v njem vsebovane samo povezave, katerih nivo je največ (vključno z) i. Sedaj lahko definiramo *hierarhijo komponent* kot drevo T, kjer je vsako vozlišče drevesa označeno z nivojem $nivo(x)$, kjer $nivo : V \to \{-1, ..., w\}$, ter prioriteto $prioriteta(x)$, kjer $prioriteta : V \to \{0, ..., n-1\}$.

Opisali bomo, kako zgraditi hierarhijo komponent $T = (V_T, E_T)$. Začnemo z množico vozlišč V grafa G in jih vključimo v V_T, kot vozlišča s pripadajočim nivojem $nivo(v) = -1$. V tem koraku ne dodamo nobenih povezav. Vsa vozlišča z $nivo(v) = -1$ so torej vozlišča, ki so vsebovana tako v drevesu T kot tudi v grafu G. Sedaj izvedemo w faz algoritma, ki gradi hierarhijo komponent. Naj bo $j = 0, ..., w$ indeks posamezne faze in v fazi j poiščemo krepko povezane komponente v grafu G_j. Nato izvedemo skrčitev vsake netrivialne[4] krepko povezane komponente v nova vozlišča. Za vsako komponento x hranimo tudi množico M_x, ki vsebuje vozlišča $v \in V$, katera so vsebovana v komponenti x. Za vsa enostavna vozlišča, tj. $t \in V$, bomo zahtevali $M_t = \{t\}$. Vsako novo vozlišče x, ki smo ga dobili s skrčitvijo, dodamo v V_T in definiramo $nivo(x) = j$. Vsa vozlišča $y \in V_T$, za katere velja $nivo(y) < nivo(x)$, ki še nimajo staršev in za katere velja $M_x \cap M_y \neq \emptyset$, povežemo v drevesu T tako, da je x starš od y. V primeru, da ima ob koncu faze j neko novo vozlišče x samo enega otroka y, odstranimo vozlišče x iz V_T, ker mora v tem primeru veljati $M_x = M_y$ in gre za dve enaki komponenti x in y. Če v nekem grafu G_i obstaja pot od nekega vozlišča u do nekega vozlišča v, ki sta del različnih krepko povezanih komponent $x, y \in V_T$ tako, da $u \in M_x$ ter $v \in M_y$, bomo potem zahtevali $prioriteta(x) < prioriteta(y)$. Očitno je, da bosta v trenutku, ko obstaja pot tudi v drugo smer, vozlišči u in v del iste krepko povezane komponente v tisti in v vseh naslednjih fazah gradnje.

Drevo, ki ga dobimo, imenujemo *hierarhija komponent*, saj je vsako vozlišče $v \in V_T$: $nivo(v) \neq -1$ krepko povezana komponenta grafa $G_{nivo(v)}$. Namerno se nismo spustili v algoritmične podrobnosti postopka gradnje hierarhije. Sprva se bomo osredotočili na uporabo hierarhije za izračun najkrajših poti. Kasneje pa bomo podali učinkovitejši in

[4]Kot trivialne krepko povezane komponente smatramo take komponente, ki so sestavljene iz enega samega vozlišča.

natančnejši postopek gradnje ter ga ustrezno analizirali iz vidika časovne in prostorske zahtevnosti.

Iz same definicije hierarhije lahko razberemo naslednje lastnosti:

1. Vsi listi v drevesa T predstavljajo vozlišča grafa G, tj. $v \in V_T \cap V$.
2. Vsako notranje vozlišče drevesa T, tj. $\forall x \in V_T : nivo(x) > -1$, ima vsaj dva otroka.
3. Če imamo dve vozlišči $x, y \in V_T$ in je v drevesu T x starš od y, potem $nivo(x) > nivo(y)$.
4. Če imata vozlišči $u, v \in V$ v drevesu T skupnega prednika $x \in V_T$, potem obstaja pot iz u do v, in obratno, v $G_{nivo(x)}$.
5. Če sta vozlišči $u, v \in V$ naslednika dveh različnih vozlišč $y, z \in V_T$ v drevesu T tako, da $u \in M_y$ in $v \in M_z$. Potem, naj bo $x \in V_T$ skupni prednik vozlišč y in z. Potem bodisi velja $prioriteta(y) < prioriteta(z)$, ali pa ne obstaja pot iz u do v v $G_{nivo(x)-1}$.

3.11.2 Obiskovanje vozlišč

Hierarhijo, kot smo jo definirali, lahko sedaj uporabimo pri obiskovanju vozlišč. Držali se bomo pogoja, da obiščemo vozlišče v samo takrat, kadar $D(v) = d(v)$, tako, kot pri Dijkstrovem algoritmu. Za razliko od Dijkstrovega algoritma, ne bomo nujno obiskovali vozlišč v naraščajočem vrstnem redu. Uporabili bomo idejo, ki smo jo pred gradnjo hierarhije že omenili in sicer je to povečevanje spodnjih ocen $\delta(v)$ tako, da le-te upoštevajo trikotniško neenakost. Algoritem bomo opisali kot algoritem, ki rešuje problem SSSP in iz tega razloga bomo vozlišče s, ki predstavlja vir, pri zapisu izpustili. Vsem listom v v drevesu T bomo dodelili zgornje ocene $D(v)$, ki predstavljajo neko razdaljo od izvora s do vozlišča v, vendar le-ta ni nujno najkrajša razdalja. Kot pri Dijkstrovem algoritmu, bomo tudi tukaj ob sproščanju povezav zgornje ocene nižali. Sprva bodo vse zgornje ocene $\forall v \in V : D(v) = \infty$ razen $\forall (s, u) \in E : D(u) = \ell(s, u)$.

Algoritem obiskovanja bo deloval tako, da si bodo vozlišča v drevesu izmenjevala sporočila v obliki intervalov. Intervali bodo potovali od korena drevesa proti listom. Vsako vozlišče $x \in V_T : nivo(x) \neq -1$ bo poskrbelo, da povečevanje spodnjih ocen na podlagi poslanih intervalov ne bo kršilo trikotniške neenakosti. Ko bo nek interval I prispel do nekega neobiskanega vozlišča $v \in V_T : nivo(v) = -1$, bo vozlišče v obiskano, če in samo, če $D(v) \in I$. Seveda, ko neko vozlišče obiščemo, izvedemo tudi sproščanje, kar pomeni, da ob obisku (lahko) znižamo vrednost $D(u)$ nekega lista u v T.

Sedaj si poglejmo proces bolj podrobno. Kot smo že omenili, bodo vozlišča v drevesu T prejela od starša nek interval I, ki predstavlja »dovoljenje«, da se spodnja ocena premakne prek I. V kolikor vozlišče x, kjer $i = nivo(x)$ in $nivo(x) \neq -1$, prejme nek interval $I = [L, R)$, potem x razbije interval I na več intervalov: $I_1 = [L, L + 2^{i-1}), I_2 = [L + 2^{i-1}, L + 2 \cdot 2^{i-1}), ..., I_k = [L + (k-1) \cdot 2^{i-1}, R)$. Za vsak $j = 1, ..., k$ nato izvede slednje: sprehodi se skozi svoj seznam otrok po naraščajočih prioritetah in za vsakega otroka y pošlje interval I_j vozlišču y, počaka dokler y ne konča dela, nato pa stopi do naslednjega otroka, ali pa do naslednje vrednosti j. Ko zadnji otrok vozlišča x konča delo in smo poslali vse intervale, javi vozlišče x svojemu staršu,

Algoritem 7 Preprost postopek obiskovanja vozlišč z uporabo hierarhije.

```
procedure Obišči(v, [L,R))
    if nivo(v) = −1 then                          ▷ ustavitveni pogoj rekurzije
        if D(v) ∈ [L, R) ∧ ¬v.obiskano then
            d(v) := D(v)
            v.obiskano := 1
            for all (v, u) ∈ E do
                D(u) := min(D(u), d(v) + ℓ(v, u))
            end for
        end if
        return
    end if
    i := nivo(v)
    I_1, I_2, ..., I_k := [L, L + 2^{i−1}), [L + 2^{i−1}, L + 2 · 2^{i−1}), ..., [L + (k − 1) · 2^{i−1}, R)
    for j := 1 to k do
        for all c ∈ v.otroci do          ▷ v.otroci urejeni naraščajoče po prioriteta(·)
            Obišči(c, I_j)
        end for
    end for
end procedure
```

da je končalo delo. Pojem »končati delo« se nanaša na ustavitveni pogoj rekurzije, ki je razviden iz psevdokode v algoritmu 7.

