

UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Lazar Milinković

**Implementacija algoritmov za
probleme najkrajših poti v
geometrijskih grafih**

MAGISTRSKO DELO
UNIVERZITETNI ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN MATEMATIKA

MENTOR: dr. Sergio Cabello

Ljubljana 2016

Rezultati diplomskega dela so intelektualna lastnina avtorja in Fakultete za matematiko in fiziko Univerze v Ljubljani. Za objavljane ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za matematiko in fiziko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil \LaTeX .

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU MAGISTRSKEGA DELA

Spodaj podpisani Lazar Milinković, z vpisno številko **27122037**, sem avtor magistrskega dela z naslovom:

Implementacija algoritmov za probleme najkrajših poti v geometrijskih grafih

S svojim podpisom zagotavljam, da:

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

V Ljubljani, dne 11. januarja 2016

Podpis avtorja:

Kazalo

Povzetek

Abstract

1	Uvod	1
2	CGAL in uporabljene matematične strukture	5
2.1	CGAL	5
2.2	Kd drevesa	6
2.3	SSSP drevesa	6
2.4	Graf kvadratne mreže	8
2.5	Delaunayeva triangulacija	8
2.5.1	Implementacija v CGAL	8
2.6	Voronoijev diagram	8
2.6.1	Voronoijev diagram v CGAL	10
2.7	Območna drevesa	11
3	Teorija	13
3.1	Predstavitev problema	13
3.2	SSSP drevo	15
3.3	Drevo najbližjega soseda	21
3.3.1	Optimizacijski problem 1	22
3.3.2	Optimizacijski problem 2	23
3.3.3	Sekanje nepremične daljice	29

3.4	Alternativni algoritmi za SSSP in minimalno ločevanje	31
3.4.1	Eksplisitni BFS	31
3.4.2	BFS z uporabo grafa kvadratne mreže	31
3.4.3	Generični algoritem za ločevanje z diski	32
4	Implementacija algoritma	33
4.1	SSSP drevo	33
4.1.1	Dodatki v razredu Point_2	33
4.1.2	Voronoijev diagram za iskanje najbližjega sosedu	34
4.1.3	Izgradnja drevesa	35
4.2	Drevo najbližjega sosedu	37
4.2.1	Kd drevo	38
4.3	Implementacija optimizacijskega problema $\Phi_\sigma(A, B)$	39
4.3.1	DualPoint - implementacija dualne točke	39
4.3.2	Območno drevo	39
4.4	Minimalno ločevanje	44
4.4.1	opt1	44
4.4.2	opt2	44
4.4.3	Kreiranje cikla	45
4.5	Obravnava posebnih vhodnih primerov	46
5	Rezultati	47
5.1	Drevo najkrajših poti	47
5.2	Minimalno ločevanje	47
6	Sklepne ugotovitve	55

Povzetek

KAZALO

Abstract

Poglavje 1

Uvod

V magistrski nalogi smo obravnavali dva geometrijska optimizacijska problema na ravnini, pri katerih osrednjo vlogo predstavljajo enotski krogi. Predstavili smo učinkovita algoritma, ki rešita omenjena problema, opisali njuno implementacijo ter predložili rezultate eksperimentov.

Pri prvem problemu bi radi izračunali drevo najkrajših poti iz neuteženega grafa presekov enotskih krogov. Kot vhod je dana množica \mathcal{D} n krogov enake velikosti, kjer je vsak krog opisan z njegovim središčem. Vozlišča grafa presekov predstavljajo krogi, pri čemer med dvema vozliščema obstaja povezava natanko takrat, ko obstaja presečišče med njunima krogoma D in D' . Graf presekov lahko predstavimo tudi na drug, bolj priročen način, kjer vozlišča predstavlja množica P središč krogov, povezava med dvema točkama p in q pa obstaja, če je njuna evklidska razdalja $\|pq\|$ manjša ali enaka premeru enotskega kroga. Za podan koren $r \in P$ lahko iz takega grafa izračunamo drevo najkrajših poti brez eksplicitne izgradnje grafa, s čimer dosežemo časovno zahtevnost $\mathcal{O}(n \log n)$ algoritma.

Drugi problem, ki ga obravnavamo, je problem minimalne ločitve. Kot vhod je dana množica \mathcal{D} n enotskih krogov na ravnini in dve točki s in t , ki nista vsebovani v nobenem od krogov iz \mathcal{D} . Pravimo, da \mathcal{D} ločuje s in t , če vsaka pot v ravnini od s do t seka nek krog v \mathcal{D} . Cilj je najti minimalno kardinalno podmnožico množice \mathcal{D} , ki ločuje s in t , kar formalno zapišemo

kot

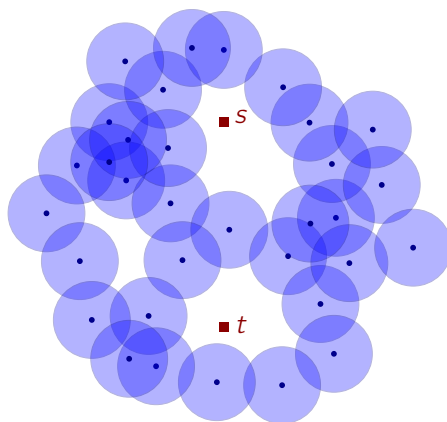
$$\begin{aligned} & \min |\mathcal{D}'| \\ & \text{tako da } \mathcal{D}' \subset \mathcal{D} \\ & \mathcal{D}' \text{ ločuje } s \text{ in } t. \end{aligned}$$

Polinomsko časovno zahtevnost algoritma $\mathcal{O}(n^2 \log^3 n)$ dosežemo tako, da za del rešitve uporabimo algoritem prvega problema za izgradnjo drevesa najkrajših poti. Oba problema sta tako povezana in ju je smiselno obravnavati skupaj.

Enotski krogi so najbolj standarden geometrijski model, ki se uporablja za brezžična senzorska omrežja (zanj se pogosto uporablja kratica UDG). Tak model predstavlja ustrezen trade-off med enostavnostjo in natančnostjo, saj je za bolj natančne modele dosti težje najti učinkovite algoritme. To pomeni tudi, da je izkoriščanje geometrijskih lastnosti modela UDG bolj zahtevno, algoritmi, ki se zanašajo na tak model, pa zaradi nerealističnih predpostavk pogosto odpovejo v praksi (kjer komunikacijski doseg ni idealen krog) [4]. Drevesa najkrajših poti v grafu enotskih krogov igrajo pomembno vlogo pri usmerjanju in se pogosto uporabljajo pri bolj kompleksnih nalogah. Primer take uporabe v senzorskih omrežjih je pri algoritmu za prepoznavanje meja, ki tvorijo luknje s premalo senzorskimi vozlišči. Zaradi takih lukenj veliko požrešnih algoritmov za posredovanje paketov po omrežju odpove, ker predpostavljajo, da je omrežje dovolj gosto [3].

Problem minimalne ločitve je obravnavan že v [2], kjer je podan tudi algoritem s časovno zahtevnostjo $\mathcal{O}(n^3)$ v najslabšem primeru, ki deluje za poljubne like. Z omejitvijo likov na enotne kroge in uporabo več različnih orodij iz računske geometrije lahko zmanjšamo časovno zahtevnost, lažje izvedljiva pa postane tudi implementacija algoritma.

V poglavju 2 je na kratko opisana programska knjižnica CGAL, ki smo jo uporabili pri implementaciji obeh algoritmov. Zanj smo se odločili, ker omogoča dostop do že implementiranih različnih geometrijskih podatkovnih



Slika 1.1: Slikovna predstavitev problema. Točki s in t ločuje množica krogov, definiranih z njihovimi središčnimi točkami.

struktur. Pri opisu smo se osredotočili na osrednje ideje jedra knjižnice, na katerem temeljijo vsi ostali deli paketa, ter tiste strukture, ki so bile uporabljene pri implementaciji algoritmov. V poglavju 3 je predstavljen teoretični del algoritmov. Podrobno so opisane vse podatkovne strukture, celotna poteka algoritmov ter njuna časovna in prostorska kompleksnost. V poglavju 4 je predstavljena implementacija algoritmov. Ponovno so opisane vse uporabljene podatkovne strukture in nekatere spremembe v datotekah CGAL-a z razredi, ki predstavljajo te strukture (ang. *headers*). Podrobno so opisani tudi deli algoritmov, kjer se pojavijo razlike med teoretičnim opisom in implementacijo. V poglavju 5 so predstavljeni rezultati; prikazani so časi izvajanja in prostorska poraba celotnega algoritma za različno število vhodnih točk. Enako smo storili posebej za nekatere podatkovne strukture, npr. Voronoijev diagram in SSSP drevesa.

Poglavje 2

CGAL in uporabljene matematične strukture

V tem poglavju so na kratko opisani programska knjižnica CGAL in podatkovne strukture, uporabljene v naših algoritmihi. Za bolj podroben opis glej (citiraj van krevelde, cgal.org in nekaj za bfs, sssp in grid graf). Vse podatkovne strukture so opisane za dvodimenzionalni primer.

2.1 CGAL

CGAL (Computational Geometry Algorithms Library) je programski paket, ki omogoča enostaven dostop do učinkovitih in zanesljivih geometrijskih algoritmov v obliki C++ knjižnice. Uporablja se na različnih področjih, ki potrebujejo geometrijsko računanje, kot so geografski informacijski sistemi, računalniško podprto načrtovanje, molekularna biologija, medicina, računalniška grafika in robotika. Knjižnica vsebuje:

- jedro z geometrijskimi osnovami, kot so točke, vektorji, črte, predikati za preizkušanje stvari (na primer relativni položaji točk) in opravila, kot so izračunavanje presekov ter razdalj
- osnovno knjižnico, ki je zbirka standardnih podatkovnih struktur in

geometrijskih algoritmov, kot so konveksna ovojnica v 2D/3D, Delaunayova triangulacija v 2D/3D, ravninski zemljevid, polieder, Voronoijev diagram, območna drevesa (range trees) itd.

- podporna knjižnica, ki ponuja vmesnike do drugih paketov, na primer za V/I in vizualizacijo.

Ker cilj magistrske naloge ni bila implementacija osnovnih geometrijskih struktur, ki sicer predstavljajo pomembno osnovo našega algoritma, smo se odločili uporabiti omenjeno knjižnico in si s tem prihraniti čas in odvečno delo. Posledično je seveda celoten algoritem implementiran v jeziku C++.

Pri nekaterih razredih knjižnice smo morali spremeniti kodo ali dodati kaj novega. Podrobnosti so razložene v nadaljevanju opisa implementacije pri ustreznih delih algoritma.

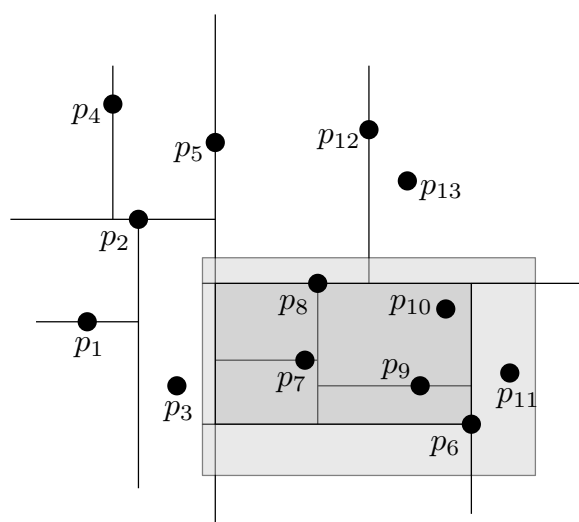
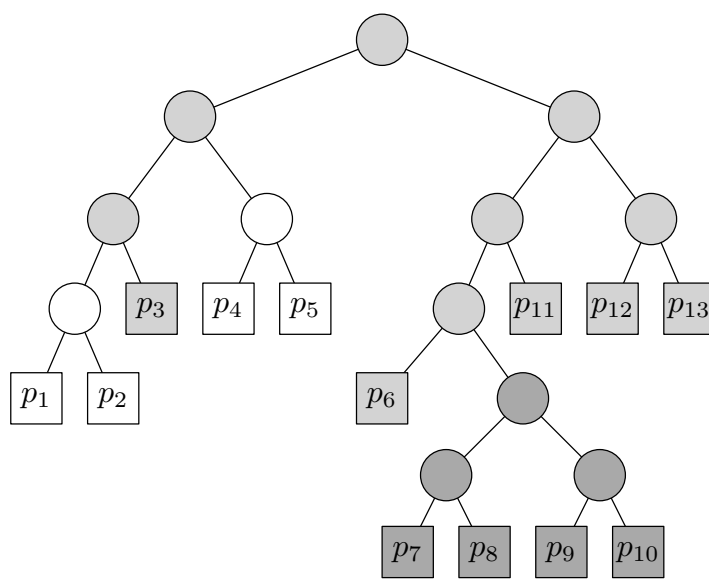
??? natancnost: cartesian, kaj pa leda, homogenous? povej kaj o traits, templated strukturami (*Point₂*)

2.2 Kd drevesa

Več na [1].

2.3 SSSP drevesa

Drevo najkrajših poti z eno izvirno točko je vpeto drevo T grafa G s korenom v , za katerega velja, da je razdalja poti od v do $u \in T$ enaka razdalji najkrajše poti od v do u . Tako drevo lahko zgradimo s pomočjo algoritma za iskanje najkrajše poti med dvema danima točkama (tipična primera sta Dijkstra in Bellman-Fordov algoritem). Izvirno točko v fiksiramo in poženemo algoritem za vse pare (v, u) , $v, u \in G$. Časovna kompleksnost Dijkstrovega algoritma je $\mathcal{O}((m + n) \log n)$, kjer je n število vozlišč, m pa število povezav v G . Ker moramo algoritem pognati za $n - 1$ parov, lahko drevo zgradimo v času $\mathcal{O}((n^2 + nm) \log n)$. Če predpostavimo, da za vhodni graf G veljajo



Slika 2.1: Dvodimenzionalno kd drevo.

določene omejitve (v primeru našega algoritma povezava med vozliščema v grafu obstaja samo, če je njuna razdalja največ 1), ki se jih da izkoristiti pri izgradnji hitrejšega algoritma, se časovna kompleksnost lahko izboljša.

