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 2017

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

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 je časovna zahtevnost algoritma enaka $\mathcal{O}(n \log n)$.

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

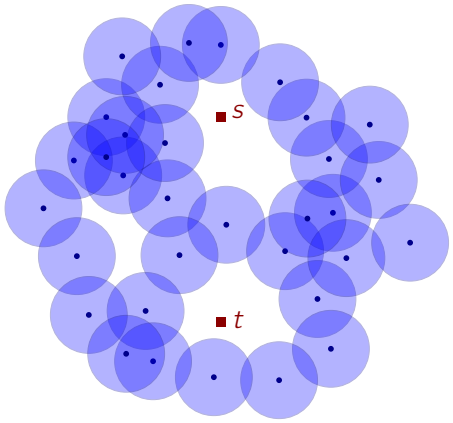
$$\begin{aligned} \min \quad & |\mathcal{D}'| \\ \text{tako da} \quad & \mathcal{D}' \subset \mathcal{D} \\ & \mathcal{D}' \text{ ločuje } s \text{ in } t. \end{aligned}$$

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 kompromis med enostavnostjo in natačnostjo, 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) [8]. 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 [6].

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. Zanjo 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 - tokrat z implementacijskega vidika - ter vse razširitve in spremembe, ki smo jih dodali h knjižnici CGAL. 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.

Poglavje 2

CGAL in uporabljene matematične strukture

V tem poglavju so na kratko opisani programska knjižnica CGAL in podatkovne strukture, uporabljene v naših algoritmih. Za bolj podroben opis glej (citiraj van kreveld, 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 ($Point_2$) $https://doc.cgal.org/4.5/Kernel_23/classKernel.html$

2.1.1 Številski tipi

http://doc.cgal.org/latest/Number_types/index.html# Chapter_Number_Types

2.1.2 Jedro knjižnice

Jedro sestavljajo nespremenljivi geometrijski primitivni objekti konstantne velikosti (točke, vektorji, daljice, trikotniki...) ter funkcije in predikati, ki jih lahko kličemo nad njimi. Poleg tega ponuja osnovne operacije, kot so računanje razdalje, afina transformacija in zaznava presečišč.

Eden od osnovnih problemov, s katerimi se soočimo pri implementaciji geometrijskih algoritmov, je eksaktno računanje z realnimi števili. Zaokroževanje pri uporabi aritmetike nad števili, predstavljenimi s plavajočo vejico, lahko privede do nekonsistentnih odločitev in nepričakovanih napakah že pri nekaterih osnovnih geometrijskih algoritmih. CGAL ponuja izbiro številskih tipov in aritmetike. Pri uporabi eksaktne aritmetike dobimo točne

rezultate, a po drugi strani se čas izvajanja in prostorska poraba algoritma povečata.

2.1.2.1 Predstavitev objektov jedra

Vsi objekti jedra so predloge s parametrom, ki omogoča uporabniku izbiro predstavitve objektov. Dve družini modelov predstavitve temeljita na kartezični predstavitvi točk, dve pa na homogenični predstavitvi. Pri kartezični predstavitvi je vsaka točka predstavljena s kartezičnimi koordinatami, ki privzamejo d-dimenzionalni afini Evklidski prostor s koordinatnim izhodiščem in d pravokotnimi osmi. Točke (in vektorji) so potem predstavljeni z d-terko $(c_o, c_1, ..., c_{d-1})$. Pri homogeni predstavitvi so točke predstavljene z $(d+1)$-terko $(h_o, h_1, ..., h_d)$. V kartezično predstavitev jih lahko pretvorimo s formulo $c_i = h_i/h_d$, ena od možnih homogenih predstavitev točke s kartezičnimi koordinatami pa je $(c_0, c_1, ..., c_{d-1}, 1)$ (ker homogene koordinate niso enolične). Prednost uporabe homogenih koordinat v CGAL-u je, da se z njimi izognemo uporabi deljenja. Vmesnik objektov v jedru je napisan tako, da hkrati omogoča uporabo kartezične in homogene predstavitve.