Koren drevesa deluje na podoben način, vendar ne prejme nobenih intervalov razen začetnega, $[0, \infty)$. Dejansko lahko kličemo *Obišči*$(koren, [0, \infty))$. Ko se izvajanje konča imamo rešitev problema SSSP v obliki vrednosti $\forall v \in V : d(v)$. Naslednji korak je dokazati, da je algoritem pravilen. Najprej bomo dokazali, da algoritem upošteva trikotniško neenakost.

Lema 3.13. *Algoritem 7 upošteva pogoje leme 2.4 (Povečevanje spodnjih ocen).*

Dokaz. Dokazati moramo, da $\forall (u, v) \in E : \delta(u) + \ell(u, v) \geq \delta(v)$ ter, da algoritem sprosti izhodne povezave nekega vozlišča $v \in V$ kadar $\delta(v) = D(v)$. Pogoj sproščanja sledi iz algoritma, saj vozlišče obiščemo natanko takrat kadar $\delta(v) = D(v)$. Dokazati moramo torej, da drži trikotniška neenakost spodnjih ocen. Vzeli bomo neko poljubno povezavo $(u, v) \in E$ in zanjo dokazali, da neenakost velja. Naj sta u, v dva lista v drevesu T. Naj sta oba vsebovana v dveh različnih vozliščih y in z drevesa T tako, da $u \in M_y$ ter $v \in M_z$ ter naj bosta y in z otroka vozlišča x v drevesu T. To je enak pogoj, kot pri 5. lastnosti hierarhije.

Sedaj imamo dve možnosti: bodisi velja $\ell(u, v) < 2^{nivo(x)-1}$, ali pa $\ell(u, v) \geq 2^{nivo(x)-1}$. Najprej si ogledamo prvo možnost. Vozlišči u in v sta prvič združeni v krepko povezano komponento x šele v nivoju $nivo(x)$. To pomeni, da ne obstaja pot tako iz u do v ter v do u, ki bi uporabljala samo povezave dolžin največ $2^{nivo(x)-1}$, saj bi v tem primeru bila vozlišča združena v neki komponenti k, kjer $nivo(k) = nivo(x) - 1$. Vendar smo predpostavili, da obstaja pot iz u do v, ki je dolžine manjše kot $2^{nivo(x)-1}$. Iz definicije hierarhije vemo, da mora potem veljati $prioriteta(y) < prioriteta(z)$. Posledično mora v algoritmu vedno veljati $\delta(u) \geq \delta(v)$, saj bo vozlišče x vedno dalo prednost otroku y v metodi *Obišči*, kar trivialno zadostuje pogoju trikotniške neenakosti.

Druga možnost je, da $\ell(u,v) \geq 2^{nivo(x)-1}$. V tem primeru se naslonimo na dejstvo, da vozlišče x razbije vsak interval na podintervale, ki so med seboj razmaknjeni največ za $2^{nivo(x)-1}$. To pomeni, da za par vozlišč u, v, $\delta(v) > \delta(u)$ velja $\delta(v) - \delta(u) \leq 2^{nivo(x)-1}$. Vendar iz predpostavke vemo, da $\ell(u,v) \geq 2^{nivo(x)-1}$, kar pomeni, da trikotniška neenakost drži, saj: $\delta(u) + 2^{nivo(x)-1} \geq \delta(v)$. □

Lema 3.14. *Algoritem 7 izračuna dolžine najkrajših poti $d(v)$ za vsa vozlišča $v \in V$.*

Dokaz. Sledi iz dokaza leme 3.13 in leme 2.4 (Povečevanje spodnjih ocen). □

3.11.3 Boljši način obiskovanja

Podali smo enostaven način obiskovanja vozlišč, za katerega smo dokazali, da je pravilen. Kljub temu ga nismo analizirali iz vidika časovne zahtevnosti, ker je časovno preveč potraten. V nadaljevanju si bomo ogledali izboljšano različico algoritma 7.

Najprej se osredotočimo na število vozlišč v drevesu T. Vemo, da je natanko n listov. Ker smo pri definiciji same hierarhije zahtevali, da je pogoj za uvedbo novega vozlišča združitev vsaj dveh že obstoječih vozlišč, je vseh dodanih vozlišč omejeno z n. To pomeni, da je $|V_T| < 2n$. Zahtevali bomo, da veljata naslednji lastnosti:

1. Starš $x \in V_T$ naj doda otroka $y \in V_T$ v seznam prejemnikov intervalov šele takrat, kadar za nek interval I in naslednika $v \in V$ otroka y velja $D(v) \in I$.

2. Starš x naj odstrani otroka y iz seznama prejemnikov intervalov takrat, kadar so bili obiskani vsi nasledniki v otroka y.

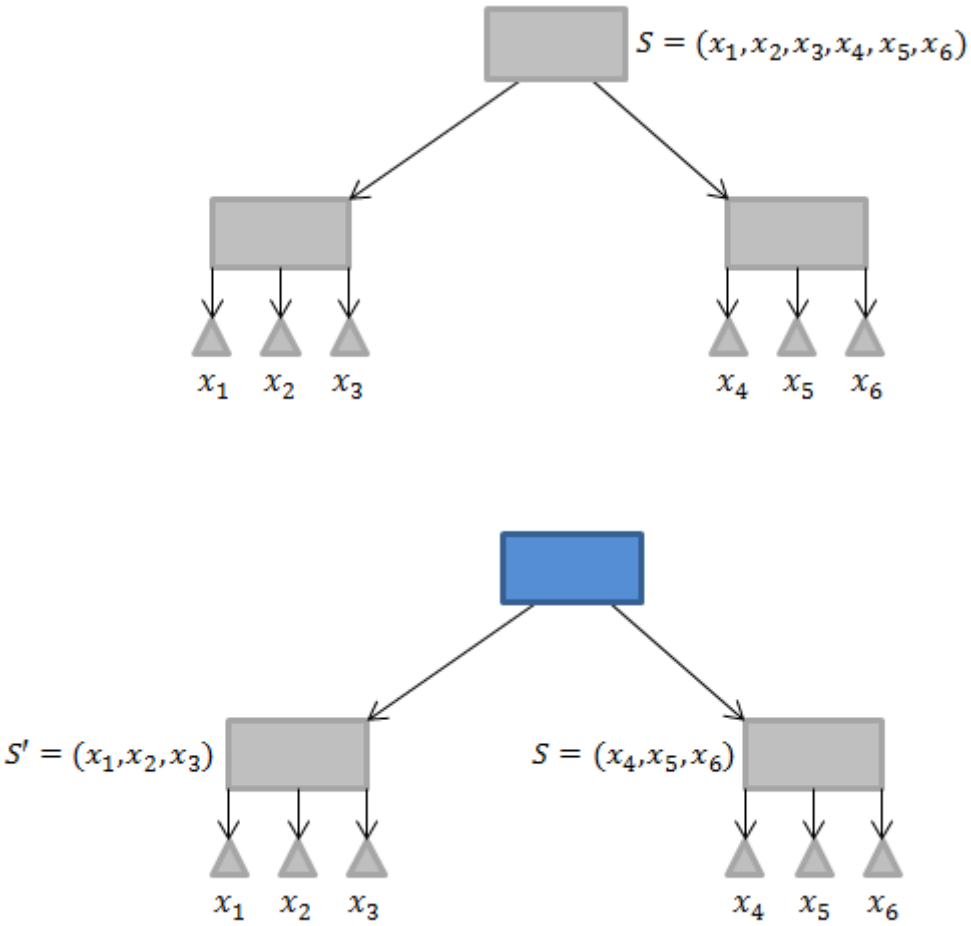
Če zgoraj našteti zahtevi držita, potem lahko omejimo število poslanih intervalov v teku izvajanja algoritma na $O(m)$, kot to prikazuje lema 3.15.