2.4 Graf kvadratne mreže

Dvodimenzionalni graf kvadratne mreže je $m \times n$ mrežni graf $G_{m,n}$, ki predstavlja kartezični produkt dveh grafov poti z $n - 1$ in $m - 1$ povezavami. Vsi mrežni grafi so dvodelni, kar se da enostavno pokazati s tem, da se vozlišča pobarva z dvema barvama z vzorcem šahovnice.

Vozlišča grafa ponavadi sovpadajo s točkami v ravnini, pri katerih koordinate predstavljajo cela števila: abscisne koordinate imajo vrednosti $1, \dots, n$, ordinatne pa $1, \dots, m$. Dve vozlišči sta povezani, če je njuna razdalja enaka 1, zato takemu grafu pravimo tudi graf enotskih razdalj.

2.5 Delaunayeva triangulacija

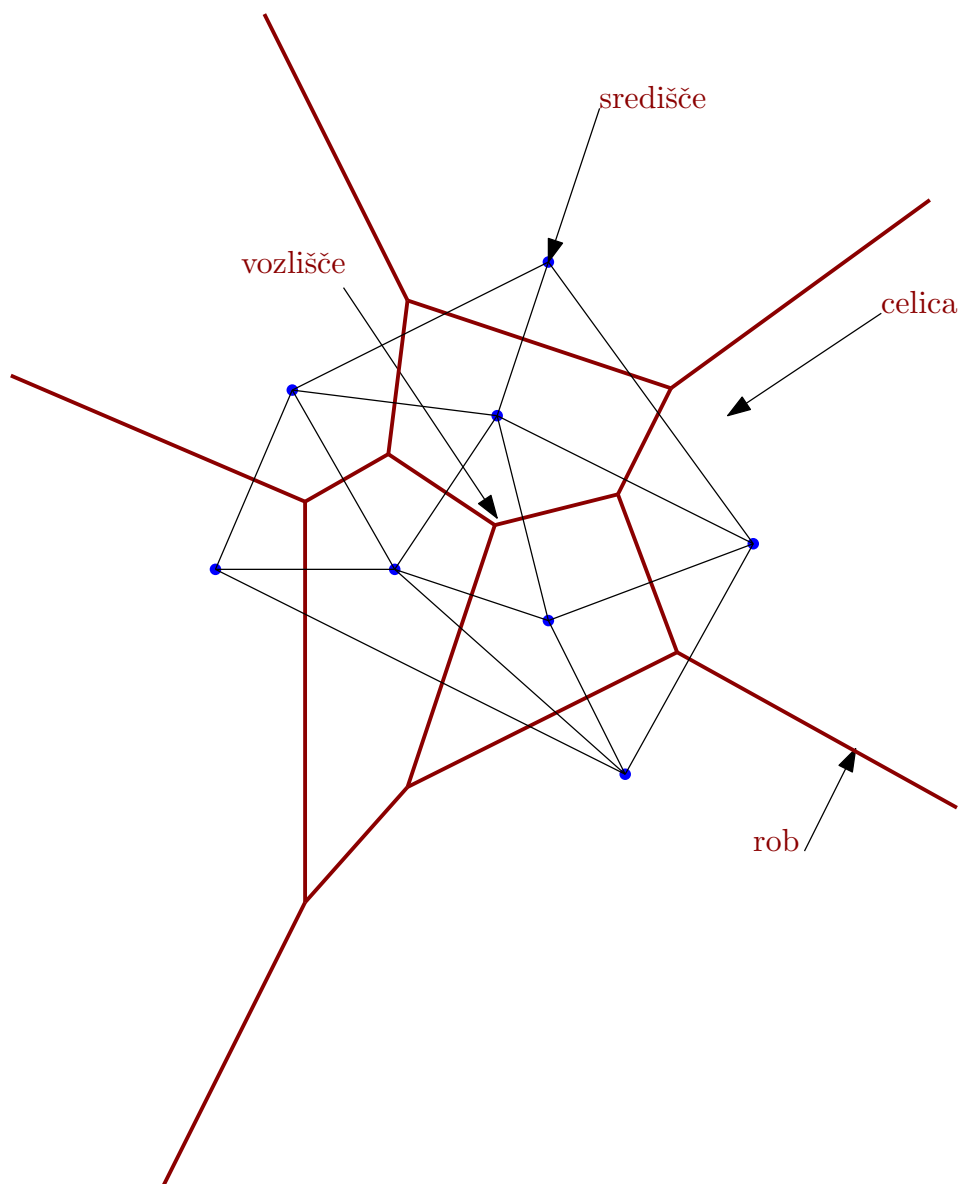
2.5.1 Implementacija v CGAL

2.6 Voronoijev diagram

Voronoijev diagram [?] je definiran na množici točk, imenovanih Voronoijeva središča (angleško *sites*), ki ležijo v nekem prostoru Σ , in z metriko oziroma funkcijo razdalje med točkami. Za ravninski Voronoijev diagram, opisan v nadaljevanju, velja $\Sigma = \mathbb{R}^2$.

Naj bo $S = \{S_1, S_2, \dots, S_n\}$ množica Voronoijevih središč in naj bo $\delta(x, S_i)$ funkcija razdalje med središčem S_i in neko točko $x \in \mathbb{R}^2$. Množica točk V_{ij} , ki so bližje središču S_i kot središču S_j na podlagi funkcije $\delta(x, \cdot)$, je množica:

$$V_{ij} = \{x \in \mathbb{R}^2 : \delta(x, S_i) < \delta(x, S_j)\}. \quad (2.1)$$



Slika 2.2: Evklidski Voronoijev diagram brez degeneracij, označen z rdečimi odebeljenimi povezavami. Voronoijeva središča, obarvana z modro barvo, predstavljajo hkrati tudi vozlišča v Delaunayevi triangulaciji, dualni Voronoijevemu diagramu, in je označena s tanjšimi črnimi povezavami.

Množico V_i točk, ki so bližje središču S_i kot kateremu koli drugemu središču, lahko potem definiramo kot množico:

$$V_i = \bigcap_{i \neq j} V_{ij}. \quad (2.2)$$

Množico V_i imenujemo tudi Voronijeva celica ali Voronijevo lice središča S_i . Lokus točk, ki so enako oddaljene od dveh središč, se imenuje Voronijev bisektor. Povezani podmnožici slednjega pravimo Voronijev rob. Točki, ki je enako oddaljena od treh ali več središč, pravimo Voronijevo vozlišče. Voronijev diagram na množici S in z metriko $\delta(x, \cdot)$ je zbirka Voronijevih celic, robov in vozlišč ter je primer ravninskega grafa.

Celice si ponavadi predstavljamo kot 2-dimenzionalne, robove kot 1-dimenzionalne in vozlišča kot 0-dimenzionalne objekte. Za določene kombinacije središč in metrik to ne drži. Voronijev diagram z metriko L_1 ali L_∞ lahko na primer vsebuje dvodimenzionalne robove. Takim Voronijevim diagramom, za katere zgornja omejitev drži in imajo lastnost, da so njihove celice preprosto povezano območje v ravnini, pravimo preprosti Voronijevi diagrami. Najbolj tipičen primer slednjih je evklidski Voronijev diagram (slika 2.2), ki ga uporabljamo v našem algoritmu.

2.6.1 Voronijev diagram v CGAL

Knjižnica CGAL ponuja razred `Voronoi_diagram_2` $\langle DG, AT, AP \rangle$, ki deluje kot prilagoditveni paket. Ta Delaunayevo triangulacijo na podlagi podanih kriterijev prilagodi pripadajočemu Voronijevemu diagramu, ki je predstavljen kot DCEL (doubly connected edge list) struktura. Paket je torej zasnovan tako, da na zunaj deluje kot DCEL struktura, znotraj pa v resnici hrani strukturo grafa, ki predstavlja graf triangulacije.

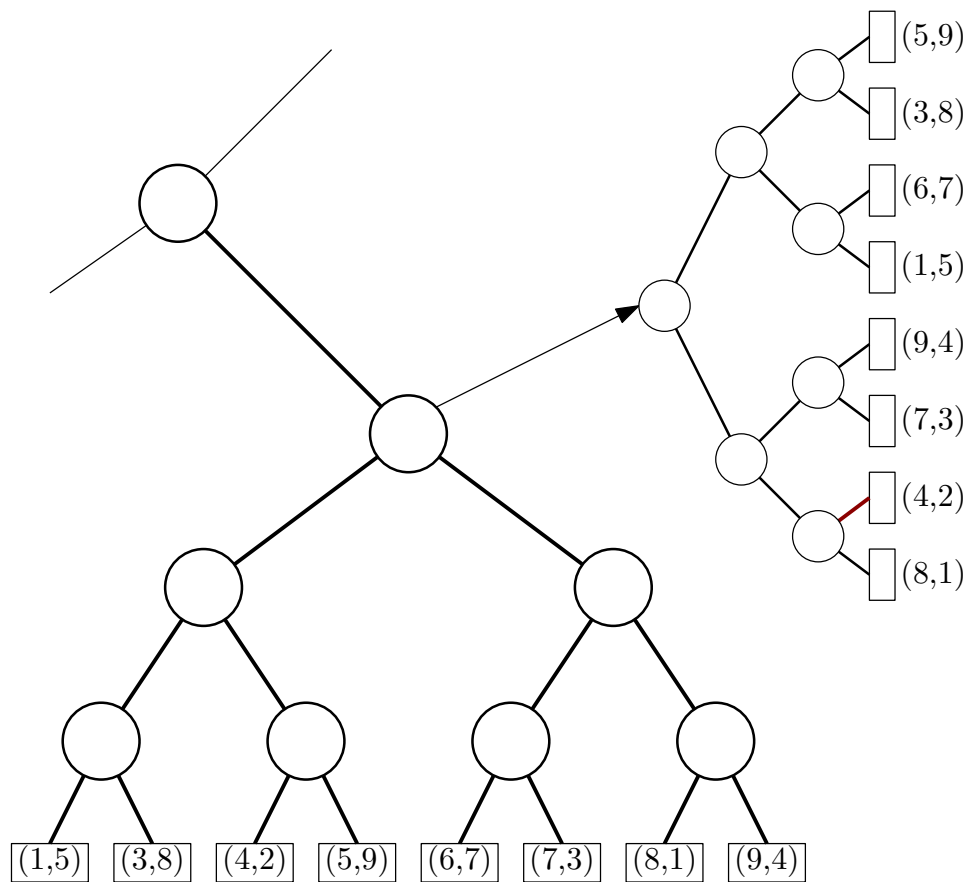
Razred je parametriziran s tremi predlogami. Prvi, DT, mora biti primer, ki predstavlja koncept razreda `DelaunayGraph_2`. Primeri takih struktur so Delaunayeva triangulacija, navadna triangulacija in Apollonov graf. Druga predloga, AT, predstavlja lastnosti prilagoditve ter definira tipe struktur in

funktorje, ki jih razred potrebuje za dostop do geometrijskih lastnosti Delaunayeve triangulacije. Funktor mora biti recimo definiran za konstrukcijo Voronoijevih vozlišč iz njihovih dualnih lic v Delaunayevi triangulaciji. Za našo implementacijo algoritma ločevanja z diski je pomemben funktor za poizvedbe najbližjih središč, ki kot rezultat vrne informacijo o tem, koliko in katera središča so enako oddaljena od točke poizvedbe. Bolj konkretno, rezultat je Delaunayevo vozlišče, lice ali rob, na katerem točka poizvedbe leži oziroma z njim sovpada. Če je na primer točka poizvedbe q , enako oddaljena od treh središč, potem sovpada z nekim Voronoijevim vozliščem, zato funktor vrne Delaunayevo lice, ki je dualno temu vozlišču. Razred, ki predstavlja Delaunayevo lice, omogoča iteracijo po Delaunayevih vozliščih, ki definirajo lice, ta pa so dualna trem Voronoijevim središčem.

Tretja predloga predstavlja režim adaptacije Delaunayeve triangulacije Voronoijevemu diagramu. Če množica središč, ki določa graf Delaunayeve triangulacije, vsebuje podmnožice središč, ki so v degeneriranem položaju, potem ima dualni graf - Voronoijev diagram - lahko robove dolžine nič in po možnosti tudi celice s ploščino nič. Režim adaptacije določa, kaj storiti v takih primerih. V našem projektu smo uporabili tip režima, imenovan *Delaunay_triangulation_caching_degeneracy_removal_policy_2*. Kot pove že ime, ta tip poskrbi, da so vse zgoraj opisane celice in robovi odstranjeni iz Voronoijevega diagrama. Poleg tega uporablja *cache* pri ugotavljanju, ali ima določena celica oziroma rob degenerirane lastnosti. Ker je slednje precej zahtevna operacija, se ta tip izplača pri vhodnih podatkih (središčih) z veliko degeneriranimi primeri.

2.7 Območna drevesa

Območna drevesa so še ena podatkovna struktura za poizvedbe nad pravokotnimi območji. V primerjavi s kd drevesi imajo boljši čas poizvedbe ($\mathcal{O}(\log^2 n)$), ampak slabšo prostorsko kompleksnost ($\mathcal{O}(n \log n)$).



Slika 2.3: Dvodimenzionalno območno drevo.

Poglavje 3

Teorija

3.1 Predstavitev problema

Naj bo \mathcal{D} množica skladnih enotskih krogov na Evklidski ravnini, ki jih definirajo njihove središčne točke. z in z' naj bosta dve dani točki, ki nista vsebovani v nobenem krogu. Za \mathcal{D} rečemo, da ločuje z in z' , če vsaka pot v ravnini od z do z' seka nek krog v \mathcal{D} .

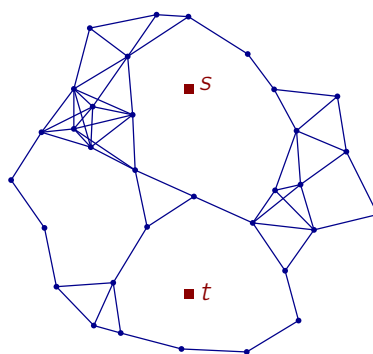
Problem iskanja minimalne kardinalne podmnožice \mathcal{D} , ki ločuje z in z' , lahko formalno zapišemo kot:

Cilj našega algoritma za ločevanje enotskih krogov je rešiti ta problem v skoraj kvadratičnem času. Do take časovne kompleksnosti pridemo tudi s pomočjo še enega našega algoritma za izgradnjo drevesa najkrajših poti iz enega, danega izhodišča. Oba algoritma sta opisana v nadaljevanju, prav tako dokaza za njuno časovno kompleksnost.