2.1.2.1.1 Številski tipi

Obe družini predstavitve objektov sta nadalje parametrizirani s številskim tipom za koordinate. CGAL ponuja dva tipa predloge za številske tipe. Eden se nanaša na algebrsko strukturo kolobar, kjer je možno seštevati, odštevati in množiti, drugi pa na polje, kjer je poleg omenjenih operacij možno tudi deliti. Vgrajeni tip *int* tako spada v prvo skupino, ker operacija deljenja ni inverz množenja. leda, gmpz (You might use limited precision integer types like int or long, use double to present your integers (they have more bits in their mantissa than an int and overflow nicely), or an arbitrary precision integer type like the wrapper Gmpz for the GMP integers, leda_integer, or MP_Float.)

Če za vhodne podatke uporabljamo kartezične koordinate, bo veliko geo-

metrijskih računanj vsebovalo samo vhodne vrednosti. To velja med drugim za taka računanja, ki vrednotijo predikate, pri katerih ne gre za nič drugega kot računanje determinante. Primeri so računanje triangulacije in konveksne ovojnice. V takih primerih je dovolj uporabiti kartezično predstavitev točk, celo s kolobarskim številskim tipom. Če pa algoritmi obsegajo tudi računanja novih točk, prihaja do uporabe deljenja, zato moramo uporabiti kartezično predstavitev s številskim tipom za polja, recimo *double*, ali homogeno predstavitev. *double* sicer ni popolnoma natančen, a ga ponavadi uporabljajo tisti, ki jim je pomembna hitrost in se zadovoljijo tudi s približnimi rezultati.

Algoritmi in podatkovne strukture v osnovni knjižnici CGAL so med drugim parametrizirani z razredom geometrijskih značilnosti, ki zajema objekte, nad katerimi algoritmi in strukture delujejo. Za večino slednjih je dovolj za to uporabiti jedro.

2.1.2.2 Geometrija in predikati jedra

CGAL razlikuje med točkami, vektorji in smermi. Točka je točka v ekvklidskem prostoru E^d, vektor je razlika dveh točk p_1, p_2 ter označuje smer in razdaljo od p_1 do p_2 v vektorskem prostoru $\mathbb{R}^d$, smer pa je vektor, pri katerem pozabimo na njegovo dolžino. Gre za različne matematične koncepte, ki se pri afinih transformacijah ne obnašajo enako. Poleg tega so v CGAL-u na voljo premice, poltraki, daljice, ravnine, trikotniki, tetraedri, krožnice, sfere, pravokotniki in kvadri.