Lema 3.15. *Če veljata zahtevi 1. in 2. potem je skupno število poslanih intervalov v času izvajanja algoritma omejeno z $O(m)$.*

Dokaz. Zaradi 1. zahteve bo prvi interval, ki ga prejme vozlišče $x \in V_T$ omogočil obisk nekega naslednika (lista) $v \in V$. V grafu G_i, kjer $i = nivo(x)$, obstaja najdaljša najkrajša pot od vozlišča v do nekega vozlišča w. Ta pot ima dolžino največ $DIAM(G_i)$, po definiciji premera grafa v poglavju 2. Število intervalov, ki jih bo prejelo vozlišče x je torej omejeno z $1 + \frac{DIAM(G_i)}{2^i}$, saj kadar vozlišče x prejme tak interval I, da $DIAM(G_i) \in I$ nato bodo obiskani vsi nasledniki vozlišča x. Posledično zaradi 2. zahteve vozlišče x ne bo prejelo dodatnih intervalov, saj ga bo njegov starš v drevesu T odstranil iz seznama prejemnikov.

Vsaka povezava $e \in E$ doprinese k razmerju $\frac{DIAM(G_i)}{2^i}$, kjer $i = nivo(x) = nivo(e)$ največ $2^0 = 1$. V nivoju $i = nivo(e) + 1$ doprinese ta ista povezava k razmerju največ 2^{-1}. Torej, splošno lahko zapišemo, da vsaka povezava $e \in E$ doprinese skupno največ $\sum_{k=0}^{\infty} 2^{-k} = 2$ k razmerjem vseh vozlišč drevesa T v katera je vključena. Oziroma drugače, lahko rečemo, da vsaka povezava $e \in E$ »plača« pošiljanje dveh intervalov. Število poslanih intervalov je torej: $\sum_{v \in V_T} 1 + \frac{DIAM(G_{nivo(v)})}{2^{nivo(v)}}$. Ker vemo, da $|V_T| < 2n$ in, da $\sum_{v \in V_T} \frac{DIAM(G_{nivo(v)})}{2^{nivo(v)}} = 2m$, je število vseh intervalov omejeno z $O(2n + 2m) = O(m)$. □

Omejili smo število poslanih intervalov, vendar moramo sedaj dodatno poskrbeti, da zahtevi 1 in 2 držita. Drugo zahtevo je preprosto zadovoljiti:



Slika 3.5: Primer izvedbe split ob obisku vozlišča (zgoraj: pred obiskom, spodaj: po obisku) tako, da se zaporedje razdeli na podzaporedja. Pravokotniki predstavljajo skrčene krepko povezane komponente, trikotniki pa vozlišča grafa G. Z modro obarvani liki so že prejeli interval.

1. V primeru, da obiščemo nek list, ga odstranimo iz seznama naših otrok.

2. Kadar smo odstranili vse liste iz seznamov otrok, sporočimo našemu staršu, da naj nas odstrani iz svojega seznama otrok.

Iz seznama lahko odstranimo vozlišče v času $O(1)$. Ker je vsako vozlišče v drevesu odstranjeno samo enkrat, in je vseh vozlišč $O(n)$ za to porabimo skupno $O(n)$ časa.

Zadovoljitev 1. zahteve pomeni, da bo moral vsak starš poskrbeti, da ima na voljo vrednosti $D(v)$ za otroke, ki še niso prejeli nobenega intervala. V trenutku, ko prejmejo prvi interval, nam tega ni potrebno več beležiti. Vrednosti $D(v)$ se s tekom izvajanja algoritma zmanjšujejo ob sproščanju, vendar je sproščanj največ $O(m)$. Prvo zahtevo se lahko zadovolji v času $O(m\alpha(m, n))$ tako, da uporabimo podatovno strukturo Split-findmin na naslednji način: sprehodimo se po listih hierarhije T od leve proti desni in tvorimo začetno zaporedje S, ki sprva pripada korenu hierarhije. Ko neko vozlišče prejme interval, izvedemo split tako, da razdelimo začetno zaporedje na podzaporedja. Podzaporedje, ki pripada nekemu otroku, vsebuje vse njegove liste. Postopek je prikazan na sliki 3.5.

Kljub temu, da je sedaj število intervalov omejeno z $O(m)$, je čas algoritma samega

še vedno prevelik: za vsak prejeti interval se mora namreč starš sprehoditi po celotnem seznamu otrok. Rešitev je, da hranimo v trenutnem seznamu poleg otrok, ki so že prejeli nek interval, samo tiste otroke, za katere velja, da bodo v tem trenutku prvič prejeli nek interval I. Ker je potrebno do otrok dostopati po naraščajočem vrstnem redu glede na prioriteto, ni mogoče enostavno odstraniti in dodati otroka v času $O(1)$, saj je položaj otroka odvisen od trenutno prisotnih otrok. Ker je prioriteta celoštevilčna vrednost od 0 do $n-1$, hranimo otroke v van Emde Boas drevesu [27]. To nam omogoča, da najdemo naslednjega trenutno vsebovanega otroka v času $O(\lg\lg n)$ ter dodamo in odstranimo otroka v istem času. Ker je tovrstnih operacij $O(n)$, je čas potreben skupno $O(n\lg\lg n)$, kar pomeni, da je za rešitev enega problema SSSP potrebnega $O(n\lg\lg n+m)$ časa, in za rešitev problema APSP potem $O(n^2\lg\lg n+nm)$ časa. Naj še omenimo, da lahko v primeru posebne pomnilniške arhitekture *Yggdrasil* uporabimo učinkovitejšo podatkovno strukturo, ki tovrstne operacije izvaja v času $O(1)$ [2]. Pri analizi izvajalnega časa celotnega algoritma bo potrebno tudi upoštevati čas, ki ga bomo potrebovali za izgradnjo hierarhije.

3.11.4 Izboljšan postopek gradnje hierarhije

Čas, ki ga algoritem porabi za reševanje problema APSP ob že izgrajeni hierarhiji, je omejen z $O(n^2\lg\lg n+mn)$. Lahko si torej privoščimo gradnjo hierarhije v času $O(mn)$. Opisali smo že postopek gradnje hierarhije s časom $O(mw)$, vendar je dolžina besede w neodvisna od problema. Tukaj si ogledamo drugačen postopek gradnje ob predpostavki $w>n$, saj je v nasprotnem primeru časovna zahtevnost prej-opisanega postopka gradnje $O(mw)=O(mn)$.

Algoritem bo hranil matriko dosegljivosti H velikosti $n\times n$, kjer ima celica matrike vrednost $(u,v)=1$, če obstaja pot iz u do v ter 0 sicer. Ker je velikost besede $w>n$, lahko hranimo vrstice in stolpce matrike v obliki nizov ničel in enk kot eno samo besedo na vrstico. Sprva ne bo nobene povezave v našem grafu. Matrika H ima torej povsod same ničle. Naj H_v predstavlja v-to vrstico matrike H v obliki bitnega vektorja ter $H_{v,k}$ naj predstavlja k-ti bit bitnega vektorja H_v. Vse kar je potrebno storiti (za posodobitev matrike) ob dodatku neke povezave (u,v), je nastaviti $H_{u,v}=1$ ter nato izvesti logično *ali* operacijo: $H_u\vee H_v$ ter za vse vrstice H_w za katere velja $H_{w,u}=1$ izvesti $H_w\vee H_u$. Vsaka operacija porabi $O(1)$ časa, saj je vrstica shranjena kot beseda. Ker je vseh vrstic n, porabi algoritem $O(n)$ časa za vsako novo povezavo.

Sedaj uporabimo zgoraj omenjeni algoritem pri gradnji hierarhije. Začnemo z vsemi vozlišči iz V ter brez povezav. Nato dodajamo povezave iz E v naraščajočem vrstnem redu glede na dolžino. Ob vsakem dodatku posodobimo matriko H. Ko dodamo vse povezave nekega nivoja, izračunamo krepko povezane komponente grafa in izvedemo kontrakcijo komponent. Ob dani matriki H je enostavno poiskati krepko povezane komponente: izberemo neko poljubno vozlišče v ter za vsako vozlišče u, da velja $H_{v,u}=1$ preverimo če velja $H_{u,v}=1$. Če velja, potem sta u in v del iste krepko povezane komponente. Ponavljamo za ostala vozlišča. Komponente lahko torej poiščemo v času $O(n)$ na komponento, vendar je sedaj še potrebno izvesti kontrakcijo. Vsako komponento $C=\{v_0,v_1,...,v_k\}$ sestavljeno iz k-tih vozlišč združimo v eno samo vozlišče tako, da izbrišemo vse bitne vektorje vozlišč $v\in C$, razen prvega vozlišča v_0 v matriki, ki sedaj predstavlja celotno komponento. Komponento nato dodamo v drevo T, ki predstavlja hierarhijo.