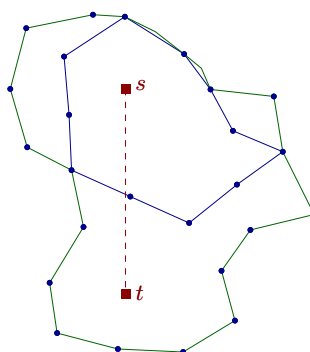
Naj bo P množica točk, ki predstavljajo središča krogov \mathcal{D} . P predstavlja vhod našega algoritma in vse operacije ter uporabljene podatkovne strukture se vrtijo okoli te množice. Kardinalnost množice je enaka n .

Naj bo $G(P)$ graf z množico vozlišč P s povezavo med točkama $p, q \in P$, če velja $|pq| \leq 1$ z evklidsko metriko. Vse povezave so neobtežene.

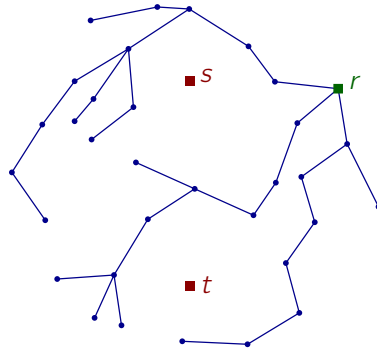
V nadaljevanju bomo namesto $G(P)$ pisali preprosto G , brez izgube splošnosti pa predpostavili, da je $z = (0, 0)$ in $z' = (0, s)$. zz' je torej daljica,



Slika 3.1: Graf $G(P)$, zgrajen nad množico P v sliki 1.1.



Slika 3.2: Minimalna podmnožica točk iz slike 1.1, ki ločuje s in t , med seboj povezanih z modrimi povezavami. Podmnožica je vedno zaprta pot.



Slika 3.3: Drevo najbližjega soseda, ki hrani 6 točk, predstavljeno v obliki seznama. Za vsako celico so prikazani indeksi točk, ki jih hrani objekt v njej.

v nadaljevanju imenovana σ , ki je vsebovana v koordinatni osi y . Če dana vhodna daljica ne leži na osi y , lahko naredimo translacijo nad σ in P ; če poleg tega tudi vertikalna ni, naredimo še rotacijo (kjer pa lahko pride do numeričnih napak).

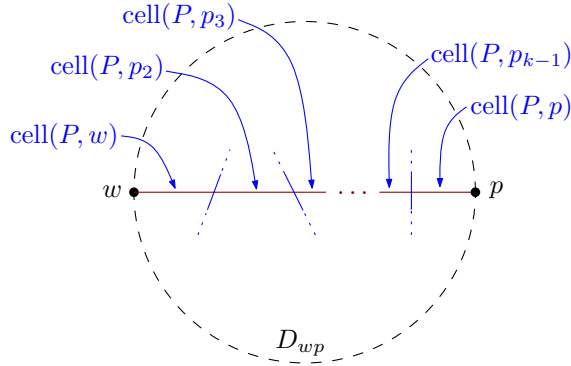
3.2 SSSP drevo

V tem poglavju je opisan algoritem za drevo SSSP (ang. single source shortest path). Gre za preiskovanje v širino nad grafom $G(P)$ brez dejanske izgradnje grafa. Vhod algoritma je torej P , pri izgradnji drevesa pa se uporablja abstrakten graf $G(P)$.

Algoritem dobi kot vhod poleg množice P izvorno točko $s \in P$ in nato inkrementalno v vsaki iteraciji dodaja točke h grafu. Množico točk, dodanih k drevesu v i -ti iteraciji, označimo kot

$$W_i = \{p \in P \mid d_{G(P)}(s, p) = i\}.$$

Velja torej $W_0 = \{s\}$. Za izgradnjo W_i ne potrebujemo celotnega grafa, zgrajenega do $i - 1$ -te iteracije, temveč samo množico W_{i-1} . Za vsak $q \in W_i$



Slika 3.4: aa

velja, da je povezan s točko $p = NN(q, W_{i-1})$. p je torej izmed točk v W_{i-1} najbližje q .

Pri izgradnji W_i ne pregledujemo vseh še ne dodanih točk. Že v samem začetku najprej zgradimo Delaunayevo triangulacijo $DT(P)$, ki nam je v pomoč pri iskanju primernih kandidatov za W_i . To so:

- točke sosednje W_{i-1} v $DT(P)$
- točke sosednje (do tega trenutka zgrajeni) W_i v $DT(P)$.

Lema 3.1. *Naj bo p točka v $P \setminus \{s\}$, za katero velja $d(s, p) < \infty$. Obstajata točka w v P in pot Π v $DT(P)$ od w do p , za kateri velja $d(s, w) + 1 = d(s, p)$ in $d(s, p_j) = d(s, p)$ za vsako notranje vozlišče p_j v Π .*

Dokaz. Naj bo $i = d(s, p)$, w pa naj bo točka z $d(s, w) = i - 1$, ki je najbližje p po evklidski razdalji. Ker $d(s, p) < \infty$, mora veljati $\|w - p\| \leq 1$. Naj bo D_{wp} krog s premerom wp .

Predpostavimo, da segment wp ne gre skozi nobeno vozlišče VD nad P . Pobljše si pogledjmo zaporedje Voronoijevih celic $cell(p_1, P), \dots, cell(p_k, P)$, ki ga seka segment wp , ko se sprehodimo od w do p . Očitno velja $w = p_1$ in $p = p_k$. Za vsak $1 \leq j < k$ je povezava $p_j p_{j+1}$ v $DT(P)$, ker sta si celici $cell(p_j, P)$ in $cell(p_{j+1}, P)$ sosedni prek neke točke v wp . Pot $\pi = p_1 p_2 \dots p_k$ je zato vsebovana v $DT(P)$ in povezuje w s p . Za katerikoli indeks

j , kjer $1 < j < k$, naj bo a_j katerakoli točka v $wp \cap \text{cell}(p_j, P)$. Ker velja $\|a_j p_j\| \leq \{\|a_j w\|, \|a_j p\|\}$, je točka p_j vsebovana v D_{wp} . Potem je celotna pot π vsebovana v D_{wp} , in ker je premer D_{wp} največ 1, je vsaka povezava poti π tudi v $G(P)$. S tem sklenemo, da je π pot v $DT(P) \cap G(P)$.

Vzemimo katerokoli točko p_j v π , ki je potem vsebovana v D_{wp} . Ker $\|w - p_j\| \leq \|w - p\| \leq 1$, velja $d(s, p_j) \leq d(s, w) + 1 = i$. Ker $\|p_j - p\| \leq \|w - p\| \leq 1$, velja $d(s, p_j) \geq d(s, p) - 1 = i - 1$. Ampak izbira w kot točke najbližje p pomeni, da $d(s, p_j) \neq i - 1$, ker $\|p_j - p\| < \|w - p\|$. Zato velja $d(s, p_j) = i$. Iz tega sklenemo, da za vsa notranja vozlišča p_j v π velja $d(s, p_j) = i$. \square

Lema 3.2. *Na koncu algoritma $\text{UnweightedShortestPath}(P, s)$ velja:*

$$\forall i \in \mathbb{N} \cup \{0\} : W_i = \{p \in P \mid d(s, p) = i\}. \quad (3.1)$$

Poleg tega za vsako točko $p \in P \setminus \{s\}$ velja $\text{dist}[p] = d(s, p)$ in če velja $d(s, p) < \infty$, potem obstaja najkrajša pot v $G(P)$ od s do p , v kateri je zadnja povezava $\pi[p]p$.

Dokaz. Za dokaz uporabimo indukcijo nad i . $W_0 = \{s\}$ je nastavljen v vrstici 6 in kasneje ne spremeni vrednosti. Za $i = 0$ potem izjava velja.

Preden obravnavamo indukcijski korak, izpostavimo, da so množice W_0, W_1, \dots parno disjunktne. To je razvidno tudi iz psevdokode. Točka p je v vrstici 21 dodana v nek W_i , po tem, ko nastavimo $\text{dist}[p] = i$ v vrstici 18. Po tem je pogoj v vrstici 17 vedno neresničen in p ni dodan v nobeno drugo množico W_j .

Vzemimo katerikoli $i \geq 0$. Po indukcijski predpostavki velja

$$W_{i-1} = \{p \in P \mid d(s, p) = i - 1\}. \quad (3.2)$$

V algoritmu dodamo točke v W_i samo v vrstici 21. Če je p dodan v W_i , potem velja $\|p - w\| \leq 1$ za nek $w \in W_{i-1}$ zaradi pogoja v vrstici 17. Potem vsak p , dodan v W_i , zadostuje pogoju $d(s, p) \leq i$. Ker $p \notin W_{i-1}$, iz disjunkcije množic W_0, W_1, \dots sledi $d(s, p) = i$. Sklenemo, da

$$W_i \subseteq \{p \in P \mid d(s, p) = i\}. \quad (3.3)$$

Da dobimo vsebovanost v drugo smer, naj bo p neka točka, za katero velja $d(s, p) = i$. Pokazati moramo, da je z algoritmom p dodan v W_i . Vzemimo točko w in pot $\pi = p_1 \dots p_k$, zagotovljeno z lemo 3.1. Z indukcijsko hipotezo imamo $w = p_1 \in W_{i-1}$ in zato je w dodan v Q v vrstici 10. V nekem trenutku je povezava $p_1 p_2$ obravnavana v vrstici 15 in točka p_2 je dodana v W_i in Q . Po *indukcijskitez*!!! so potem vse točke p_3, \dots, p_k dodane v W_i in Q (po možnosti v različnem vrstnem redu). Sledi, da je $p_k = p$ dodan v W_i in zato

$$W_i = \{p \in P \mid d(s, p) = i\}. \quad (3.4)$$

Ker je p dodan v W_i ob istem času, ko je nastavljen $dist[p] = i$, sledi, da $dist[p] = i = d(s, p)$. Ker $\pi[p] \in W_{i-1}$ in $\|p - \pi[p]\| \leq 1$ (vrstice 16, 17 in 19), obstaja najkrajša pot v $G(P)$ od s do p , ki uporablja $(i-1)$ -to povezavo poti od s do $\pi[p]$, po indukciji pa ji sledi povezava $\pi[p]p$. \square

Da dobimo odgovor na vprašanje, ali je nek kandidat $p \in W_i$, uporabimo podatkovno strukturo, ki zna v zglednem času najti najbližjega soseda q in preveriti, če je razdalja med njima manjša ali enaka 1.

Celotna psevdokoda algoritma za izgradnjo drevesa je opisana spodaj.

Lema 3.3. *Za izgradnjo drevesa SSSP potrebuje algoritem $UnweightedShortestPath(P, s)$ $\mathcal{O}(n \log n)$ časa, kjer je n velikost množice P .*

Dokaz. Glavne opazke, uporabljene v dokazu, so sledeče: vsaka točka v P je dodana v Q največ enkrat v vrstici 10 in enkrat v vrstici 20, izvajanje vrstic 13-21 za točko q vzame $\mathcal{O}(\deg_{DT(P)}(q) \log n)$, vsota stopenj v $DT(P)$ je $\mathcal{O}(n)$, in v vrstici 9 porabimo $\mathcal{O}(n \log n)$ časa skupaj za vse iteracije. Sledijo podrobnosti.

Delaunayeva triangulacija nad n točkami se lahko izračuna v času $\mathcal{O}(n \log n)$. Inicializacija v vrsticah 1-7 tako vzame $\mathcal{O}(n \log n)$ časa. Dokazati moramo še, da zanka v vrsticah 8-22 vzame $\mathcal{O}(n \log n)$ časa.

Algorithm 1 Algoritem za izgradnjo SSSP drevesa

```

1: procedure BUILDTREE
2:   for  $p \in P$  do
3:      $\text{dist}[p] \leftarrow \infty$ 
4:    $\text{dist}[r] \leftarrow 0$ 
5:   zgradi Delaunayevo triangulacijo  $DT(P)$ 
6:    $W_0 \leftarrow \{s\}$ 
7:    $i \leftarrow 1$ 
8:   while  $W_{i-1} \neq \emptyset$  do
9:     zgradi pod. strukturo za poizvedbe najbližjega sosedu v  $W_{i-1}$ 
10:     $Q \leftarrow W_{i-1}$  ( $\star$  generator točk kandidatk  $\star$ )
11:     $W_i \leftarrow \emptyset$ 
12:    while  $Q \neq \emptyset$  do
13:       $q$  naj bo poljubna točka v  $Q$ 
14:      odstrani  $q$  iz  $Q$ 
15:      for povezava  $qp$  v  $DT(P)$  do
16:         $w \leftarrow \text{NN}(W_{i-1}, p)$ 
17:        if  $\text{dist}[p] = \infty$  and  $|pw| \leq 1$  then
18:           $\text{dist}[p] \leftarrow i$ 
19:           $\pi[p] \leftarrow w$ 
20:          dodaj  $p$  v  $Q$ 
21:          dodaj  $p$  v  $W_i$ 
22:         $i \leftarrow i + 1$ 
23:   return  $\text{dist}[\cdot]$ 

```

Izvajanje vrstic 9-11 vzame $\mathcal{O}(|W_{i-1}| \log |W_{i-1}|) = \mathcal{O}(|W_{i-1}| \log n)$ časa. Vsaka kasnejša poizvedba najbližjega soseda se izvede v času $\mathcal{O}(\log n)$.

Vsaka izvedba vrstic 16-21 se izvede v času $\mathcal{O}(\log n)$, kjer je najbolj zahteven korak poizvedba v vrstici 16. Vsaka izvedba vrstic 13-21 se izvede v času $\mathcal{O}(\deg_{DT(P)}(q) \cdot \log n)$, ker se vrstice 16-21 izvedejo $\deg_{DT(P)}(q)$ -krat.