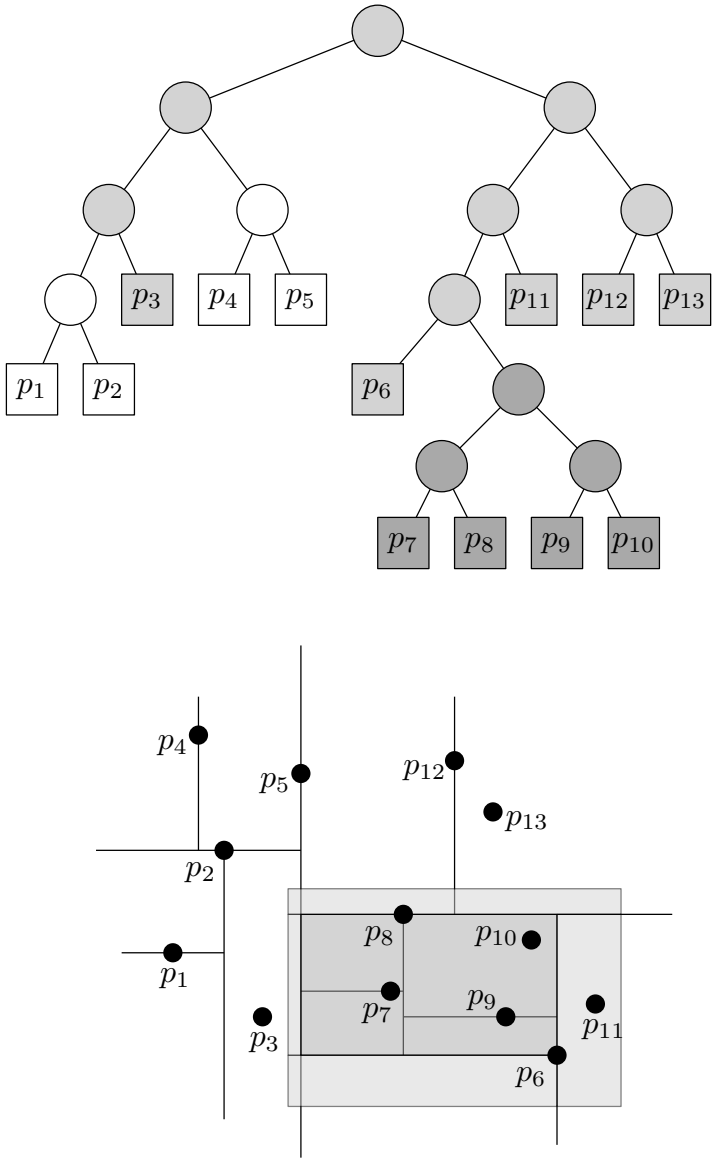
Dve poljubni točki na premici definirata njeno orientacijo ter razdelitev ravnine na pozitivno in negativno stran. Poltrak je orientiran iz smeri njegovega končnega vozlišča, vozlišča daljic pa so urejena in definirajo isto orientacijo kot premica, na kateri daljice ležijo. Vsi geometrijski objekti imajo metodo za testiranje relativne lokacije dane točke glede na objekt. Ker so objekti in njihove meje predstavljene z istim tipom (na primer sfera in krogla ali krožnica in krog), svojo okolico razdelijo na omejen in neomejen del (krožnica) ali dva neomejena dela (hiperravnina). Vsi objekti so privzeto orientirani, kar pomeni, da je eden od delov označen kot pozitivna, drugi pa

kot negativna stran. Prej omenjena metoda torej določa, ali se dana točka nahaja na pozitivni strani, negativni strani ali na meji objekta.

Predikati so osrčje jedra, saj predstavljajo osnovno enoto pri izgradnji algoritmov, zato je njihova pravilnost ključnega pomena pri pravilni implementaciji. Predikati v CGAL-u ne vračajo nujno *boolean* vrednosti, ampak tudi vrednosti tipa *enum*. Na voljo so predikati za orientacijo množice točk, primerjavo točk glede na dano zaporedje, primerjavo razdalj in testi za vsebovanost točke v krogu. Vse ostale metode, ki niso tipa *boolean* ali *enum*, se imenujejo konstrukcije, ker vsebujejo računanja novih numeričnih vrednosti, ki so lahko nenatančna, če ni uporabljeno jedro z eksaktnim številskim tipom. Primer take metode, ki je na voljo za večino geometrijskih objektov, je afina transformacija. Druga dva tipična primera sta računanje presečišč med objekti in kvadrata razdalje.

2.2 Kd drevesa

Kd drevo je podatkovna struktura za organiziranje k-dimenzionalnih točk z razbitjem prostora. Najpogosteje se uporablja za območna iskanja in iskanja najbližjih sosedov. Kd drevo je v osnovi binarno drevo, kjer vsako notranje vozlišče razdeli prostor na dva polprostora (polravnini), ki ju ločuje hiperravnina (premica). Točke levo od hiperravnine so vsebovane v levem poddrevesu vozlišča, točke desno od hiperravnine pa v desnem poddrevesu. Smer hiperravnine je odvisna od tega, po kateri dimenziji vozlišče razdeli prostor. Za i-ti nivo drevesa velja, da je hiperravnina, ki razdeli prostor, pravokotna na koordinatno os, ki predstavlja $j = ((i-1) mod k + 1)$-to dimenzijo prostora. Vozlišče na drugem nivoju dvodimenzionalnega kd drevesa tako razdeli ravnino s premico, ki je vzporedna abscisni osi. Točke v spodnji polravnini se nahajajo v levem poddrevesu vozlišča, točke v zgornji polravnini pa v desnem poddrevesu. Hiperravnina, ki jo predstavlja neko vozlišče v v i-tem nivoju, gre skozi tisto točko, čigar j-ta koordinata predstavlja mediano v množici vseh točk, ki se nahajajo v drevesu s korenom. Točke so shranjene v listih



Slika 2.1: Dvodimenzionalno kd drevo.

drevesa.