Ker je vseh povezav m in porabimo $O(n)$ na povezavo za posodobitev matrike H,

to znese $O(mn)$ časa. Pri skrčitvi komponent je čas $O(n)$ na komponento, kar skozi celoten algoritem znese $O(mn)$ časa.

4 NOVI ALGORITMI

V tem poglavju predstavimo nove algoritme za reševanje problema APSP in problema, ki mu je soroden.

4.1 NAJKRAJŠE POTI MED NEKATERIMI PARI VOZLIŠČ

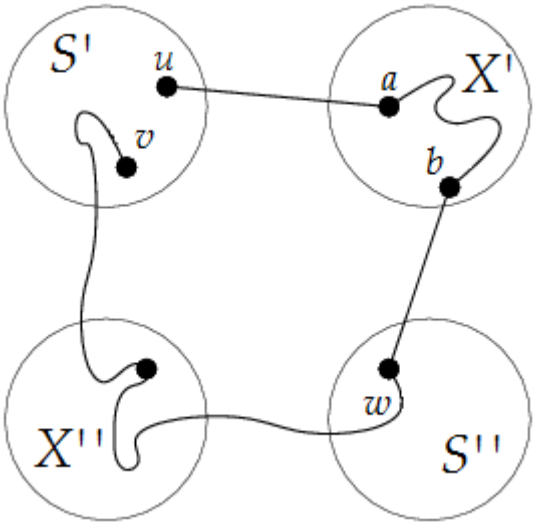
Algoritem rešuje problem, ki je nekoliko podoben problemu APSP. Tako kot pri problemu APSP, kjer nas zanimajo najkrajše poti med vsakim parom vozlišč, lahko definiramo problem, kjer nas zanimajo najkrajše poti samo med nekaterimi pari vozlišč. Problem, ki ga rešuje Algoritem A, definira za nek poljuben graf $G = (V, E)$ množici $S \subset V$ ter $X \subset V$, kjer velja $S \cup X = V$ ter $S \cap X = \emptyset$. Zanimajo nas najkrajše poti med vsemi pari vozlišč iz množice S. Pri tem dovolimo, da gre lahko neka pot iz $s \in S$ v $s' \in S$ preko vozlišč iz množice X.

Omenjen problem se pojavi pri eni izmed variacij problema trgovskega potnika, ki se imenuje problem potujočega obiskovalca (angl. *travelling visitor problem, TVP*) [6].

Algoritem 8 Najkrajše poti med nekaterimi pari vozlišč.

```
1:  procedure NekateriPari(S,X,W)
2:      Floyd-Warshall(X, W)
3:      for all k ∈ X do
4:          for all i ∈ X do
5:              for all j ∈ S do
6:                  W_ij := min(W_ij, W_ik + W_kj)
7:              end for
8:          end for
9:      end for
10:     for all k ∈ X do
11:         for all i ∈ S do
12:             for all j ∈ S do
13:                 W_ij := min(W_ij, W_ik + W_kj)
14:             end for
15:         end for
16:     end for
17:     Floyd-Warshall(S, W)
18: end procedure
```

Lema 4.1. *Naj bo $G = (X \cup S, E)$ graf, kjer $X \cap S = \emptyset$, $x = |X|$ in $s = |S|$. Definiramo S-pot kot zaporedje povezav iz E, ki se začne in konča v S ter lahko obišče vozlišča iz*



Slika 4.1: Vozlišča predstavljajo povezane komponente, ki sestojijo v celoti bodisi iz vozlišč iz X ali iz vozlišč iz S. Povezave predstavljajo poljubno mnogo povezav med komponentami.

X. Obstaja algoritem, ki izračuna najkrajše S-poti med vsemi pari vozlišč $u, v \in S$ v času $s^3 + x^3 + s^2x + x^2s$.

Dokaz. Graf G razdelimo na več komponent, kjer vsaka komponenta vsebuje bodisi vozlišča iz S ali iz X. Slika 4.1 prikazuje primer delitve S na komponenti S' in S'' in delitev X na komponenti X' in X'', kjer velja $S = S' \cup S''$, $X = X' \cup X''$, ter $S' \cap S'' = \emptyset$, $X' \cap X'' = \emptyset$. Pri dokazu bomo uporabili primer iz slike 4.1. V splošnem lahko sicer S in X razpadeta na več komponent, vendar bo enostavno videti, da večje število komponent ne vpliva na pravilnost dokaza in delovanje algoritma. Naj bo π_{uv} pot med vozliščema $u, v \in S$. Pot se prične in konča v S, vendar gre lahko skozi X, ali skače med X in S. Naj bo vozlišče $w \in S$ tako vozlišče, da pot π_{uv} prvič ponovno vstopi v S pri vozlišču w. Zaradi optimalnosti podpoti, sta poti π_{uw} in π_{wv} tudi najkrajši poti med u in w ter w in v. Torej, če najprej izračunamo te poti, bo sproščanje v Floyd-Warshall algoritmu našlo pot π_{uv}.

Naj bo $\pi_{uw} = u \to \pi_{ab} \to w$, tj. pot π_{uw} vstopi v X pri a in izstopi pri b. Tako kot prej, je tudi π_{ab} najkrajša pot med $a, b \in X$, ki pa je v celoti v X. Podobno, poti $u \to \pi_{ab}$ in $\pi_{ab} \to w$ sta tudi najkrajši poti. Torej, če smo izračunali omenjene poti, bo sproščanje našlo najkrajšo pot π_{uw}.

Sedaj si ogledamo algoritem 8:

- Najprej izračunamo najkrajše poti med vsemi pari vozlišč iz množice X (glej π_{ab}), za kar potrebujemo x^3 časa.
- Potem (vrstice 3-9) izračunamo najkrajše poti med vsemi vozlišči $a \in X$ in $w \in S$ (glej $\pi_{ab} \to w$), za kar potrebujemo x^2s časa.
- V vrsticah 10-16 izračunamo najkrajše poti med vsemi vozlišči $u \in S$ in $b \in X$ (glej $u \to \pi_{ab}$), za kar potrebujemo xs^2 časa.
- Zadnji klic algoritma Floyd-Warshall na množici S izračuna najkrajše poti med

vsemi pari vozlišč v S, upoštevajoč pod-poti, ki smo jih izračunali zgoraj. Za to porabimo s^3 časa.

Seštejemo in dobimo, da algoritem porabi v celoti $x^3 + x^2s + xs^2 + s^3$ časa, kar ima ekstrem v $4s^3$ pri $x = s$. V primeru, da bi ta isti problem rešili z uporabo Floyd-Warshall algoritma, bi pri zgornjem ekstremu ta porabil $(x + s)^3 = 8s^3$. □

4.2 NAJKRAJŠE POTI MED VSEMI PARI VOZLIŠČ

V poglavju 3 smo si ogledali nekaj algoritmov, ki rešujejo problem SSSP na grafu $G = (V, E)$. Omenjeni algoritmi rešujejo tudi problem APSP tako, da jih enostavno izvedemo za vsak izvor $v \in V$. Denimo, da imamo na voljo SSSP algoritem ψ, ki rešuje problem na nenegativno uteženem grafu. Naj bo asimptotična časovna zahtevnost algoritma ψ enaka $O(T_\psi(m, n))$ in prostorska zahtevnost enaka $O(S_\psi(m, n))$. Očitno je, da bomo potrebovali n klicev algoritma ψ za reševanje APSP, kar znaša skupno $O(nT_\psi(m, n))$ časa. Poleg prostora, ki ga potrebuje algoritem ψ, bo potrebno še $\Theta(n^2)$ prostora za hrambo matrike razdalj, kar znaša skupno $O(S_\psi(m, n) + n^2)$ prostora. V tem razdelku se ukvarjamo z vprašanjem: »Ali je to najboljše kar se da?«