Obravnavajmo eno izvedbo vrstic 9-22. Točke so dodane v Q v vrsticah 10 in 20. V slednji je točka p dodana v Q natanko takrat, ko je dodana v W_i (v vrstici 21). Iz tega sledi, da je p dodana v Q natanko takrat, ko pripada množici $W_{i-1} \cup W_i$. Poleg tega je vsaka točka iz $W_{i-1} \cup W_i$ dodana v Q natanko enkrat: za vsako točko p , ki je dodana v Q , velja $\text{dist}[p] \leq i < \infty$, in da ne bo dodana nikoli več zaradi pogoja v vrstici 17. Iz tega sledi, da se zanka v vrsticah 12-22 izvede v času

$$\sum_{q \in W_{i-1} \cup W_i} \mathcal{O}(\deg_{DT(P)}(q) \cdot \log n). \quad (3.5)$$

Tako lahko omejimo porabljen čas v zanki v vrsticah 8-22 z

$$\sum_i \mathcal{O} \left(|W_i| \log n + \sum_{q \in W_{i-1} \cup W_i} (\deg_{DT(P)}(q) \cdot \log n) \right). \quad (3.6)$$

Z uporabo leme 3.2, ki pravi, da so množice W_0, W_1, \dots parno disjunktne, ter relacijama $\sum_i |W_i| \leq n$ in

$$\sum_{q \in P} \deg_{DT(P)}(q) = 2 \cdot |E(DT(P))| = \mathcal{O}(n), \quad (3.7)$$

časovna kompleksnost iz 3.6 postane $\mathcal{O}(n \log n)$. \square

Izrek 3.4. *Naj bo P množica n točk v ravnini in naj bo s točka v P . V času $\mathcal{O}(n \log n)$ lahko iz neuteženega grafa $G(P)$ izračunamo drevo najkrajših poti s korenem s .*

Dokaz. Zaradi leme 3.3 porabi algoritem *UnweightedShortestPath*(P, s) $\mathcal{O}(n \log n)$ časa. Zaradi leme 3.2 tabela $\pi[\cdot]$ pravilno opisuje drevo najkrajših poti v $G(P)$ s korenem s in $\text{dist}[\cdot]$ pravilno opisuje razdalje najkrajših poti v $G(P)$. \square

3.3 Drevo najbližjega soseda

Lema 3.5. *Naj bo P množica uteženih točk na ravnini. V času $\mathcal{O}(n \log n)$ lahko konstruiramo tako podatkovno strukturo, ki v času $\mathcal{O}(\log^2 n)$ za točko poizvedbe q najde točko v*

$$\arg \min \{w_p \mid p \in P, |pq| < 1\}.$$

Dokaz. Točke P sortiramo nepadajoče po njihovih utežeh in zgradimo uravnoreženo binarno iskalno drevo \mathcal{T} . Točke P vstavimo v liste \mathcal{T} tako, da se zaporedji, ki izhajata iz \mathcal{T} in sortirane množice P , ujemata.

Za vsako vozlišče ν v \mathcal{T} :

- Označimo $P(\nu)$ kot množico točk, shranjenih v poddrevesu s korenem ν . Taki množici rečemo *kanonična podmnožica*.
- Označimo $U(\nu)$ kot unijo enotskih krogov s središči $P(\nu)$.
- Zgradimo point-location podatkovno strukturo $DS(P(\nu))$, ki v ozadju uporablja Voronoijev diagram ali kakšno drugo podobno podatkovno strukturo za iskanje najbližjega soseda. Za dano točko poizvedbe q DS pove, če se nahaja v $U(\nu)$. Najprej poišče najbližjega soseda q tako, da najde celico c Voronoijevega diagrama, v kateri se nahaja q in s tem središče S_i , ki definira c . Nato preveri, če je razdalja med q in S_i največ ena merska enota.

Čas predprocesiranja vozlišča ν je $\mathcal{O}(|P(\nu)|)$, čas poizvedbe pa $\mathcal{O}(\log |P(\nu)|) = \mathcal{O}(\log n)$.

Analizirajmo čas izgradnje naše podatkovne strukture. Točke P sortiramo leksikografsko v korenu drevesa \mathcal{T} , točke $P(\nu)$ pa za vsak ν dobimo že sortirane od starša ν . Ker so kanonične podmnožice na vsakem nivoju drevesa med seboj disjunktne, porabimo za vsak nivo $\mathcal{O}(n)$ časa. Uravnoreženost \mathcal{T} zagotavlja, da je $\mathcal{O}(\log n)$ nivojev, tako da skupaj za izgradnjo potrebujemo $\mathcal{O}(n \log n)$ časa.

Potrebna je še analiza časa poizvedbe. Za točko poizvedbe q preverimo, če je vsebovana v $U(r)$, kjer je r koren drevesa. Če ni, potem nobena točka iz P ni dovolj blizu q . Sicer označimo $\nu = r$ in nadaljujemo z iskanjem od vrha navzdol po T , dokler ne pridemo do listov drevesa. Ko ν ni list, označimo njegovega levega in desnega otroka z ν_ℓ in ν_r . Če se q nahaja v $U(\nu_\ell)$, označimo $\nu = \nu_\ell$ in nadaljujemo navzdol po poddrevesu otroka ν_ℓ , sicer to storimo za desnega otroka. V vsakem trenutku poizvedbe ohranjamo sledečo invarianto:

$$P(\nu) \cap \arg \min\{w_p \mid p \in P, |pq| < 1\} \neq \emptyset.$$

Pot poizvedbe v T ima $\mathcal{O}(\log n)$ vozlišč, za vsako vozlišče pa porabimo $\mathcal{O}(\log n)$ časa, da ugotovimo, če se q nahaja v $U(\nu)$. Skupni čas poizvedbe je torej $\mathcal{O}(\log^2 n)$. \square

3.3.1 Optimizacijski problem 1

Obravnavajmo sledeč optimizacijski problem dveh množic uteženih točk A in B :

$$\begin{aligned} \Phi(A, B) &:= \min w_a + w_b \\ \text{s.t. } &a \in A, b \in B \\ &|ab| \leq 1. \end{aligned}$$

Lema 3.6. *Naj bosta A and B množici največ n uteženih točk v ravnini. $\Phi(A, B)$ lahko izračunamo v času $\mathcal{O}(n \log^2 n)$.*

Dokaz. Za B zgradimo podatkovno strukturo iz prejšnje leme. Za vsak $a \in A$ naredimo poizvedbo na podatkovni strukturi, da dobimo

$$b^*(a) \in \arg \min\{w_b \mid b \in B, |ab| \leq 1\}.$$

Nato najdemo tak a , ki minimizira vsoto $w_a + w_{b^*(a)}$.

$\mathcal{O}(n \log n)$ časa rabimo za izgradnjo podatkovne strukture, $\mathcal{O}(\log^2 n)$ pa za vsako poizvedbo. Ker je poizvedb največ n , je časovna kompleksnost enaka $\mathcal{O}(n \log^2 n)$. \square

3.3.2 Optimizacijski problem 2

Naj bo σ daljica v ravnini in brez izgube splošnosti lahko predpostavimo, da σ leži na osi y s krajiščema $(0, 0)$ in $(0, s)$. Naj bo A množica točk z negativno koordinato x in B množica točk z nenegativno koordinato x . Vsaka točka p množice $A \cup B$ ima utež ω_p . Minimizarati hočemo vsoto $\omega_a + \omega_b$ za vse take pare $(a, b) \in A \times B$, pri katerih je daljica ab dolžine največ 1 in seka daljico σ :

$$\begin{aligned} \Phi_\sigma(A, B) &:= \min w_a + w_b \\ \text{s.t. } &a \in A, b \in B \\ &|ab| \leq 1 \\ &ab \text{ seka } \sigma. \end{aligned}$$

Za vsako točko $a \in A$ definiramo množici

$$\begin{aligned} B(a) &= \{b \in B \mid ab \text{ seka } \sigma\}, \\ B_{\leq 1}(a) &= \{b \in B \mid ab \text{ seka } \sigma \text{ in } |ab| \leq 1\} = \{b \in B(a) \mid |ab| \leq 1\} \end{aligned}$$

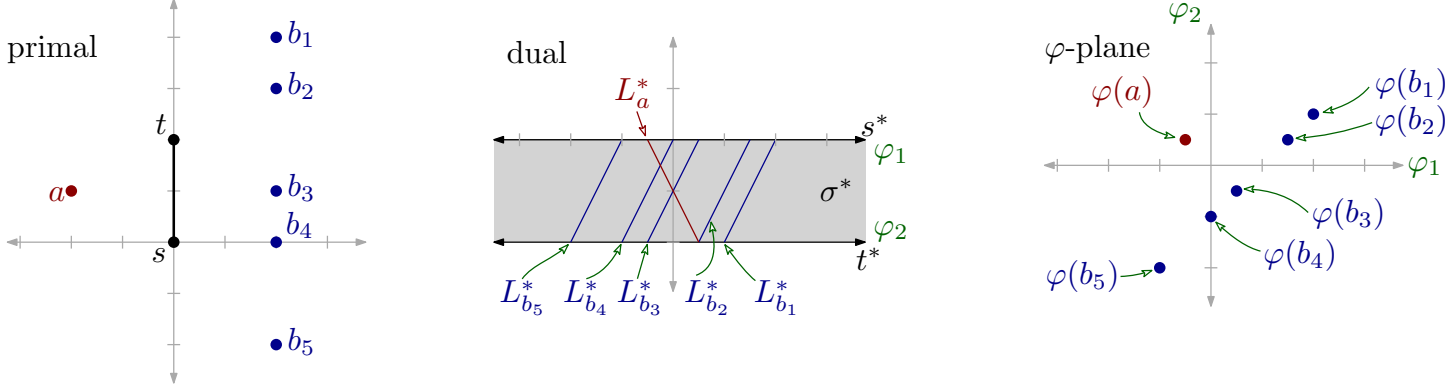
in optimizacijski problem

$$\Phi_\sigma(a, B) = w_a + \min\{w_b \mid b \in B_{\leq 1}(a)\}.$$

Če združimo oba optimizacijska problema skupaj, dobimo

$$\Phi_\sigma(A, B) = \min_{a \in A} \Phi_\sigma(a, B).$$

V nadaljevanju bomo opisali podatkovno strukturo, s katero lahko kompaktno dobimo množico $B(\cdot)$ in pokazali, da njene lastnosti ustrezajo lastnostim območnih dreves.



Slika 3.5: .

Dualnost in dualni prostor

Opiši iz poglavja knjige.

Lema 3.7. *Obstaja družina $\{B_1, \dots, B_t\}$ podmnožic množice B in podatkovna struktura $\mathcal{D}(B)$ z naslednjimi lastnostmi*

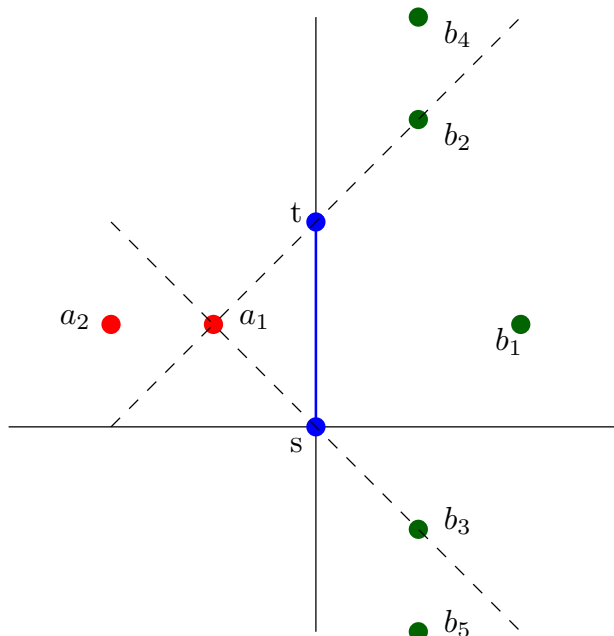
- $\sum_{i=1}^t |B_i| = \mathcal{O}(n \log n)$;
- $\mathcal{D}(B)$ je velikosti $\mathcal{O}(n \log n)$ in se jo da konstruirati v času $\mathcal{O}(n \log n)$;
- za vsako točko a z negativno koordinato x obstaja podmnožica indeksov $I(a) \subset \{1, \dots, t\}$, tako da velja $|I(a)| = \mathcal{O}(\log^2 n)$, $B(a)$ pa je disjunktna unija množic $\{B_i\}_{i \in I(a)}$;
- za vsako točko poizvedbe a z $a_x < 0$ podatkovna struktura $\mathcal{D}(B)$ vrne $I(a)$ v $\mathcal{O}(\log^2 n)$ času.

Dokaz. Za potrebe dokaza uporabimo dualnost, opisano zgoraj.

Naj bo \mathbb{L} množica nevertikalnih premic, σ^* pa množica točk dualnih nevertikalnim daljicam, ki sekajo daljico σ :

$$\sigma^* = \{l^* \mid l \in \mathbb{L}, l \cap \sigma \neq \emptyset\}$$

V dualnem prostoru je množica σ^* *horizontal slab*



Slika 3.6: Množici A (rdeči točki) ter B (zelene točke) levo in desno od daljice st (zgornja slika). Črtkani črti predstavljata premici, čigar koeficienta sta uporabljena za izračun koordinat dualne točke $\varphi(a_1)$, predstavljata pa tudi vidno polje točke a_1 .