Časovna zahtevnost izgradnje drevesa je $\mathcal{O}(n \log n)$ (na vsakem nivoju drevesa je potrebno izračunati mediano, kar vzame $\mathcal{O}(n)$ časa), prostorska zahtevnost drevesa pa $\mathcal{O}(n)$ [1].

2.2.1 Poizvedbe

Časovna zahtevnost poizvedbe je $\mathcal{O}(n^{(1-\frac{1}{d})} + k)$, kjer je k število poročanih točk. V splošnem za dvodimenzionalno kd drevo velja, da je območje $reg(v)$, ki ga predstavlja vozlišče v, pravokotnik, ki je lahko neomejen z več strani. Omejen je s hiperravninami (premicami), shranjenih v starših v. Točka je vsebovana v drevesu s korenom v, če je vsebovana v $reg(v)$. Drevo s korenom v med poizvedbo obiščemo le, če poizvedbeno območje (pravokotnik) seka $reg(v)$. Če je $reg(v)$ v celoti vsebovana v poizvedbenem območju, v rezultat dodamo celotno poddrevo v. Če pridemo do lista, preverimo, če je vozlišče shranjeno v njem vsebovano v območju. Primer poizvedbe je prikazan na sliki 2.1 (pri čemer je treba opozoriti, da pri drevesu na sliki za razdelitev prostora ni bila vedno izbrana mediana).

2.2.2 Kd drevesa v CGAL-u

Implementacija podatkovne strukture dosledno sledi njenemu teoretičnem opisu. Za poizvedbe je uporabljen standarden algoritem, ki rekurzivno preišče drevo. Ko pridemo do nekega vozlišča v drevesu, najprej obiščemo otroka, ki je bližje poizvedbeni točki.

2.3 Drevesa najkrajših poti z eno izvorno točko (SSSP)

Drevo najkrajših poti z eno izvorno 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 Dijkstrov in Bellman-Fordov algoritem). Izvorno 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 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,...,n, ordinatne pa 1,...,m. Dve vozlišči sta povezani, če je njuna razdalja enaka 1, zato takemu grafu pravimo tudi graf enotskih razdalj.

2.5 Delaunayeva triangulacija

Delaunayeva triangulacija nad točkami P je triangulacija $DT(P)$, ki izpolnjuje Delaunayev pogoj. Ta pravi, da se nobena točka iz P ne nahaja znotraj očrtanega kroga poljubnega trikotnika triangulacije. Taka triangulacija maksimizira minimalni kot med vsemi koti trikotnikov. Množica P s kolinearnimi točkami predstavlja izrojen primer, za katerega ne obstaja nobena Delaunayeva triangulacija. Če točke v P ležijo na istem krogu, rešitev ni enolična.

Med točko p in njenim najbližjim sosedom q, kjer $p, q \in P$, vedno obstaja

povezava $pq \in E(DT(P))$, ker je graf najbližjih sosedov nad množico P podgraf Delaunayeve triangulacije. Za dolžino najkrajše poti $dist(\pi(p,q))$ med dvema vozliščema p, q v Delaunayevem grafu velja:

$$dist(\pi(p,q)) \leq 2.418 \times dist(p,q). \tag{2.1}$$

Za izgradnjo triangulacije poznamo več algoritmov. Najpogosteje uporabljeni so:

algoritem z obračanjem Pomembna lastnost dveh trikotnikov ABD in BCD s skupno stranico BD v Delaunayevi triangulaciji je, da je vsota kotov α in γ manjša od 180°. Če ta lastnost ne velja, jo lahko dosežemo z obračanjem trikotnikov, tako da dobimo trikotnika ABC in ACD s skupno stranico AC. Algoritem z obračanjem zgradi navadno triangulacijo, nato pa izvaja operacijo obračanja, dokler vsi trikotniki ne izpolnjujejo Delaunayevega pogoja. Časovna zahtevnost v najslabšem primeru je $\mathcal{O}(n^2)$.