Ideja pohitritve SSSP algoritmov, kadar se uporabljajo kot reševalniki za APSP, ni nova. Obstaja algoritem z imenom *Hidden Paths Algorithm* [17], ki izboljša časovno zahtevnost Dijkstrovega algoritma z $O(mn+n^2 \lg n)$ na $O(m^*n+n^2 \lg n)$, kjer $m^* \leq m$ predstavlja število povezav $(u, v) \in E$, ki so uporabljene v najkrajših poteh. To storijo tako, da Dijkstrov algoritem poženejo iz več izvorov postopoma: najprej poiščejo eno najkrajšo pot za vsako vozlišče, nato naslednjo, itd. V algoritmu izrabijo dejstvo, da vozlišče v lahko ignorira povezavo (u, w) dokler neko sosednje vozlišče u ne uporabi povezave (u, w) kot najkrajšo pot iz u do w, kar pa pomeni, da je (u, w) del najkrajših poti. Slabost omenjenega algoritma je ta, da gre za izrecno modifikacijo Dijkstrovega algoritma in ne deluje za nek poljuben algoritem ψ.

V tem razdelku podamo nov, splošen način uporabe (poljubnega) algoritma ψ. Nova časovna zahtevnost je $O(m^*n + m \lg n + nT_\psi(m^*, n))$. Naša izboljšava, kadar je ψ Dijkstrov algoritem, je asimptotično enaka omenjenemu algoritmu. Razlika je predvsem ta, da je naš pristop splošen in uporaben za vsak algoritem ψ. Ena izmed zanimivih posledic je asimptotično hitrejši (v meri m^*) algoritem za problem APSP v poljubno-uteženih acikličnih grafih, ki deluje v času $O(m^*n + m \lg n)$. Poleg učinkovitosti je naš algoritem zanimiv tudi iz inženirskega stališča, saj je algoritem agnostičen glede implementacije ψ, kar olajša integracijo v algoritemske knjižnice.

Poleg novih algoritmov, podamo še zanimivo povezavo med časovno zahtevnostjo problema iskanja najkrajših poti iz enega izvora in problemom urejanja najkrajših poti med vsemi pari vozlišč.

4.2.1 Definicije

V nadaljevanju bomo predpostavili, da nas ne zanimajo najkrajše poti, ki pričnejo in končajo v vozlišču $s \in V$. V kolikor iščemo take poti, jih lahko enostavno najdemo v času $O(n^2)$, če imamo dane ostale poti. Spomnimo, da v Dijkstrovem algoritmu poiščemo najkrajše poti v naraščajočem vrstnem redu glede na oddaljenost od izvora. Na podoben način bomo tukaj definirali urejene sezname najkrajših poti. Za pot π pravimo, da je k-ta najkrajša pot vozlišča v, če je vsebovana na k-tem mestu seznama

najkrajših poti vozlišča v. Do sedaj smo pri SSSP algoritmih iskali najkrajše poti iz enega izvora do vseh ciljev. Za lažjo razlago bomo v tem razdelku obrnili notacijo in iskali najkrajše poti iz enega cilja do vseh izvorov. V kolikor želimo obratni izpis, lahko obrnemo vse povezave v grafu brez izgube pravilnosti. Kadar pišemo k-ta najkrajša pot vozlišča v torej mislimo na k-to najkrajšo vhodno pot v vozlišče v.

Definicija 4.2. (Urejen seznam najkrajših poti P_v) *Naj bo $P_v = \{\pi_1, \pi_2, ..., \pi_{n-1}\}$ seznam najkrajših vhodnih poti v neko vozlišče $v \in V$. Potem, naj $P_{v,k}$ predstavlja k-ti element seznama P_v. Seznam najkrajših poti P_v je urejen glede na dolžino poti, torej velja:*

$$\forall i, j : 0 < i < j < n \Rightarrow \ell(\pi_i) \leq \ell(\pi_j).$$

Ključnega pomena za naš algoritem bo naslednji izrek o najkrajših poteh.

Izrek 4.3. *Da lahko določimo $P_{v,k}$ potrebujemo povezave $\{(u, v) \in E \mid \forall u \in V\}$ in prvih k elementov iz vsakega seznama P_u, kjer $(u, v) \in E$.*

Dokaz. Denimo, da smo našli k najkrajših poti v vsa sosednja vozlišča v-ja in sedaj iščemo k-to najkrajšo pot v vozlišče v, ki jo označimo kot π_k. Sedaj imamo dve možnosti: bodisi je π_k sestavljena iz ene same povezave (u, v), pri čemer imamo že vse potrebne informacije, ali pa je π_k stik neke poti π do vozlišča u in povezave (u, v). Pokazati moramo, da je π že vsebovana v $P_{u,i}$ kjer $i \leq k$.

Trditev bomo dokazali s protislovjem. Trdimo nasprotno, da π ni vsebovana v P_u, ali da je vsebovana na mestu $i > k$. To pomeni, da obstaja neka pot π', ki je vsebovana v P_u na mestu $i \leq k$ in za katero po definiciji velja $\ell(\pi') \leq \ell(\pi)$. Potem lahko enostavno vzamemo π_k kot stik (u, v) in π', kar je krajša pot kot stik (u, v) in π. Čeprav je pot krajša, to še ne pomeni, da gre za najkrajšo pot med izvorom $s \in V$ in ciljem v, saj lahko v že pozna krajšo pot med s in v.

Spomnimo, da P_u vsebuje vsaj k najkrajših poti, tj. poti iz k različnih izvorov s. Za razliko, seznam P_v vsebuje največ $k - 1$ najkrajših poti. Enostaven argument s štetjem pravi, da obstaja neka pot π' vsebovana v P_u z indeksom $i \leq k$, ki izvira iz nekega vozlišča s, katero ni vsebovano kot izvor poti v seznamu P_v. Torej je $\pi' \to (u, v)$ najkrajša pot med s in v, ter velja $\ell(\pi') \leq \ell(\pi)$, kar nas privede do protislovja. □

4.2.2 Algoritem

Denimo, da imamo na voljo SSSP algoritem ψ, ki ga lahko pokličemo v nekem drugem algoritmu kot $\psi(V, E, s)$, kjer V predstavlja množico vozlišč in E množico povezav, s pa predstavlja izvorno vozlišče. Predstavljeni algoritem ne predpostavlja ničesar o funkciji razdalje ℓ razen tega, da je nenegativna. Če ima algoritem ψ posebne predpostavke, kot npr. posebno obliko funkcije razdalje, potem je ta zahteva implicitno prisotna tudi v našem algoritmu, saj uporablja ψ.

Najprej predstavimo enostavnejšo inačico algoritma, ki teče v času $O(mn+nT_\psi(m^*, n))$. Interakcijo z algoritmom ψ omejimo zgolj na izvajanje ψ ter branje izhoda, ki ga algoritem vrne. Da lahko izboljšamo čas algoritma, konstruiramo graf $G = (V', E')$, nad katerim pokličemo ψ. V našem algoritmu se izmenično izvajata dva procesa: reševalnik APSP, ki deluje na grafu G ter SSSP algoritem ψ, ki deluje na grafu G'. Naj bo $n' = |V'|$ in $m' = |E'|$. Skozi življenjsko dobo algoritma bomo zagotovili, da vedno velja $m' \leq m^* + n$ in $n' = n + 1$. Algoritem ima $n - 1$ faz, vsaka faza pa je sestavljena iz

treh korakov: (1) pripravi graf G'; (2) poženi ψ na grafu G'; in (3) interpretiraj izhod algoritma ψ.

Algoritem dejansko deluje na $n-1$ različnih grafih, vendar so si grafi med seboj zelo podobni. Algoritem bomo opisali z vidika enega grafa G' z zmožnostjo spreminjanja uteži povezav in uvedbo novih povezav v G'. Sprva definiramo $V' = V \cup \{i\}$, kjer je i neko novo vozlišče, ki ni vsebovano v G. Ustvarimo n povezav iz vozlišča i do vseh vozlišč $v \in V$, tj. $E' = \bigcup_{v \in V} \{(i, v)\}$. Dolžine novih povezav sprva nastavimo na neko poljubno vrednost.