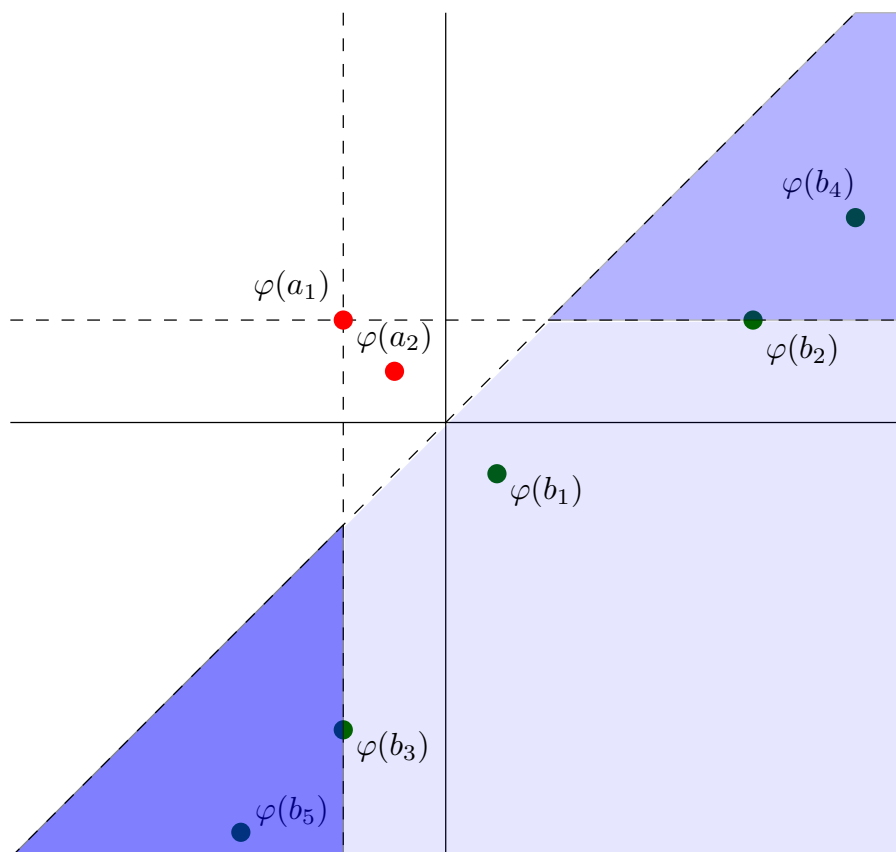
$$\sigma^* = \{(m, -c) \in \mathbb{R}^2 \mid 0 \leq c \leq s\}.$$

Za vsako točko $b \in B$ naj bo L_b^* množica točk, dualnih premicam, ki gredo skozi b in sekajo σ :

$$L_b^* = \{\ell^* \mid \ell \in \mathbb{L}, b \in \ell, \text{ and } \sigma \cap \ell \neq \emptyset\}.$$

V dualnem prostoru je L_b^* daljica, ki je popolnoma vsebovana v slabu in ima krajišči $(\varphi_1(b), 0)$ in $(\varphi_2(b), s)$ na obeh njegovih mejah. $\varphi_1(b)$ predstavlja smerni koeficient premice, ki seka točki $(0, s)$ in b , $\varphi_2(b)$ pa smerni koeficient premice, ki seka točki $(0, 0)$ in b .

Definirajmo točko preslikave $\varphi(b) = (\varphi_1(b), \varphi_2(b))$. Funkcija preslikave φ torej preslika točke na desni strani koordinatne osi y v točke v ravnini.



Slika 3.7: Spodaj so prikazane točke iz slike 3.6 v dualnem prostoru. Obarvano območje v sredini predstavlja prostor, v katerem se nahajajo vse take dualne točke $\varphi(b)$, za katere velja $b \in B(a_1)$. Obarvani območji levo in zgoraj predstavljata prostor dualnih točk $\varphi(b)$, za katere velja, da je presečišče premice a_1b z osjo y pod oziroma nad daljico st .

Za vsak $b \in B$ velja neenakost $\varphi_1(b) \geq \varphi_2(b)$. Točke B lahko razdelimo v tri skupine glede na predznaka koordinat točke preslikave $\varphi(b)$ in za vsako skupino je neenakost očitna:

$$\begin{aligned} b_1 &\in \{(x, y) \mid (x, y) \in B, y < 0\} \Rightarrow \varphi_1(b), \varphi_2(b) < 0 \text{ and } \varphi_1(b) > \varphi_2(b) \\ b_2 &\in \{(x, y) \mid (x, y) \in B, 0 \leq y < s\} \Rightarrow \varphi_1(b) > 0, \varphi_2(b) < 0 \\ b_3 &\in \{(x, y) \mid (x, y) \in B, y \geq s\} \Rightarrow \varphi_1(b) > 0, \varphi_2(b) \geq 0 \text{ and } \varphi_1(b) > \varphi_2(b) \end{aligned}$$

Enakost velja le v primeru, ko sta premici, ki definirata obe koordinati, isti. Do slednjega pride pri točkah b s koordinato x enako 0.

Iz zgornje neenakosti sledi, da točke preslikave $\varphi(b)$ vedno ležijo na polravnini $x \geq y$ in da je smerni koeficient premice, na kateri leži daljica L_b^* , nepozitiven. Podobno lahko ugotovimo, da za vsak $a \in A$ točka $\varphi(a)$ leži na polravnini $\varphi_2(a) > \varphi_1(a)$ in da ima premica, na kateri leži daljica L_a^* , pozitiven smerni koeficient.

Naj bo $a \in A$ in $b \in B$. Daljica ab seka daljico σ natanko takrat, ko L_a^* seka L_b^* , ker daljico ab v dualnem prostoru predstavlja ravno presečišče L_a^* in L_b^* . Iz tega sledi naslednja lastnost:

$$\begin{aligned} ab \cap \sigma \neq \emptyset &\iff (\varphi_1(a) - \varphi_1(b)) \cdot (\varphi_2(a) - \varphi_2(b)) < 0 \\ &\varphi_1(a) \leq \varphi_1(b) \\ &\varphi_2(a) \geq \varphi_2(b). \end{aligned}$$

Z drugimi besedami: za točko $a \in A$ množico točk $b \in B$, kjer ab seka σ , sestavljajo točke b , pri katerih se $\varphi(b)$ nahaja v drugem kvadrantu koordinatnega sistema z izhodiščem $\varphi(a)$. (Bolj natančno, gre za točke $\varphi(b)$, ki se nahajajo v preseku drugega kvadranta omenjenega koordinatnega sistema s polravnino $x \geq y$. Glej sliko.)

Za shranjevanje množice točk $\varphi(B)$, kjer je vsaka točka $b \in B$ asociirana s točko preslikave $\varphi(b)$, lahko uporabimo dvodimenzionalno območno drevo.

Za vsako točko poizvedbe $a \in A$ lahko točke $b \in B$, kjer ab seka σ , dobimo s poizvedbo na območnem drevesu, ki vrne točke $\varphi(B)$ v kvadrantu

$$\{(x, y) \mid \varphi_1(a) < x \text{ and } \varphi_2(a) > y\}.$$

Območna drevesa so bolj podrobno opisana v poglavju x.

□

Lema 3.8. $\Phi_\sigma(A, B)$ lahko izračunamo v času $\mathcal{O}(n \log^4 n)$.

Dokaz. Zgradimo množice $\{B_1, \dots, B_t\}$ in podatkovno strukturo $\mathcal{D}(B)$, opisano v lemi 3.7. Za vsak B_j , kjer $j = 1, \dots, t$, zgradimo podatkovno strukturo, opisano v lemi 3.1. Ker velja $\sum_{i=1}^t |B_i| \log |B_i| = \mathcal{O}(n \log n)$, potrebujemo za to

$$\sum_{i=1}^t \mathcal{O}(|B_i| \log |B_i|) = \mathcal{O}(n \log^2 n)$$

časa.

Obravnavajmo poljubno točko $a \in A$. Naredimo poizvedbo na $\mathcal{D}(B)$, ki nam vrne množico indeksov $I(a)$, za katero velja

$$B(a) = \bigcup_{i \in I(a)} B_i.$$

Za vsak $i \in I(a)$ naredimo poizvedbo s točko a nad podatkovno strukturo, asociirano z B_i , da dobimo

$$b_i^*(a) \in \arg \min \{w_b \mid b \in B_i, |ab| \leq 1\}.$$

Potem velja

$$\min \{w_b \mid b \in B_{\leq 1}(a)\} = \min \{w_{b_i^*(a)} \mid i \in I(a)\},$$

s čimer lahko dobimo $\Phi_\sigma(a, B)$ iz točk $b_i^*(a)$, $i \in I(a)$.

Za točko a potrebujemo $\mathcal{O}(\log^2 n)$ časa, da izračunamo $I(a)$, in nato $\mathcal{O}(\log^2 n)$ časa za vsak indeks i v $I(a)$. Skupna časovna kompleksnost za vsako točko $a \in A$ je tako $\mathcal{O}(\log^4 n)$. □

3.3.3 Sekanje nepremične daljice

Naj bo $G(P)$ neutezen graf nad množico točk P (v nadaljevanju samo G), in naj bo $d_G(r, p)$ dolžina najkrajše poti v G med r in p .

Za vsako točko $r \in P$ naj bo T_r drevo najkrajših poti v G iz r . S $T_r[p]$ označimo pot v T_r iz r v p . E_r naj bo množica povezav v G , ki se ne pojavijo v T_r . Velja torej $E_r = E(G) \setminus E(T_r)$.

Za vsak r in p v P naj bo $\gamma(r, p)$ poligonalna pot, definirana s $T_r[p]$. Za vsak r in vsako povezavo $pq \in E_r$ naj bo $\gamma(r, pq)$ zaprta poligonalna pot, ki jo dobimo z združitvijo $\gamma(r, p)$, pq in potjo, nasprotno $\gamma(r, q)$. Z $\text{len}(r, pq)$ definiramo število povezav v $\gamma(r, pq)$. Tako velja $\text{len}(r, pq) = d_G(r, p) + d_G(r, q) + 1$.

Naj bo σ daljica, za katero predpostavimo, da ne vsebuje nobene točke iz P . Za vsako poligonalno pot γ naj bo $X(\gamma, \sigma)$ število presečišč γ s σ po modulu 2. Za vsak $r, p \in P$ definiramo $N(r, p) = X(\gamma(r, p), \sigma)$ ter za vsak $r \in P$ in $pq \in E_r$ definiramo $N(r, pq) = X(\gamma(r, pq), \sigma)$.

Definirajmo sledeč optimizacijski problem:

$$\begin{aligned} \Psi_\sigma(P) &:= \min \text{len}(r, pq) \\ \text{s.t. } &r \in P \\ &pq \in E_r \\ &N(r, pq) = 1. \end{aligned}$$

Iz rezultatov Cabella in Giannopoulou (referenca!) sledi naslednje:

Izrek 3.9. *Naj bo (r^*, p^*g^*) optimalna rešitev problema $\Psi_\sigma(P)$. Potem so točke v $T_{r^*}[p] \cup T_{r^*}[q]$ optimalna rešitev problema ločevanja s krogi.*

Za izračun $\Psi_\sigma(P)$ iteriramo po točkah r , tako da definiramo tudi

$$\begin{aligned} \Psi_\sigma(r, P) &:= \min d_G(r, p) + d_G(r, q) \\ \text{s.t. } &pq \in E_r \\ &N(r, pq) = 1. \end{aligned}$$

Optimizacijski problem lahko potem definiramo kot

$$\Psi_\sigma(P) = 1 + \min_{r \in P} \Psi_\sigma(r, P).$$

Fiksirajmo nek $r \in P$ in se osredotočimo na rešitev problema $\Psi_\sigma(r, P)$.

Izračunamo drevo najkrajših poti T_r . Skupaj z vsako točko $p \in P$ shranimo $d_G(r, p)$. Nato izračunamo vrednosti $N(r, p)$, $p \in P$ s preprostim prehodom od spodaj navzgor po T_r . Začnemo z inicializacijo $N_r[r] := 0$. Za vsako točko $p \in P$, čigar starš v T_r je p' , potem velja $N_r[p] = N_r[p'] + |\sigma \cap pp'| \pmod{2}$. Če torej pp' seka σ , velja $N_r[p] = N_r[p'] + 1 \pmod{2}$, sicer velja $N_r[p] = N_r[p']$.

Predpostavimo, da je σ navpična daljica s krajišči $(0, 0)$ in $(0, s)$. Iskanje optimalne povezave pq razdelimo na tri primere:

- pq ima obe krajišči na isti strani osi y , za kateri velja $N(r, p) \neq N(r, q)$, ali
- pq ima krajišči na različnih straneh osi y , $N(r, p) \neq N(r, q)$ in pq ne seka σ , ali
- pq ima krajišči na različnih straneh osi y , $N(r, p) \neq N(r, q)$ in pq seka σ .

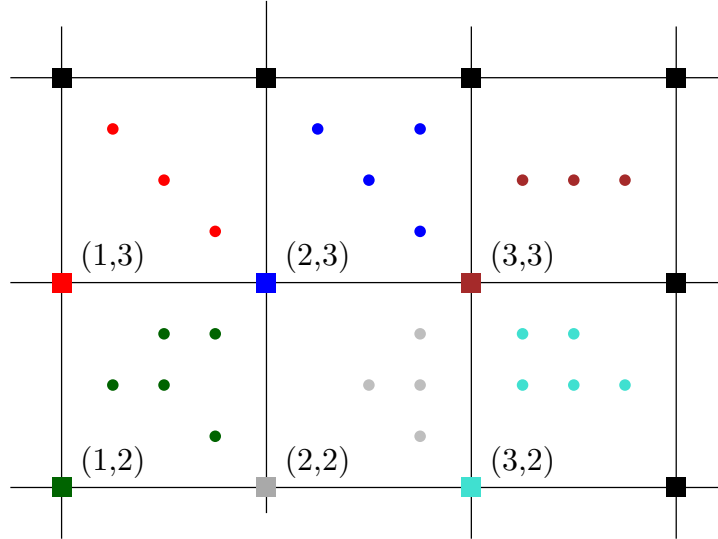
Lema 3.10. $\Psi_\sigma(r, P)$ lahko izračunamo v času $\mathcal{O}(n \log^4 n)$.

Dokaz. Za $i \in \{0, 1\}$ naj bo L_i podmnožica točk iz P levo od osi y in z $N(r, p) = i \pmod{2}$. Očitno velja

$$P = L_0 \cup L_1 \cup R_0 \cup R_1.$$

Naj bosta σ_+ in σ_- poltraka na osi y , ki ju dobimo, če odstranimo σ . Z uporabo $d_G(r, p)$ kot utež točke $p \in P$ potem dobimo

$$\begin{aligned} \Psi_\sigma(r, P) = \min \{ & \Phi(L_0, L_1), \Phi(R_0, R_1), \Phi_{\sigma_+}(L_0, R_1), \Phi_{\sigma_+}(L_1, R_0), \\ & \Phi_{\sigma_-}(L_0, R_1), \Phi_{\sigma_-}(L_1, R_0), \Phi_\sigma(L_0, R_0), \Phi_\sigma(L_1, R_1) \}. \end{aligned}$$



Slika 3.8: .

Vsako posamezno instanco lahko izračunamo s pomočjo lem 3.2 in 3.8v času $\mathcal{O}(n \log^4 n)$. \square

Izrek 3.11. *Problem minimalnega ločevanja s krogi lahko rešimo v času $\mathcal{O}(n^2 \log^4 n)$.*

Dokaz. Za vsak $r \in P$ izračunamo drevo najkrajših poti T_r v G iz r , razdalje $d_G(r, p), p \in P$ in paritete $N(r, p)$. Potem rešimo $\Psi_\sigma(r, P)$ s pomočjo leme 3.10.