inkrementalni algoritem z vstavljanjem Dodaja točke zaporedoma v obstoječo triangulacijo. Ko je nova točka dodana, se tiste dele Delaunayevega grafa, na katere nova točka vpliva, popravi z obračanjem trikotnikov. V najslabšem primeru je časovna zahtevnost algoritma $\mathcal{O}(n^2)$, ker moramo za vsako dodano točko najti trikotnik, ki jo vsebuje ($\mathcal{O}(n)$), in popraviti vse trikotnike ($\mathcal{O}(n)$). Z nekoliko izboljšanim algoritmom za iskanje lokacije nove točke je pričakovana časovna zahtevnost $\mathcal{O}(n \log n)$, ker je v povprečju v vsakem koraku obrnjenih $\mathcal{O}(1)$ trikotnikov, iskanje lokacije nove točke pa vzame v povprečju $\mathcal{O}(\log n)$ časa.

deli in vladaj Točke so rekurzivno razdeljene v dve množici z določeno premico. Delaunayeva triangulacija je izračunana za vsako množico točk posebej, nato pa sta obe triangulaciji združeni vzdolž premice. Združevanje vzame $\mathcal{O}(n)$ časa, tako da je časovna zahtevnost algoritma

$\mathcal{O}(n \log n)$. V praksi je deli in vladaj najhitrejši algoritem za izgradnjo Delaunayeve triangulacije.

2.5.1 Delaunayeva triangulacija v CGAL-u

Razred Delaunay_triangulation_2 implementira Delaunayevo triangulacijo in deduje iz razreda Triangulation_2. Iz slednjega uporablja nekatere metode, kot so vstavljanje ali lociranje točke, nekatere pa tudi povozi, kot je recimo obračanje trikotnikov, ker trikotnikom v navadni triangulaciji ni potrebno izpolnjevati Delaunayevega pogoja. Poleg tega vsebuje dodatne metode za iskanje najbližjega soseda in konstrukcijo elementov Delaunayi triangulaciji dualnega Voronoijevega diagrama.

Razred je parametriziran z dvema predlogama; prvi je razred geometrijskih značilnosti, drugi pa razred podatkovne strukture za triangulacijo. Za slednjega CGAL ponuja privzeti razred Triangulation_data_structure_2, ki vsebuje zbirki vozlišč in lic. Razred geometrijskih značilnosti mora biti model koncepta DelaunayTriangulationTraits_2, ki je dodelana verzija koncepta TriangulationTraits_2. Ta vsebuje predikata za primerjavo koordinat dveh točk in test orientacije treh točk. DelaunayTriangulationTraits_2 dodatno vsebuje še predikat side_of_oriented_circle, ki za štiri dane točke p, q, r, s določi položaj točke s glede na krožnico, ki gre skozi točke p, q in r.

Izgradnja triangulacije uporablja inkrementalni algoritem z obračanjem.

2.5.1.1 Operacije nad strukturo

lociranje točke Je implementirano s sprehodom po povezavah. Sprehod se začne pri vozlišču podanega lica, ali pri naključnem vozlišču, če lice ni podano. Povprečna časovna zahtevnost je $\mathcal{O}(\sqrt{n})$.

vstavljanje nove točke Najprej je uporabljena metoda *locate* za lociranje lica, ki vsebuje novo točko, nato pa se lice razbije na tri dele. Novo triangulacijo se nato popravi z $\mathcal{O}(d)$ obračanji (v povprečju $\mathcal{O}(1)$), kjer je d stopnja novega vozlišča

iskanje najbližjega soseda Uporabljena je metoda *locate*, nato pa je potrebnega $\mathcal{O}(1)$ časa za najdbo najbližjega soseda pri enakomerno porazdeljenih vozliščih, v najslabšem primeru pa $\mathcal{O}(n)$ časa.