Definicija 4.4. (Seznam najkrajših poti) *Seznam najkrajših poti za neko vozlišče $v \in V$ pišemo kot S_v. Dolžina seznama je največ $n+1$ in vsebuje pare oblike (a, b) kjer $a \in V \cup \{null\}$ ter $b \in \mathbb{R}^+$. Prvi element seznama je vedno $(v, 0)$, zadnji element pa vedno $(null, \infty)$. Za vse vmesne elemente (a, b) velja $b = d(a, v)$. Seznam s $k \leq n-1$ vmesnimi elementi je torej:*

$$S_v = \Big((v, 0), (a_1, b_1), (a_2, b_2), ..., (a_k, b_k), (null, \infty)\Big).$$

Sledi opis podatkovnih struktur. Vsako vozlišče $v \in V$ hrani seznam najkrajših poti S_v (definicija 4.4), ki sprva vsebuje samo para $(v, 0)$ in $(null, \infty)$. Poleg seznama, vozlišče hrani še kazalec $p[u, v]$ za vsako vhodno povezavo oblike $(u, v) \in E$. Kazalec kaže na nek element v seznamu S_u (na začetku je to prvi element).

Definicija 4.5. (Sprejemljiv par za vozlišče v) *Nek par (a, b) je sprejemljiv za vozlišče v, če velja: $\forall (a', b') \in S_v : a \neq a'$.*

Definicija 4.6. (Trenutno najboljši par za vozlišče v, (a_v, b_v)) *Nek par $(a_v, b_v) \in S_w$, kjer $(w, v) \in E$ je trenutno najboljši par za vozlišče v, če velja, da je (a_v, b_v) sprejemljiv za v in, da: $\forall (u, v) \in E : \forall (a', b') \in S_u : (a', b')$ sprejemljiv za v in $b' + \ell(u, v) \geq b_v + \ell(w, v)$.*

Prvi korak, ki ga ubere algoritem v vsaki fazi je priprava grafa G', ki si ga v nadaljevanju podrobneje ogledamo. Najprej vsako vozlišče $v \in V$ poišče trenutno najboljši par (definicija 4.6) (a_v, b_v). Da lahko določi trenutno najboljši par, se v sprehodi po elementih, na katere kaže njegov kazalec $p[u, v]$. Za vsak kazalec $p[u, v]$ vozlišče v premika kazalec naprej po seznamu S_u, dokler ne najde sprejemljivega para. Če pridemo do konca seznama, vzamemo kot sprejemljiv par $(null, \infty)$. Temu procesu pravimo *nalaganje*.

Ko se korak nalaganja zaključi, spremenimo dolžine povezav v grafu G'. Naj bo za neko vozlišče $v \in V$ trenutno najboljši par $(a_v, b_v) \in S_w$ kjer $(w, v) \in E$, potem nastavimo $\ell(i, v) \leftarrow b_v + \ell(w, v)$. Nato pokličemo $\psi(V', E', i)$. Denimo, da SSSP algoritem vrne polje $\pi[]$ dolžine n. Naj bo vsak element $\pi[v]$ par (c, d) kjer je d dolžina najkrajše poti iz i do v v grafu G' in c je prvo vozlišče na tej poti. Pri določanju prvega vozlišča na poti od i do v izpustimo i, tj. če je pot $\pi_v = \{(i, v)\}$ potem $\pi[v].c = v$. Vključitev prvega vozlišča na poti je zgolj zaradi lažje razlage in ga, v kolikor ga SSSP algoritem ne vrne, lahko enostavno poiščemo v drevesu najkrajših poti. Vsako vozlišče $v \in V$ potem doda par $(a_{\pi[v].c}, \pi[v].d)$ na konec (pred zadnjim elementom, ki je vedno $(null, \infty)$) svojega seznama S_v. Enostavno je videti, da so povezave $(i, v) \in E'$ dejansko okrajšave za poti v G. Temu procesu pravimo *širjenje*.

Po izvedbi širjenja spremenimo graf G' na naslednji način: za vsako vozlišče $v \in V$ za katero velja $\pi[v].c = v$ preverimo, če je trenutno najboljši par $(a_v, b_v) \in S_w$, ki

je bil izbran v postopku nalaganja, prvi element seznama S_w. Če je, potem dodamo povezavo (u, v) v E'.

S tem smo zaključili opis delovanja algoritma. Opisan proces formaliziramo v psevdokodi kot algoritem 9. Algoritem izrablja izrek 4.3. Kot je bilo dokazano, je za določitev k-te najkrajše poti vozlišča v potrebnih največ k najkrajših poti sosedov vozlišča v. Korak nalaganja v algoritmu 9 najde k-to najkrajšo pot samo v primeru, kadar je za določitev le-te potrebnih največ $k-1$ sosednjih poti. V primeru, da je potrebnih k najkrajših poti sosednjih vozlišč, korak nalaganja ne najde pravilne poti. Sedaj uporabimo SSSP algoritem, ki izbere najkrajšo izmed dveh možnosti: pot, ki je tvorjena z uporabo vseh $k-1$ sosednjih poti in pot, ki je tvorjena z uporabo samo k-tih sosednjih poti. Pravilnost algoritma nam zagotovita naslednji lemi.

Algoritem 9 Najkrajše poti med vsemi pari vozlišč

```
1:  procedure APSP(V, E, ψ)
2:      V' := V ∪ {i}
3:      E' := ⋃_{∀v∈V} {(i, v)}
4:      for all v ∈ V do
5:          S_v.append( (v, 0) )
6:      end for
7:      for k := 1 to n − 1 do
8:          for all v ∈ V do                          ▷ Korak nalaganja
9:              best[v] := (null, ∞)                  ▷ Trenutno najboljši par za v
10:             for all u ∈ V s.t. (u, v) ∈ E do
11:                 while solved[v, p[u, v].a] do
12:                     p[u, v].next()                ▷ Konec seznama predstavlja sprejemljiv par
13:                 end while
14:                 if p[u, v].b + ℓ(u, v) < best[v].b then
15:                     best[v].a := p[u, v].a
16:                     best[v].b := p[u, v].b + ℓ(u, v)
17:                 end if
18:             end for
19:             ℓ(i, v) := best[v].b                  ▷ Z upoštevanjem samo k − 1 sosednjih poti
20:         end for
21:         π[] := ψ(V', E', i)
22:         for all v ∈ V do                          ▷ Interpretiramo rezultate
23:             S_v.append( (best[π[v].c].a, π[v].d) )
24:             solved[v, best[π[v].c].a] := true
25:             if π[v].c = v and best[v] was the first element of some list S_u then
26:                 E' := E' ∪ (u, v)
27:             end if
28:         end for
29:     end for
30: end procedure
```

Lema 4.7. *Za vsako vozlišče $v \in V$, katerega k-ta najkrajša pot je bila najdena v koraku nalaganja, $\psi(V', E', i)$ najde povezavo (i, v) kot najkrajšo pot do v.*

Dokaz. V primeru, da je k-ta najkrajša pot vozlišča v odvisna od neke poti na poziciji $j < k$ v seznamu najkrajših poti soseda vozlišča v, jo algoritem najde v koraku

nalaganja. Kar je potrebno pokazati je, da je najdena pot ohranjena kot najkrajša pot po izvedbi SSSP algoritma. Naj bo k-ta najkrajša pot za vozlišče $v \in V$ najdena v koraku nalaganja. Ta pot je v grafu G' predstavljena kot povezava (i, v), katere dolžina je enaka dolžini poti. Denimo, da SSSP algoritem najde neko drugo pot kot najkrajšo pot v v. Vsaka izmed izhodnih povezav vozlišča i predstavlja pot v grafu G. To pomeni, da lahko dodamo še preostale povezave, ki so potrebne da dosežemo v v G' in dobimo po dolžini enako pot v G. Torej lahko vzamemo omenjeno pot v vozlišče v in dobimo krajšo pot kot k-to najkrajšo pot v G. To je protislovje, razen, če dobljena pot ni najkrajša pot med dvema vozliščema v G (torej poznamo krajšo).