TODO

\square

3.4 Alternativni algoritmi za SSSP in minimalno ločevanje

3.4.1 Eksplicitni BFS

3.4.2 BFS z uporabo grafa kvadratne mreže

Graf enotske kvadratne mreže je med drugim uporaben za preiskovanje v širino kot alternativa klasičnemu algoritmu BFS nad grafom enotskih Vsaki

točki iz dane množice P lahko izračunamo njen ključ tako, da poiščemo spodnje levo vozlišče celice grafa kvadratne mreže, v kateri se točka nahaja (glej sliko 3.8). Graf $G(P)$ lahko tako predstavimo s slovarjem, kjer ključi predstavljajo vozlišča grafa kvadratne mreže, kot vrednosti pa hranijo seznam vseh točk v celici, ki se nahaja zgoraj desno od ključa. Preiskovanje v širino nad takim grafom poteka tako, da za dano točko preverimo samo tiste kandidate, ki se nahajajo v isti ali eni izmed sosednjih celic grafa kvadratne mreže, točki pa povežemo, če je njuna razdalja največ 1.

3.4.3 Generični algoritem za ločevanje z diski

Poglavje 4

Implementacija algoritma

4.1 SSSP drevo

Algoritem za izgradnjo drevesa SSSP smo implementirali z mislijo na možnost njegove neposredne uporabe v algoritmu za ločevanje z diski. Posledično smo spustili nekaj funkcionalnosti dreves, ki jih kasneje v programu ne potrebujemo, po drugi strani pa dodali stvari, ki se ne tičejo drevesa, so pa potrebne kasneje. Tako na primer ne moremo dostopati do otrok vozlišč drevesa, po drugi strani pa pri dodajanju točk k drevesu te sproti tudi uvrstimo v eno izmed množic $L0$, $L1$, $R0$ in $R1$.

Za implementacijo smo uporabili razred *SSSPTree* s štirimi zasebnimi atributi. Vsi so sezname točk tipa vector in vsaka hrani eno izmed omenjenih množic. Konstruktor kot vhodne parametre sprejme seznam točk P , točko izvora r in daljico st ter zgradi drevo. Metoda *getAllSets* v seznamu vrne vse štiri attribute.

4.1.1 Dodatki v razredu Point_2

Algoritem zgradi drevo implicitno. To pomeni, da kot rezultat ne dobimo nobene nove strukture, ampak konstruktor vsako točko doda v enega od štirih seznamov in pri tem spremeni vrednost dveh njenih atributov:

- `dist`: hrani razdaljo do korena drevesa in je tipa *unsigned_int*
- `parent`: hrani svojega starša in je tipa deljen kazalec (*ang.* shared pointer) - zakaj že shared???

Noben imed omenjenih atributov ni del razreda *Point_2* v CGAL, zato smo jih sami dodali. S pomočjo kazalca na starša se je tako od vsake točke drevesa možno sprehoditi do korena, nasprotno pa to ne velja.

4.1.2 Voronoijev diagram za iskanje najbližjega soseda

Kot smo omenili pri opisu algoritma v prejšnjem poglavju, pri gradnji množice W_i točke kandidatke testiramo tako, da poiščemo njihove najbližje sosede v množici W_{i-1} . Nad slednjo zgradimo VD, točke kandidatke pa uporabimo za poizvedbe nad tako strukturo.

Vse potrebne objekte in funkcije nam nudi že CGAL. Nad objektom razreda *Voronoi_Diagram_2* lahko delamo poizvedbe s funkcijo `locate(Point_2 q)`, ki vrne objekt tipa *Locate_result*. Za slednjega je potrebno ugotoviti, v katerega izmed treh tipov objekta, ki so del strukture VD, se ga da pretvoriti: lice, vozlišče ali enosmerno povezavo. Ker kot rezultat hočemo vrniti VD središče, ga moramo dobiti prek takega objekta. Razred tipa *Face_handle* ima funkcijo *dual*, ki vrne vozlišče v DT, dualno takemu licu. Prek vozlišča VD lahko pridemo do dualnega lica v DT in nato iteratorja vozlišč, ki ga definirajo. Enosmerna povezava *Halfedge_handle* ima funkciji *up* in *down*, ki vrneta vozlišče v DT, dualno licu v VD nad oziroma pod povezavo.

Da dobimo dejanski objekt tipa *Point_2*, ki definira vozlišče v DT oziroma središče v VD, uporabimo funkcijo *point()*. Na koncu moramo samo še preveriti evklidsko razdaljo med dobljeno točko in točko poizvedbe.

Opisano proceduro smo združili pod funkcijo *query* in jo dodali v našem razredu *VoronoiDiagram*, razširjenem nad *Voronoi_Diagram_2*. Koda funkcije *query* je prikazana spodaj.

```

1  std::tuple<bool, Point_2*> VoronoiDiagram::query(Point_2 q) {
2
3      Locate_result lr = locate(q);
4      // delaunay vertex == voronoi site
5      Delaunay_vertex_handle df;
6      if (Vertex_handle* v = boost::get<Vertex_handle>(&lr)) {
7          // query point coincides with a voronoi vertex
8          // vertex is built based on at most three sites, return the first↔
              one
9          df = (*v)->site(0);
10     }
11     else if (Face_handle* f = boost::get<Face_handle>(&lr)) {
12         // if query point lies on Voronoi face, return DT vertex inside ↔
              that face
13         df = (*f)->dual();
14     }
15     else if (Halfedge_handle* e = boost::get<Halfedge_handle>(&lr)) {
16         // query point lies on Voronoi edge, return the site above it
17         df = (*e)->up();
18     }
19     Point_2 faceSitePoint = df->point();
20     Point_2 *fcp = &df->point();
21     // use squared distance
22     double dist = CGAL::squared_distance(*fcp, q);
23
24     if (dist <= 1) {

```

Kot vidimo v kodi, metoda poleg najbližjega soseda vrne tudi spremenljivo logičnega tipa, ki nam pove, če je razdalja med točko poizvedbe in njenim najbližjim sosedom manjša ali enaka 1.

Naš razred ima še eno dodatno funkcijo. *Voronoi_Diagram_2* omogoča vstavljanje samo objektov tipa *Site_2* (in *Point_2*, ker zna CGAL samodejno pretvarjati med obema tipoma). Za potrebe drevesa SSSP smo v razredu *VoronoiDiagram* omogočili tudi vstavljanje objektov tipa *Delaunay_Vertex_Handle*, iz katerega je moč enostavno dostopati do točke tipa *Point_2*.

4.1.3 Izgradnja drevesa

Konstruktor najprej zgradi DT nad P . Nato za izvirno točko $r \in P$ s pomočjo metode *locate* poišče vozlišče v DT, s katerim sovpada. Pri tem preventivno preveri, da je tip rezultata, ki ga vrne *locate*, resnično

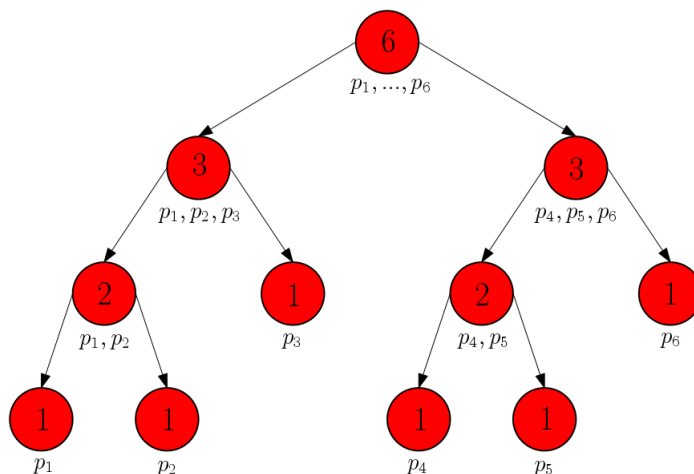
Delaunay_Vertex_Handle, sicer $r \notin P$. Za vse točke v P velja predpostavka, da sta vrednosti njihovih atributov *dist* in *parent* ponastavljeni. Velja torej $\forall p \in P : p.dist = \infty \wedge p.parent = \text{nullptr}$.

Za generatorja točk kandidatk uporabljamo objekt tipa *deque*, primeren za hranjenje elementov v vrsti. Manjša razlika se potem pojavi pri dostopanju do točk v vrsti. V psevdokodi algoritma (vrstica 13) je q poljubna točka v Q , medtem ko je v kodi q vedno prva točka v vrsti (priklicana z metodo *front*, *pop_front* pa jo nato tudi odstrani iz vrste). Za W_i in W_{i-1} v zanki hranimo seznam Delaunayevih vozlišč (objekte tipa *Vertex_Handle*). Voronoijev Diagram tipa *VoronoiDiagram* zgradimo s seznamom W_{i-1} .

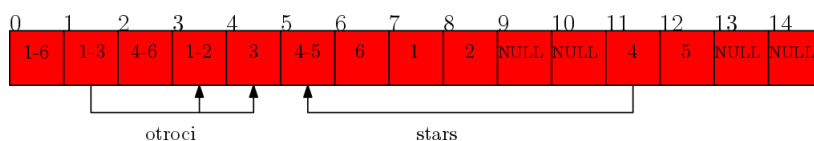
Povezave oziroma sosede točke q najdemo s pomočjo metode v DT *incident_vertices(q)*. Ker ima v CGAL implementaciji DT poleg standardnih vozlišč še eno neskončno vozlišče, ki je sosedno vsem ostalim, takega soseda ne obravnavamo. Za ostale sosede p najprej preverimo, če je njihova razdalja do $r \infty$ (oziroma bolj konkretno, enaka `numeric_limits<int>::max()`), potem pa poiščemo njihovega najbližjega soseda w v W_{i-1} z metodo *query* našega razreda *VoronoiDiagram*. Če $|pw| \leq 1$, naredimo naslednje:

- $p.setDist(i)$
- $Q.insert(p)$
- $W_i.insert(p)$
- $p.setParent(\text{shared_ptr}\langle Point_2 \rangle w)$
- $p.setNr(updateNr(w, p, st))$
- $categorize(p, st)$

Zadnji dve točki nista omenjeni v psevdokodi, ker njuna uporabnost pride v poštev šele kasneje pri separaciji z diski. *updateNr* vrne novo vrednost za atribut *nr* točke p . Funkcija preveri, če daljica pw seka st in če jo, vrne $(w.getNr() + 1) \% 2$, sicer vrne $w.getNr()$. *categorize* točko p na podlagi njenega atributa *nr* in relativnega položaja glede na st (ki je ali levo ali desno) doda v enega od štirih seznamov $l0$, $l1$, $r0$ ali $r1$.



Slika 4.1: Drevo najbližjega soseda, ki hrani 6 točk, prikazano v drevesni strukturi. Vrednosti vozlišč ponazarjajo število točk, shranjenih v objektu, ki omogoča poizvedbe najbližjega soseda, pod vozlišči pa so nanizane konkretne točke.



Slika 4.2: Drevo najbližjega soseda, ki hrani 6 točk, predstavljeno v obliki seznama. Za vsako celico so prikazani indeksi točk, ki jih hrani objekt v njej.

4.2 Drevo najbližjega soseda

Drevo najbližjega soseda, opisano v poglavju 3.3, bi lahko implementirali s kazalci ali seznamom. Odločili smo se slednjega. Seznam hrani kazalce (*shared_ptr*) na objekte podatkovne strukture $DS(P(\nu))$ za poizvedbe najbližjega soseda. Velikost seznama, ki hrani n točk, je enaka $2^{\lceil \log_2 n \rceil + 1} - 1$. Nekatera vozlišča na najnižjem nivoju drevesa so lahko tudi prazna (torej brez objektov). Primer takega drevesa je prikazan na sliki 4.1. Otroka vozlišča, ki se v seznamu nahajata na mestu i , se nahajata na mestu $2i + 1$ in $2i + 2$. Podobno se starš vozlišča na mestu i nahaja na mestu $\lfloor (i - 1)/2 \rfloor$.

Za podatkovno strukturo s poizvedbami najbližjega soseda smo sprva ho-

teli uporabiti VD, vendar smo se kasneje odločili za Kd drevo. Razlog za to je implementacija VD v knjižnici CGAL. Ker je prostorska kompleksnost VD $\mathcal{O}(n)$, bi pričakovali, da poraba prostora raste linearno z večanjem VD. Izkaže se, da je rast počasnejša od linearne, kar pomeni, da na primer 100 VD objektov velikosti 10 porabi več prostora kot 10 objektov velikosti 100. Posledično poraba prostora ni enaka za vsak nivo drevesa, temveč z globino raste. Razlike se v celotnem algoritmu potem še potencirajo, ker zgradimo VD v (skoraj) vsakem vozlišču drevesa najbližjega sosedu, vsako tako drevo pa v vsakem vozlišču sekundarnega drevesa v območnem drevesu. Ko smo testirali implementacijo Kd dreves v CGAL, smo ugotovili tudi, da je čas konstrukcije bistveno krajši kot pri VD. Primerjave med konstrukcijama obeh podatkovnih struktur (procesorski čas in poraba RAM-a) so prikazane v poglavju Rezultati.

4.2.1 Kd drevo

Kot osnovo smo za poizvedbe najbližjega sosedu uporabili funkcijo *search* v razredu *Kd_tree*, ki kot argument sprejme *OutputIterator*, kamor se shranjujejo objekti, ki jih poizvedba vrne, in *FuzzyQueryItem*, ki je v dvodimenzionalnem primeru lahko krog ali pravokotnik in določa območje iskanja. Kot argument lahko sprejme vrednost ϵ , ki določa stopnjo mehkosti (ang. fuzzyness) in se jo uporablja pri aproksimacijskih poizvedbah, ki pa jih v našem algoritmu nismo potrebovali. Časovna kompleksnost funkcije *search* je enaka $\mathcal{O}(\log n + k)$, kjer je k število vrnjenih točk znotraj območja iskanja. Ker naš algoritem zahteva $\mathcal{O}(\log n)$, smo razredu *Kd_tree* dodali podobno funkcijo, ki pa iskanje zaključi v istem trenutku, ko ugotovi, da se vsaj ena točka nahaja znotraj območja iskanja. Če na primer za neko notranje vozlišče Kd drevesa ugotovi, da vse točke v njegovem poddrevesu ustrezajo kriterijem iskanja, jih ne doda enega po enega v *OutputIterator*, temveč doda samo "dummy" točko in zaključi. Če najde samo eno točko (ko pride do lista Kd drevesa), potem tako točko tudi vrne. To si lahko privoščimo, ker nas v notranjih vozliščih drevesa najbližjega sosedu ne zanimajo konkre-

tne točke, ampak samo informacija, ali se vsaj ena točka nahaja v območju iskanja. Šele ko pridemo do lista NN drevesa, nas zanima konkretna točka. S tem, ko je vedno vrnjena ena točka, se časovna kompleksnost spremeni v $\mathcal{O}(\log n)$.