2.5.1.2 Hierarhija triangulacij

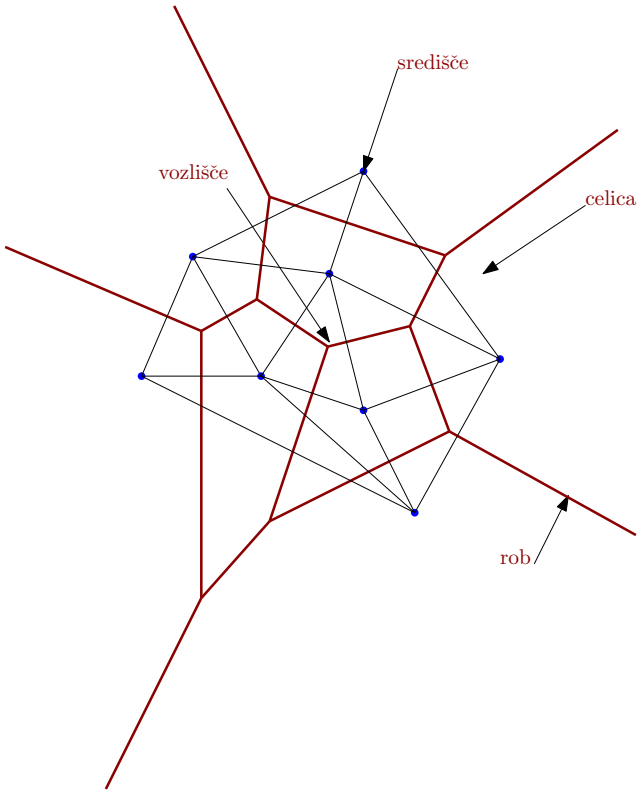
Za učinkovito lociranje točk lahko uporabimo razred Triangulation_hierarchy_2, ki hrani hierarhijo triangulacij. Na najnižjem nivoju se nahaja izvirna triangulacija, nad katero se izvajajo operacije in lociranje točk. Na vsakem višjem nivoju je nato zgrajena triangulacija nad manjšo množico naključno izbranih vozlišč iz triangulacije na predhodnem nivoju (recimo 3% vozlišč). Lociranje točke se izvede od zgoraj navzdol po strukturi: na najvišjem nivoju se naivno izvede poizvedba najbližjega soseda, nato pa se na vsakem naslednjem nivoju najde najbližjega soseda s sprehodom po povezavah z začetkom pri najbližjem sosedu, najdenem na višjem nivoju. Taka struktura je najbolj primerna ravno za Delaunayeve triangulacije. Poleg hitrega delovanja na realnih podatkih ima dobro časovno zahtevnost v najslabšem primeru, in nizko prostorsko kompleksnost [5].

2.6 Voronoijev diagram

Voronoijev diagram [7] je definiran na množici točk, imenovanih Voronoijeva središča (angleško *sites*), ki ležijo v nekem prostoru $\sum$, in z metriko oziroma funkcijo razdalje med točkami. Za ravninski Voronoijev diagram, opisan v nadaljevanju, velja $\sum = \mathbb{R}^2$.