Brez izgube splošnosti predpostavimo, da je $\pi' = \{(i, u), (u, v)\}$. Ker velja $\ell(\pi') \leq \ell(i, v)$, to pomeni, da vozlišče v ni uspelo poiskati sprejemljivega para v seznamu najkrajših poti vozlišča u, sicer bi bila krajša pot izbrana v koraku nalaganja. To pa pomeni, da so vsi izvori (prvi elementi parov) vsebovani v seznamu S_u vsebovani tudi v seznamu S_v. Ker je trenutno najboljši par (a_u, b_u) po definiciji sprejemljiv par za u, je potemtakem tudi sprejemljiv par za v. To nas privede do protislovja, saj π' predstavlja najkrajšo pot v G. □

Lema 4.8. *$\psi(V', E', i)$ najde k-to najkrajšo pot v vsa vozlišča $v \in V$.*

Dokaz. Primer, kadar je k-ta najkrajša pot odvisna od največ $k - 1$ najkrajših poti sosedov, je že dokazan v lemi 4.7. Ostane nam še možnost, da je k-ta najkrajša pot v vozlišče v odvisna od k-te najkrajše poti v neko sosednje vozlišče u. V tem primeru velja, da je k-ta najkrajša pot v vozlišče v enaka k-ti najkrajši poti v u z vključitvijo dodatne povezave (u, v). Enak argument je moč uporabiti za odvisnost vozlišča u od nekega njegovega soseda. Ob vsaki taki odvisnosti se pot skrajša za eno povezavo in nas argument eventuelno privede do poti vsebovane na mestu $j < k$ v seznamu najkrajših poti (to je vključno s potmi, ki so sestavljene iz ene same povezave) nekega soseda vozlišča w. Vendar je vozlišče w tako pot že odkrilo v koraku nalaganja in je ta pot ohranjena kot najkrajša pot v w zaradi leme 4.7.

Sedaj sledimo enakemu argumentu kot v lemi 4.7 s katerim smo prišli do protislovja. V tem primeru ne gre za protislovje, saj je ravno to, kar želimo dokazati. Kar sledi je, da vsaka pot π od vozlišča i do vozlišča v v grafu G', ki izvira v vozlišču $w \in V$ v grafu G in zanjo velja $\ell(\pi) < \ell(i, v)$, nujno predstavlja sprejemljiv par $(w, \ell(\pi))$ za v. Enostavno je potem videti, da je najkrajša izmed takih poti, ki vodijo iz vozlišča i v vozlišče v najkrajša pot med i in v v G' ter tudi k-ta najkrajša pot v vozlišče v v grafu G. □

4.2.3 Časovna in prostorska zahtevnost

Najprej si ogledamo časovno zahtevnost algoritma 9. Glavna zanka algoritma (vrstice 7-29) se izvede $n-1$ krat. Zanka nalaganja (vrstice 8-20) se sprehodi po vseh povezavah $(u, v) \in E$, za kar potrebuje m korakov. Skupno to znaša $O(mn)$ časa. Seznami najkrajših poti so dolžine n, kar pomeni, da se vsak kazalec premakne naprej $O(n)$ krat skozi življenjsko dobo algoritma. Skupno je m kazalcev, kar pomeni, da čas za premike znaša $O(mn)$. Algoritem ψ je pognan $n - 1$ krat. Skupen čas algoritma 9 je potem $O(mn + nT_\psi(m^*, n))$.

Sedaj si ogledamo še prostorsko zahtevnost algoritma 9. Vsako vozlišče hrani svoj seznam najkrajših poti, ki je velikosti $n + 1$, kar znaša $\Theta(n^2)$, če štejemo vsa vozlišča. V skupnem je natanko m kazalcev, torej je potreben prostor $\Theta(m)$. Poleg

omenjenega prostora potrebujemo še toliko prostora, kolikor ga potrebuje algoritem ψ. Seštejemo prostorske zahteve in dobimo, da ima algoritem 9 prostorsko zahtevnost $O(n^2 + S_\psi(m^*, n))$.

4.2.4 Posledice

V nadaljevanju bomo pokazali, kako lahko časovno zahtevnost algoritma 9 še dodatno izboljšamo. Preden se lotimo izboljšave, si bomo ogledali nekaj posledic algoritma v trenutni obliki.

Definicija 4.9. (Urejene najkrajše poti med vsemi pari vozlišč, *SAPSP*) *Problem SAPSP(m, n) je problem iskanja najkrajših poti med vsemi pari vozlišč v nenegativno uteženem grafu $G = (V, E)$ z m povezavami in n vozlišči, kjer je rezultat podan v obliki $P_v \mid \forall v \in V$ (glej definicijo 4.2).*

Izrek 4.10. *Naj bo T_{SSSP} časovna zahtevnost problema iskanja najkrajših poti iz enega izvora v nenegativno uteženem grafu z m povezavami in n vozlišči. Časovna zahtevnost problema SAPSP je največ $O(nT_{SSSP})$.*

Dokaz. Ob danem algoritmu ψ, ki reši problem SSSP, konstruiramo algoritem za SAPSP, ki deluje v času $O(nT_\psi)$ po zgledu algoritma 9. Seznami najkrajših poti, ki jih najde algoritem 9 predstavljajo ravno željeno obliko izpisa $P_v \mid \forall v \in V$ za problem SAPSP. □

Čeprav je izrek 4.10 zelo enostavna posledica algoritma 9, ima zanimivo interpretacijo: bodisi je urejanje rezultata enostavno, kar pomeni, da lahko pri pisanju hitrejših algoritmov za APSP sledimo Dijkstrovemu postopku obiskovanja; bodisi je urejanje bistveno težje, kar pomeni, da lahko s pomočjo tega dokažemo spodnjo mejo za SSSP. Oba odgovora bi vodila do bistvenih napredkov na področju, saj ne poznamo $O(mn)$ algoritma za APSP in ne poznamo nobenih spodnjih mej za SSSP razen trivialne $\Omega(m)$.

4.2.5 Izboljšava algoritma

Algoritem 9 ima časovno zahtevnost $O(mn+nT_\psi(m^*, n))$. Tukaj pokažemo, kako lahko časovno zahtevnost algoritma znižamo na $O(m^*n + m \lg n + nT_\psi(m^*, n))$. Najprej uredimo vhodne povezave v vozlišča $\forall v \in V : E_v = \bigcup_{(u,v)\in E}\{(u, v)\}$ po velikosti v naraščajočem vrstnem redu. Z uporabo algoritma za urejanje lahko to naredimo v času $O(m \lg n)$.

Hranili bomo samo tiste kazalce $p[u, v]$, za katere vemo, da je povezava (u, v) najkrajša pot med u in v. Poleg tega bo vsako vozlišče $v \in V$ hranilo še en dodaten kazalec $p[w, v]$, za katerega ne bomo vedeli, če je (w, v) najkrajša pot med w in v. Ker so povezave urejene po velikosti, lahko neko vozlišče v ignorira povezavo na mestu t v seznamu E_v dokler ne ve, če je povezava na mestu $t - 1$ del ene od najkrajših poti. Za neko povezavo (w, v) velja, da je del ene od najkrajših poti, če vozlišče v uporabi prvi element seznama S_w in ga doda v svoj seznam. Podobno, neka povezava (w, v) ni del nobene od najkrajših poti, če vozlišče v ugotovi, da prvi element seznama najkrajših poti vozlišča w ni sprejemljiv par v koraku nalaganja. Kadar zaznamo eno izmed dveh možnosti, dodamo novo povezavo iz seznama in: bodisi ohranimo kazalec $p[w, v]$, če je povezava (w, v) del ene od najkrajših poti; bodisi izbrišemo kazalec $p[w, v]$ (ga nadomestimo z novim), če povezava (w, v) ni del nobene od najkrajših poti.

Skupno število kazalcev je potem največ $m^* + n$, kar je $O(m^*)$, saj $m^* \geq n$. Čas, ki ga porabi izboljšan algoritem je potem $O(m^*n + m \lg n + nT_\psi(m^*, n))$.