4.3 Implementacija optimizacijskega problema

$$\Phi_\sigma(A, B)$$

Za rešitev optimizacijskega problema, predstavljenega v poglavju 3.3.2, smo uporabili implementacijo območnih dreves v CGAL, za uporabo dualnosti pa napisali lasten razred (bolj natančno *struct*) *DualPoint* za točke v dualnem prostoru.

4.3.1 DualPoint - implementacija dualne točke

Odločili smo se razlikovati po tipu med navadnimi točkami in točkami v dualnem prostoru, ki izhajajo iz premic v primarnem prostoru. Razred *DualPoint* hrani dva atributa. Prvi je točka v dualnem prostoru tipa *Point_2*, poimenovan *point*. Drugi je kazalec na točko tipa *Point_2*, poimenovan *originalPoint*. Če *point* predstavlja točko $(\varphi_1(b), \varphi_2(b))$ (glej poglavje 3.3.2), potem *originalPoint* predstavlja točko b . Konstruktor razreda kot argumenta sprejme točko b in daljico σ in s pomočjo funkcije *imagePoint* izračuna točko preslikave.

4.3.2 Območno drevo

Za dvodimenzionalna območna drevesa smo uporabili implementacijo v knjižnici CGAL, ki jo predstavlja razred *Range_tree_2*. Vsako vozlišče hrani par $\langle \text{ključ}, \text{vrednost} \rangle$. Vrednost dodatno opisuje vozlišče in je lahko tudi prazna, ključ pa se uporablja pri izgradnji drevesa in poizvedbah. Oba argumenta sta v razredu podana kot *template*, kar pomeni, da moramo pri

konstrukciji drevesa zanju podati konkretna tipa. Za ključ smo uporabili naš razred *Dual_Point*, kot vrednost pa vsako vozlišče hrani svoje drevo NN.

Pri inicializaciji drevesu podamo seznam parov dualnih točk in praznega drevesa NN. Nato se iterativno sprehodimo po prvem nivoju drevesa (primarnem drevesu) - tako kot pri preiskovanju v širino - s funkcijo *traverse_and_populate_with_data*. Za vsako vozlišče pokličemo funkcijo *build_NN_Tree_on_layer2*, ki rekurzivno od spodaj navzgor za vsako vozlišče v sekundarnem drevesu, na katerega kaže vozlišče v primarnem drevesu, vstavi v (v tistem trenutku prazno) njegovo drevo NN točke, ki jih hranita drevesi NN njegovih otrok. Pri listih je v drevo NN vstavljena točka, čigar dualna točka je uporabljena kot ključ vozlišča. Rekurzivni klic funkcije se torej kliče pred vstavljanjem točk v drevo NN. S pomočjo rekurzije je vsakemu notranjemu vozlišču v sekundarnem drevesu seznam točk v njegovem poddrevesu, ki ga potrebujemo za izgradnjo njegovega drevesa NN, implicitno podan preko dreves NN njegovih otrok. Če bi drevesa NN gradili od zgoraj navzdol v sekundarnem drevesu, bi morali za vsako vozlišče narediti sprehod po njegovem poddrevesu. V primarnem drevesu so vsa drevesa NN prazna.

Poizvedbe v območnem drevesu

Razredu *Range_tree_d* smo za poizvedbe dodali funkcijo *window_query_impl_modified*, ki kot osnovo uporablja obstoječo funkcijo *window_query_impl*. Argumenti funkcije so *OutputIterator*, kamor se vstavlja vrnjene točke, interval I , ki ga definirata dve dualni točki, tretji argument, ki ga v obstoječi funkciji ni, pa je točka a , ki jo uporabimo za poizvedbe najbližjega sosedu v drevesih NN. Pri dvodimenzionalnih območnih drevesih je geometrijska predstavitev intervala pravokotnik; levo krajišče intervala ustreza spodnjemu levemu oglišču pravokotnika, desno krajišče pa zgornjemu desnemu oglišču. Interval je zaprt.

Če za vozlišče w v primarnem drevesu ugotovimo, da njegovo celotno poddrevo ustreza pogoju poizvedbe, nadaljujemo s preiskovanjem v njegovem sekundarnem drevesu. Če za vozlišče v v sekundarnem vozlišču ugotovimo, da so vsa vozlišča v njegovem poddrevesu vsebovana v intervalu I , ne do-

4.3. IMPLEMENTACIJA OPTIMIZACIJSKEGA PROBLEMA $\Phi_\sigma(A, B)$

damo vso poddrevo v rezultat, kot je to storjeno v osnovni funkciji, temveč opravimo še poizvedbo najbližjega soseda točke a v drevesu NN vozlišča v . Za vsako obiskano vozlišče w v primarnem drevesu prav tako preverimo, če je vsebovano v intervalu I . Če je temu tako, točko, iz katere izhaja dualna točka, ki predstavlja ključ vozlišča w , dodamo v seznam vrnjenih točk. Enako velja za obiskana vozlišča v sekundarnem drevesu. [To pomeni vsi listi v sekundarnem drevesu. Naj listi sploh hranijo kd drevo velikosti 1?]

```

1  template <class OutputIterator , class FuzzyQueryItem>
2  OutputIterator search_exists(OutputIterator it , const FuzzyQueryItem& q←
   , Kd_tree_rectangle<FT, D>& b) const
3  {
4      if (is_leaf()) {
5          Leaf_node_const_handle node = static_cast<Leaf_node_const_handle←
              >(this);
6          if (node->size()>0)
7              for (iterator i = node->begin(); i != node->end(); i++)
8                  if (q.contains(*i))
9                      {
10                         *it = *i; ++it;
11                         break;
12                     }
13     }
14     else {
15         Internal_node_const_handle node = static_cast<←
             Internal_node_const_handle>(this);
16         // after splitting b denotes the lower part of b
17         Kd_tree_rectangle<FT, D> b_upper(b);
18         b.split(b_upper , node->cutting_dimension() , node->cutting_value←
             ());
19
20         if (q.outer_range_contains(b)) {
21             *it = Point_d(0, 0); ++it;
22             //it = node->lower()->tree_items(it);
23         }
24         else
25             if (q.inner_range_intersects(b))
26                 it = node->lower()->search_exists(it , q , b);
27
28         if (q.outer_range_contains(b_upper)) {
29             *it = Point_d(0,0); ++it;
30             //it = node->upper()->tree_items(it);
31         }

```

```

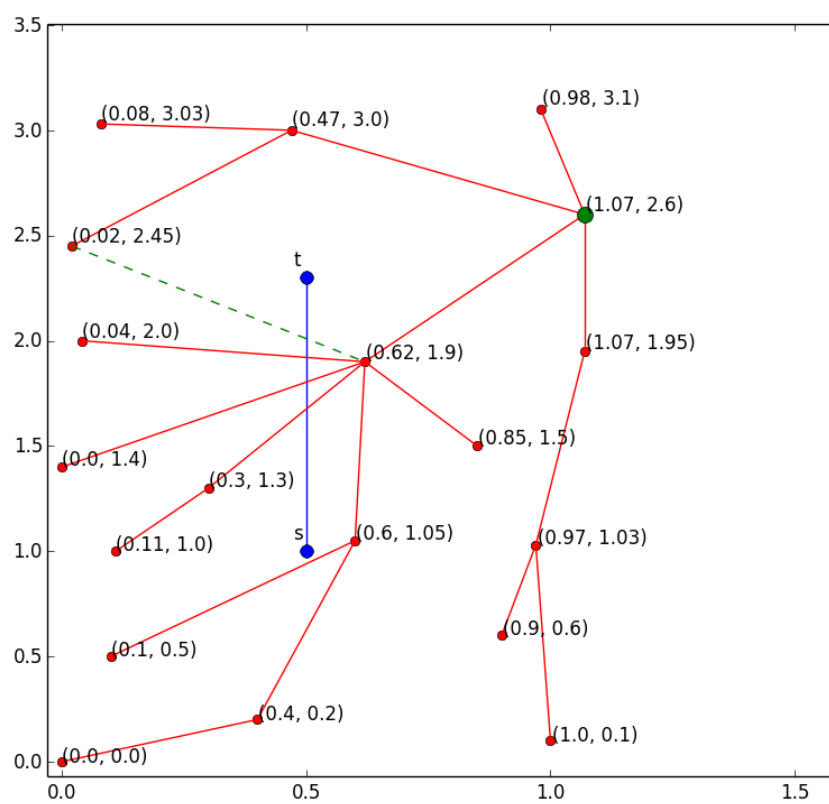
32         else
33             if (q.inner_range_intersects(b_upper))
34                 it = node->upper()->search_exists(it, q, b_upper);
35     };
36     return it;
37 }

```

```

1 tuple<bool, Point_2> NNTree::search(Point_2 q) {
2     //tuple<bool, Point_2*> res = make_tuple(true, new Point_2(0, 0));
3     tuple<bool, Point_2> res = query(q, 0);
4     if (std::get<0>(res) == false) {
5         return res;
6     }
7
8     int i = 0;
9     while (2*i+1 < A.size()) {
10         res = query(q, 2 * i + 1);
11         if (get<0>(res) == true)
12             i = 2 * i + 1;
13         else
14             i = 2 * i + 2;
15     }
16     // same value as in last loop iteration, if leaf node is left child
17     res = query(q, i);
18     return res;
19 }
20
21 tuple<bool, Point_2> NNTree::query(Point_2 q, int idx)
22 {
23     Fuzzy_circle exact_range(q, 1);
24     list<Point_2> result;
25     A[idx]->search_exists(back_inserter(result), exact_range);
26     if (result.size() == 0) {
27         return tuple<bool, Point_2> {false, Point_2(0, 0)};
28     }
29     else {
30         Point_2 first = result.front();
31         return tuple<bool, Point_2> {true, first};
32     }
33 }

```

Slika 4.3: Dodaj primer množic L_0 , L_1 , R_0 , R_1 .

4.4 Minimalno ločevanje

Za rešitev optimizacijskega problema $\Psi_\sigma(r, P)$ (glej lemo 3.10) smo napisali dve funkciji, ki rešita optimizacijska problema $\Phi(A, B)$ in $\Phi_\sigma(A, B)$.

4.4.1 opt1

Funkcija $findMinPair(S_i, S_{i+1}, currentMinWeight)$ reši optimizacijska problema $\Phi(L_0, L_1)$ in $\Phi(R_0, R_1)$. Parameter $currentMinWeight$ hrani trojko $(a, b, w_a + w_b)$, kjer točki a in b s povezavo ab tvorita trenutno najbolj optimalno pot.

Nad točkami $p \in S_{i+1}$ zgradimo drevo $NN(S_{i+1})$. Za vsako točko $q \in S_i$ naredimo poizvedbo najbližjega soseda v drevesu, če je razdalja q do korena drevesa $SSSP$ manjša od dolžine trenutno najbolj optimalne poti. Za točko b , ki jo vrne poizvedba, preverimo, če je vsota uteži $w_q + w_b$ manjša od $w_a + w_b$. Če je temu tako, posodobimo $currentMinWeight$ z $(q, b, w_q + w_b)$.

4.4.2 opt2

Funkcija $findminpairRi(L_i, L_j, R_i, currentMinWeight, st)$. Šest instanc problema $\Phi_\sigma(A, B)$ v $\Psi_\sigma(r, P)$ ne rešujemo neodvisno, ker bi sicer morali šestkrat konstruirati območno drevo. Probleme z isto podmnožico R_i združimo pod skupni problem, ki ga s funkcijo rešimo na sledeč način.

Zgradimo območno drevo s točkami $p \in R_i$. Za vsako točko $p_i \in L_i$ naredimo poizvedbo v območnem drevesu, pri čemer za območje iskanja podamo drugi kvadrant koordinatnega sistema, ki ima za izhodišče točko $\varphi(p_i)$. Levo krajišče tako predstavlja točka $(\varphi_1(p_i), -\infty)$, desno pa točka $(\infty, \varphi_2(p_i))$. Parameter st predstavlja daljico σ , s katero lahko izračunamo dualne točke. Parameter $currentMinWeight$ hrani trojko $(a, b, w_a + w_b)$, kjer točki a in b s povezavo ab tvorita trenutno najbolj optimalno pot, izračunano z do zdaj rešenimi optimizacijskimi problemi Φ in Φ_σ . Za vse točke q_k , ki jih vrne poizvedba, preverimo, če je vsota uteži $w_{p_i} + w_{q_k}$ manjša od $w_a + w_b$. Če je temu tako, posodobimo $currentMinWeight$ z $(p_i, q_k, w_{p_i} + w_{q_k})$.

Funkcijo smo optimizirali tako, da smo zmanjšali število poizvedb. Poizvedb ne delamo s točkami, pri katerih je razdalja do korena drevesa SSSP večja od vsote razdalj točk a in b , ki tvorita trenutno optimalno zaprto pot. S poizvedbami nad drugim kvadrantom rešimo problem $\Phi_\sigma(L_i, R_i)$.

Funkcija potem nadaljuje s poizvedbami za vsako točko $p_j \in L_j$. Za vsak p_j opravimo dve poizvedbi. Pri eni za območje iskanja podamo prvi kvadrant, pri drugi pa tretji kvadrant koordinatnega sistema, ki ima za izhodišče točko $\varphi(p_j)$. Za vse točke q_l , ki jih vrnete poizvedbi, nato zopet preverimo vsoto uteži $w_{p_j} + w_{q_l}$ in po potrebi posodobimo *currentMinWeight*.

S poizvedbami nad prvim kvadrantom rešimo problem $\Phi_{\sigma_+}(L_j, R_i)$, s poizvedbami nad tretjim kvadrantom pa problem $\Phi_{\sigma_-}(L_j, R_i)$. Glej tudi sliko 3.6. Območje iskanja v teoretičnem pogledu ne določajo samo kvadranti, temveč tudi polravnina $x \geq y$. Ampak presek obeh območij ni pravotnik, za območje iskanja v območnem drevesu pa ne moremo vzeti poljubnega poligona. Omejitev območja iskanja tudi ne prinaša nobenih prednosti. Vse točke $b \in B$ se nahajajo v omenjeni polravnini, točke na drugi strani polravnine, ki se nahajajo v prvem, drugem ali tretjem kvadrantu, pa pripadajo množici točk na levi strani osi y , ki niso shranjene v območnem drevesu.