Naj bo S $= S = \{S_1, S_2, ..., S_n\}$ množica Voroijevih 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_2 na podlagi funkcije $\delta(x, \cdot)$, je množica:

$$V_{ij} = \{x \in \mathbb{R}^2 : \delta(x, S_i) < \delta(x, S_j)\}. \tag{2.2}$$



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:

$$V_i = \bigcap_{i \neq j} V_{ij}. \tag{2.3}$$

Množico V_i imenujemo tudi Voronoijeva celica ali Voronijevo lice središča S_i. Lokus točk, ki so enako oddaljene od dveh središč, se imenuje Voronijev bisektor. Povezani podmnožici slednjega pravimo Voronoijev rob. Točki, ki je enako oddaljena od treh ali več središč, pravimo Voronoijevo vozlišče. Voronoijev diagram na množici S in z metriko $\delta(x, \cdot)$ je zbirka Voronoijevih 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 Voronoijevim diagramom, za katere zgornja omejitev drži in imajo lastnost, da so njihove celice preprosto povezano območje v ravnini, pravimo preprosti Voronoijevi diagrami. Najbolj tipičen primer slednjih je evklidski Voronoijev diagram (slika 2.2), ki ga uporabljamo v našem algoritmu.

2.6.1 Voronoijev diagram v CGAL-u

Knjižnica CGAL vsebuje razred Voronoi_diagram_2$< DG, AT, AP >$, ki deluje kot prilagoditveni paket. Ta Delaunayevo triangulacijo na podlagi podanih kriterijev prilagodi pripadajočemu Voronoijevemu 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 model koncepta razreda DelaunayGraph_2. Primeri takih struktur so Delaunayeva triangulacija, navadna triangulacija, Apollonov graf in hierarhija triangulacij. Druga predloga, AT, predstavlja lastnosti pretvorbe Delaunayeve trian-

gulacije v Voronoijev diagram 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 podmmnožice središč, ki so v izrojenem položaju, 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 režim Delaunay_triangulation_caching_degeneracy_removal_poli
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 izrojenimi 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 ($\mathcal{O}(\log^2 n + k)$), ampak slabšo prostorsko kompleksnost ($\mathcal{O}(n \log n)$). Namesto, da razdelimo prostor izmenično po abscisni in ordinatni osi, imamo

tukaj kot glavno drevo uravnoteženo dvojiško iskalno drevo $\mathcal{T}$, zgrajeno nad x koordinatami točk. Vsako vozlišče v drevesa ima potem kazalec na uravnoteženo dvojiško iskalno poddrevo $\mathcal{T}_{assoc}(v)$, zgrajeno nad y koordinatami točk, ki so shranjene v listih poddrevesa v $\mathcal{T}$ s korenom v.

Poizvedba $[x : x'] \times [y : y']$ v taki strukturi poteka na sledeč način: v glavnem drevesu iščemo z intervalom $x : x'$, dokler ne pridemo do vozlišča v_{split}, kjer se iskanje razcepi na dva dela. Iskanje nadaljujemo v smeri njegovega levega otroka in pri vsakem obiskanem vozlišču v, kjer se pot iskanja nadaljuje levo, označimo njegovega desnega otroka. Podobno nadaljujemo v smeri desnega otroka vozlišča v_{split} in označimo levega otroka obiskanega vozlišča v, pri katerem se iskanje nadaljuje v desno. V tem trenutku imamo označenih $\mathcal{O}(\log n)$ poddreves. Za vsako poddrevo s korenom v $P(v)$ naredimo poizvedbo $[y : y']$ na sekundarnem drevesu $\mathcal{T}_{assoc}(v)$, ki hrani vse točke iz $P(v)$.

Dejanske točke so shranjene v drevesih $\mathcal{T}_{assoc}$. Na danem nivoju drevesa $\mathcal{T}$ je neka točka $p \in P$ shranjena v natanko enem drevesu $\mathcal{T}_{assoc}$. Iz tega sledi, da vsa drevesa $\mathcal{T}_{assoc}$ na poljubnem nivoju drevesa T porabijo skupaj $\mathcal{O}(n)$ prostora. Ker je globina drevesa $\mathcal{T}$ $\mathcal{O}(\log n)$, je prostorska zahtevnost območnih dreves $\mathcal{O}(n \log n)$.