Izrek 4.11. *Naj je ψ algoritem, ki rešuje problem najkrajših poti iz enega izvora na nenegativno uteženih grafih. Torej obstaja algoritem, ki rešuje problem najkrajših poti med vsemi pari vozlišč v nenegativno uteženih grafih v času $O(m^*n + m \lg n + nT_\psi(m^*, n))$ in prostoru $O(n^2 + S_\psi(m^*, n))$. Količina $T_\psi(m, n)$ predstavlja čas, ki ga porabi algoritem ψ na grafu z m povezavami in n vozlišči, in $S_\psi(m, n)$ je prostor, ki ga potrebuje algoritem ψ na istem grafu.*

Dokaz. Glej diskusijo zgoraj in v razdelku 4.2.2. □

4.2.6 Usmerjeni aciklični grafi

Tukaj pokažemo, kako združitev nekaterih znanih pristopov in algoritma 9 vodi do $O(m^*n + m \lg n)$ algoritma za problem najkrajših poti med vsemi pari vozlišč v poljubno uteženih usmerjenih acikličnih grafih. Prvi korak je preslikava prvotnega grafa, ki ima lahko negativne uteži, v nenegativno obliko. To storimo z Johnsonovim postopkom spremembe uteži, kot smo ga opisali v razdelku 3.9. Ker imamo opravka z usmerjenimi ackikličnimi grafi, lahko namesto algoritma Bellman-Ford v Johnsonovem postopku uporabimo algoritem topološkega urejanja. Dobimo usmerjen acikličen graf, ki je nenegativno utežen.

Sedaj uporabimo časovno izboljšan algoritem 9 kot je prikazan v razdelku 4.2.5. Kot algoritem ψ ponovno vzamemo algoritem topološkega urejanja. Če je graf G usmerjen acikličen graf, potem je graf G' tudi usmerjen acikličen graf. Dokaz je preprost: graf G' se razlikuje od G po uvedbi vozlišča i in povezav oblike $\forall v \in V : (i, v)$. Ker vozlišče i nima vhodnih povezav, je acikličnost grafa ohranjena. Časovne zahtevnosti so $O(m)$ za Johnsonov korak in $O(m^*n + m \lg n + nT_\psi(m^*, n))$ za korak APSP, kjer $T_\psi(m^*, n) = O(m^*)$. Skupen čas je potem: $O(m^*n + m \lg n)$. Prostorska zahtevnost je $\Theta(n^2)$.

Izrek 4.12. *Obstaja algoritem, ki reši problem najkrajših poti med vsemi pari vozlišč v usmerjenih acikličnih grafih v času $O(m^*n + m \lg n)$ in prostoru $\Theta(n^2)$.*

Dokaz. Glej diskusijo zgoraj. □

Pristop, ki smo ga tukaj uporabili na usmerjenih acikličnih grafih, lahko posplošimo na poljubne »posebne grafe«. V kolikor imamo SSSP algoritem ψ, ki rešuje problem SSSP ob predpostavki neke posebne lastnosti grafa, ga lahko pohitrimo samo, če ima graf G' tudi to lastnost. Pohitritev se torej lahko uporabi za grafe, ki so neusmerjeni, celoštevilčno uteženi, itd. Ne more pa se uporabiti npr. za planarne grafe, saj G' ni nujno planaren.

5 UGOTOVITVE IN NADALJNJE DELO

V magistrskem delu smo se srečali s problemom iskanja najkrajših poti v grafih. Podrobno smo si ogledali tako klasične, kot tudi novejše algoritme, ki ta problem učinkovito rešujejo. Glavni prispevki magistrskega dela so predvsem v predstavitvi novih algoritmov.

Prvi algoritem rešuje problem soroden problemu APSP: poiskati najkrajše poti med vsemi pari vozlišč neke podmnožice $S \subset V$ v grafu G. Nov algoritem, ki temelji na klasičnem algoritmu Floyd-Warshall, reši ta problem z uporabo manj operacij, kot, če bi uporabljali algoritem Floyd-Warshall.

Nato smo pokazali, da lahko izboljšamo klasičen način uporabe SSSP algoritma za izračun APSP tudi, če ne vemo ničesar o samem SSSP algoritmu. V poglavju 3 smo se že srečali s t. i. hierarhičnimi algoritmi, ki so med asimptotično najbolj učinkovitimi algoritmi za problem najkrajših poti. Naš pristop lahko pohitri samo takšne hierarhične algoritme, ki so učinkoviti pri reševanju SSSP problema. Takšen algoritem je trenutno le Thorupov [24] linearni algoritem za izračun SSSP v $\mathbb{Z}^+$ uteženih, neusmerjenih grafih. Glavna težava pri pohitritvi ostalih hierarhičnih algoritmov je, da je njihov postopek gradnje hierarhije časovno potraten. Ker algoritem 9 deluje na $n-1$ različnih grafih, bi bilo potrebno hierarhijo izračunati za vsak graf. Zanimivo bi bilo vedeti, če je možno hierarhijo postopoma posodabljati brez izgradnje na novo.

Način pohitritve, ki smo ga podali, deluje zgolj za nenegativno utežene grafe. Vendar, če obstaja $o(mn)$ algoritem za problem SSSP v poljubno uteženih grafih, potem je z uporabo Johnsonovega postopka spremembe uteži naša pohitritev zanimiva tudi za splošne grafe. Če tak algoritem ne obstaja, potem pohitritev nima smisla v asimptotičnem pogledu. Naj še omenimo, da obstajajo $o(mn)$ algoritmi za nekatere posebne oblike funkcije dolžine povezav [12, 13]. Za problem na poljubnih grafih trenutno ne poznamo nobenega $o(mn)$ algoritma.

Podali smo tudi povezavo med časovno zahtevnostjo problema najkrajših poti iz enega izvora in problema urejenih najkrajših poti med vsemi pari vozlišč. Zanimivo je, da lahko SSSP algoritem pomaga pri problemu urejanja. V kolikor lahko dokažemo spodnjo mejo $\omega(mn)$ za problem SAPSP, bi to pomenilo $\omega(m)$ spodnjo mejo za problem SSSP. Če take spodnje meje ni, to pomeni, da obstaja algoritem za problem APSP, ki obišče vozlišča v naraščajočem vrstnem redu glede na razdaljo od izvora (Dijkstrov pristop) in deluje v času $O(mn)$.

Literatura

[1] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[2] A. Brodnik, S. Carlsson, J. Karlsson, in J. I. Munro. Worst case constant time priority queue. V *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, pages 523–528, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 2001.

[3] J. Cheriyan in K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.

[4] T. H. Cormen, C. Stein, R. L. Rivest, in C. E. Leiserson. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2nd ed., 2001.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] M. Djordjevic, M. Grgurovič, J. Žibert, in A. Brodnik. Methods for solving the traveling visitor problem. V *Proceedings of the IBC 2012 : 1st International Internet & Business Conference*, BIT Society: Faculty of Economics & Business, Zagreb. 2012.

[7] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[8] L. R. Ford, Jr. Network flow theory. Paper P-923, The RAND Corporation, Santa Monica, California, 1956.

[9] M. L. Fredman in R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[10] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. V *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 90–100, IEEE Computer Society, Washington, DC, USA. 1985.

[11] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74:107–114, 2000.

[12] H. N. Gabow in R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[13] A. V. Goldberg. Scaling algorithms for the shortest paths problem. V *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pages 222–231, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 1993.

[14] T. Hagerup. Improved shortest paths on the word RAM. V *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, ICALP '00, pages 61–72, Springer-Verlag, London, UK. 2000.

[15] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[16] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[17] D. Karger, D. Koller, in S. J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.

[18] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.

[19] S. Pettie. Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time. V *Proceedings of the 16th international conference on Algorithms and Computation*, ISAAC'05, pages 964–973, Springer-Verlag, Berlin, Heidelberg. 2005.

[20] S. Pettie in V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM Journal on Computing*, 34(6):1398–1431, 2005.

[21] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[22] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[23] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[24] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, May 1999.

[25] M. Thorup. Equivalence between priority queues and sorting. V *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 125–134, IEEE Computer Society, Washington, DC, USA. 2002.

[26] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. V *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, STOC '03, pages 149–158, ACM, New York, NY, USA. 2003.

[27] P. van Emde Boas, R. Kaas, in E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10(1):99–127, 1976.

[28] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, January 1962.