Funkcija se izvede dvakrat, kar pomeni, da tudi območno drevo zgradimo dvakrat; enkrat za R_0 in enkrat za R_1 . Za točke v L_0 in L_1 naredimo tri poizvedbe; po enkrat v prvem, drugem in tretjem kvadrantu.

4.4.3 Kreiranje cikla

Ko dobimo par točk $a, b \in P$, ki določata minimalno zaprto pot, ki ločuje točki s in t , v seznam shranimo vse točke, ki tvorijo cikel. S pomočjo funkcije *getParent()* lahko za vsako točko p v drevesu $SSSP(r, P)$ dobimo njegovega starša in s tem pot $T_r[p]$ v obratnem vrstnem redu (z začetkom pri točki p in koncem pri točki r), ki jo označimo kot $T_r^{-1}[p]$. Pot $T_r[p]$ brez točke r označimo z $T_{r-1}[p]$. Cikel lahko potem dobimo z združitvijo dveh poti $T_r^{-1}[a] + T_{r-1}[b]$, s čimer dobimo pot (a, \dots, r, \dots, b) . Vrnjen seznam predstavlja cikel, v katerem je začetna točka shranjena samo enkrat. Za dolžino najbolj

optimalne poti se do konca izvajanja programa hrani vsota dolžin točk a in b . Dolžina, ki jo program vrne kot rezultat, je povečana za 1.

4.5 Obravnava posebnih vhodnih primerov

Pri algoritmu ločevanja z diski smo večkrat omenili, da je za daljico σ privzeto, da je vertikalna. Če želimo uporabiti algoritem na primeru, kjer daljica ni vertikalna, je potrebno vse vhodne točke predhodno rotirati

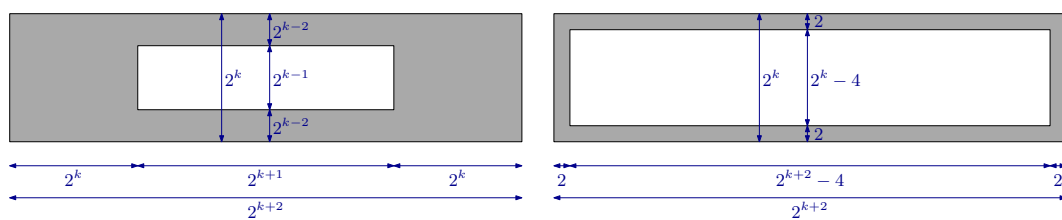
točke, ki ležijo na y osi, grejo k b (preveri tip neskončne vrednosti v `dualpoint`)

Poglavje 5

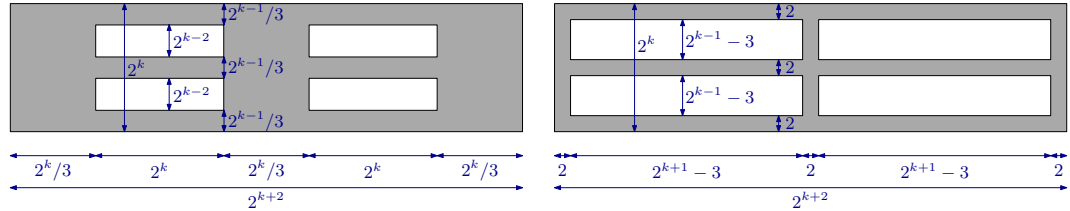
Rezultati

5.1 Drevo najkrajših poti

5.2 Minimalno ločevanje



Slika 5.1: .



Slika 5.2: .

number of točk	50k	100k	250k	500k	750k	1mio
CPU(VD) [s]	0.0859	0.196	0.562	1.2667	2.0495	2.9304
CPU(kd drevo) [s]	0.0025	0.0047	0.0126	0.025	0.0375	0.0497

Tabela 5.1: Časi konstrukcij Voronoijevega diagrama in kd drevesa v sekundah. Vsaka konstrukcija se je izvedla desetkrat, na podlagi tega pa se je izračunal povprečni čas izvedbe.

number of točk	50k	100k	250k	500k	750k	1mio
RAM(VD) [mb]	7.2	14.4	34.3	68.9	103.6	137.9
RAM(kd drevo) [mb]	1.9	3.8	9.5	19	28.5	38.1

Tabela 5.2: Velikost prostora v megabajtih, ki ga zasede en objekt Voronoijevega diagrama ali kd drevesa.

Pravokotnik brez lukenj		20K točk		
dimenzije pravokotnika		4×1	8×2	16×4
predprocesiranje SSSP		0.025	0.027	0.025
povprečje za koren s SSSP		0.131	0.130	0.127
predprocesiranje BFS		25.057	20.433	17.773
povprečje za koren z BFS		3.406	1.359	0.404
mreža		1.605	1.647	0.695
		32×8	64×16	128×32
predprocesiranje SSSP		0.025	0.025	0.026
povprečje za koren s SSSP		0.126	0.129	0.136
predprocesiranje BFS		17.734	17.347	17.179
povprečje za koren z BFS		0.088	0.025	0.009
mreža		0.227	0.089	0.053
		50K točk		
		4×1	8×2	16×4
predprocesiranje SSSP		0.091	0.091	0.070
povprečje za koren s SSSP		0.592	0.562	0.375
predprocesiranje BFS		>3min	159.812	144.965
povprečje za koren z BFS	omejitev RAM-a		9.378	2.789
mreža		11.567	13.660	4.592
		32×8	64×16	128×32
predprocesiranje SSSP		0.071	0.070	0.069
povprečje za koren s SSSP		0.377	0.372	0.366
predprocesiranje BFS		140.404	131.854	132.475
povprečje za koren z BFS		0.584	0.144	0.044
mreža		1.346	0.432	0.187

Tabela 5.3: Times for shortest paths in Pravokotniks without luknje.

Pravokotnik 1 small hole		10K točk		
dimenzije pravokotnika	4×1	8×2	16×4	
predprocesiranje SSSP	0.021	0.012	0.012	
povprečje za koren s SSSP	0.104	0.059	0.060	
predprocesiranje BFS	8.500	4.300	4.100	
povprečje za koren z BFS	1.183	0.318	0.091	
mreža	0.486	0.513	0.168	
	32×8	64×16	128×32	
predprocesiranje SSSP	0.012	0.012	0.012	
povprečje za koren s SSSP	0.061	0.064	0.070	
predprocesiranje BFS	4.100	4.000	4.000	
povprečje za koren z BFS	0.026	0.008	0.003	
mreža	0.072	0.035	0.026	
		20K točk		
	4×1	8×2	16×4	
predprocesiranje SSSP	0.027	0.026	0.025	
povprečje za koren s SSSP	0.142	0.137	0.136	
predprocesiranje BFS	24.813	19.817	18.396	
povprečje za koren z BFS	3.253	1.328	0.467	
mreža	2.181	2.627	0.668	
	32×8	64×16	128×32	
predprocesiranje SSSP	0.025	0.025	0.025	
povprečje za koren s SSSP	0.136	0.138	0.145	
predprocesiranje BFS	17.976	17.542	17.313	
povprečje za koren z BFS	0.108	0.031	0.011	
mreža	0.262	0.104	0.060	

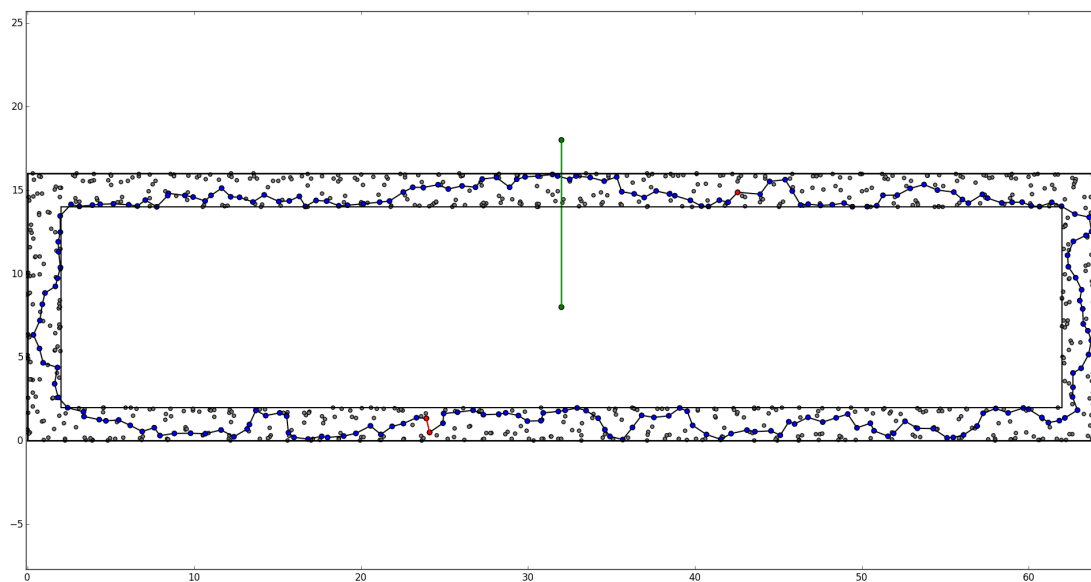
Tabela 5.4: Times for shortest paths in Pravokotniks with a small hole.

Pravokotnik 4 small luknje		10K točk		
dimenzije pravokotnika	32×8	64×16	128×32	
SSSP preprocessing	0.021	0.018	0.018	
povprečje za koren s SSSP	0.056	0.058	0.064	
predprocesiranje BFS	6.291	6.102	6.364	
povprečje za koren z BFS	0.033	0.010	0.004	
mreža	0.064	0.031	0.023	
20K točk				
predprocesiranje SSSP	0.027	0.026	0.026	
povprečje za koren s SSSP	0.125	0.126	0.131	
predprocesiranje BFS	18.325	17.887	17.256	
povprečje za koren z BFS	0.102	0.031	0.010	
mreža	0.230	0.096	0.055	

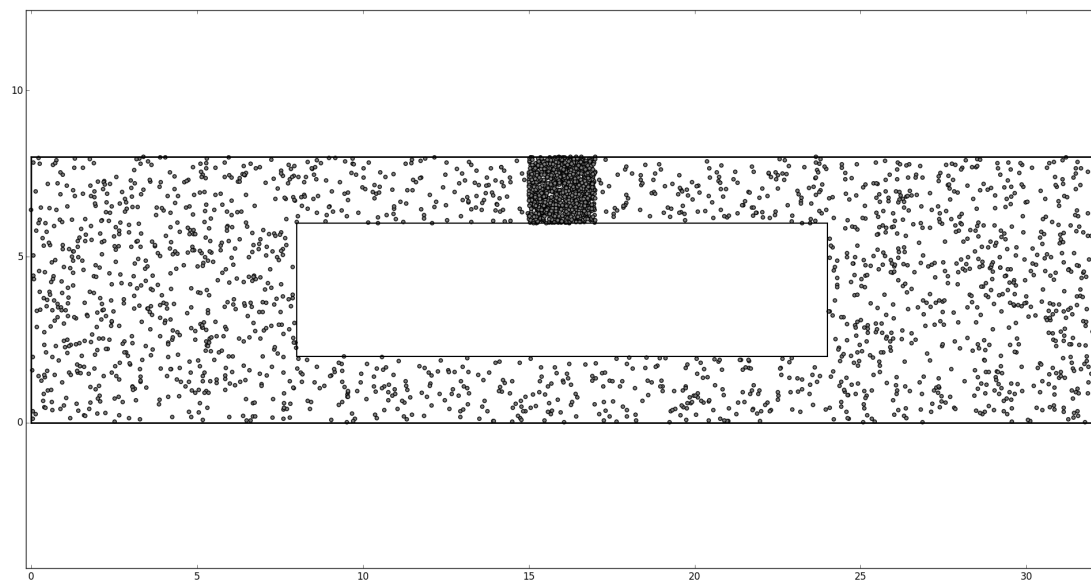
Tabela 5.5: Časi za najkrajše poti v pravokotnikih s 4 manjšimi luknjami.

Pravokotnik 4 večje luknje		5K točk		
dimenzije pravokotnika	32×8	64×16	128×32	
predprocesiranje SSSP	0.007	0.009	0.009	
povprečje za koren s SSSP	0.027	0.028	0.026	
predprocesiranje BFS	1.420	1.420	1.390	
povprečje za koren z BFS	0.007	0.003	0.002	
mreža	0.019	0.013	0.010	
10K točk				
predprocesiranje SSSP	0.012	0.018	0.019	
povprečje za koren s SSSP	0.057	0.054	0.053	
predprocesiranje BFS	5.740	5.660	5.720	
povprečje za koren z BFS	0.028	0.013	0.007	
mreža	0.060	0.038	0.027	

Tabela 5.6: Časi za minimalno ločevanje s 4 večjimi luknjami.



Slika 5.3: .



Slika 5.4: .

Pravokotnik 1 manjša luknja		2K točk			
dimenzije pravokotnika	8×2	16×4	32×8	64×16	
nov algoritem za ločevanje	65	64	53	38	
generičen algoritem	215	87	43	29	

Tabela 5.7: Časi za minimalno ločevanje z eno manjšo luknjo.

Pravokotnik 4 luknje		2K točk, manjše luknje		
dimenzije pravokotnika	32×8	64×16	128×32	
nov algoritem za ločevanje	29	35	35	
generičen algoritem	80	40	30	
		5K točk, večje luknje		
nov algoritem za ločevanje	413	451	388	
generičen algoritem	416	266	206	

Tabela 5.8: Časi za minimalno ločevanje s 4 luknjami.

Poglavje 6

Sklepne ugotovitve

Literatura

- [1] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd ed. edition, 2008.
- [2] S. Cabello and P. Giannopoulos. The complexity of separating points in the plane. *Algorithmica*, 74(2):643–663, 2016.
- [3] Q. Fang, J. Gao, and L. J. Guibas. Locating and bypassing routing holes in sensor networks. In *Proc. Mobile Networks and Applications*, volume 11, pages 187–200, 2006.
- [4] Y. Wang, J. Gao, and J. S. Mitchell. Boundary recognition in sensor networks by topological methods. In *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking*, MobiCom '06, pages 122–133, 2006.