Časovna zahtevnost enodimenzionalne poizvedbe v poddrevesu $\mathcal{T}_{assoc}(v)$ je $\mathcal{O}(\log n + k_v)$, kjer je k_v število vrnjenih točk. Celotni čas poizvedbe je $\sum_v \mathcal{O}(\log n + k_v)$, torej vsota vseh obiskanih vozlišč v T. Ker je dolžina obeh iskalnih poti (ki se razcepita pri vozlišču v_{split}) enaka $\mathcal{O}(\log n)$, in je vsota $\sum_v k_v$ enaka številu vseh vrnjenih točk poizvedbe k, je časovna zahtevnost celotne poizvedbe $\mathcal{O}(log^2 n + k)$.

2.7.1 Območna drevesa v CGAL-u

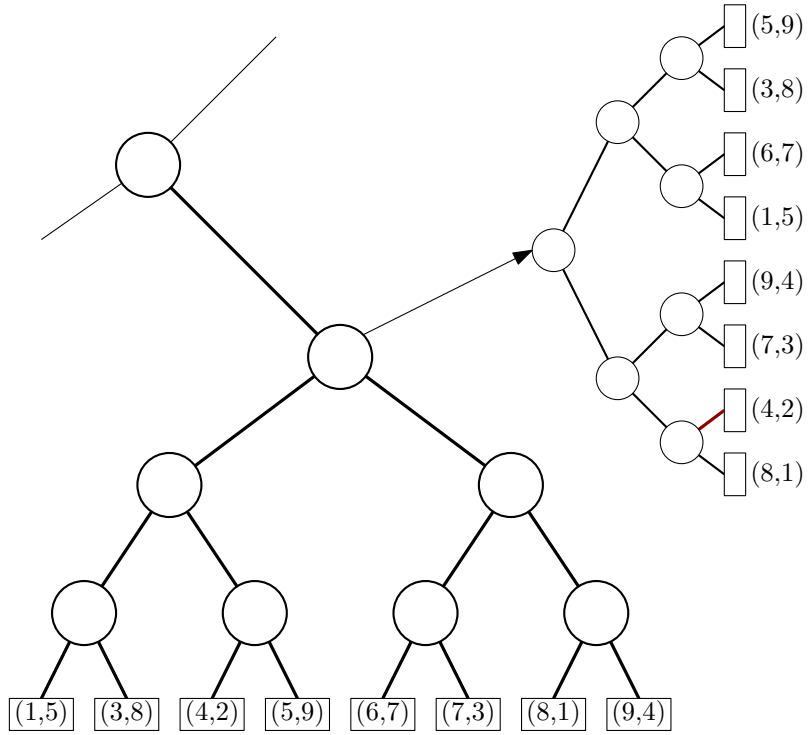
Vhodni podatkovni tip za d-dimenzionalno območno drevo je zbirka, ki vsebuje podatkovni tip za d-dimenzionalno točko in opcijsko tip vrednosti, s katerim lahko shranimo poljubni podatek. Primer uporabe slednjega bi lahko bila množica točk, ki sestavljajo več večkotnikov. Vsako vozlišče drevesa bi

potem kot ključ shranilo neko točko, kot vrednost pa poligon, kateremu ta točka pripada.

Razred za območno drevo je popolnoma generičen, kar pomeni, da z njim lahko definiramo večnivojska drevesa. d-dimenzionalno večnivojsko drevo (z d nivoji) je na d-tem nivoju preprosto drevo, k-ti nivo, $1 \leq k \leq d-1$, pa drevo, kjer vsako notranje vozlišče vsebuje večnivojsko drevo dimenzije $d-k+1$. $k-1$-dimenzionalno drevo, ki je vgnezdeno drevo v k-dimenzionalnem drevesu $\mathcal{T}$, imenujemo podnivojsko drevo drevesa $\mathcal{T}$.

Razred je neodvisen tako od vrste podatkov kot od njihove konkretne predstavitve. Za tako splošnost moramo definirati, kako je predstavljeno drevo na vsakem nivoju in kako so vhodni podatki organizirani. Drevesa z dimenzijo k, $1 < k < 4$, so zato v CGAL-u definirana v lastnih razredih, pripravljenih za takojšnjo uporabo. Ko je drevo enkrat zgrajeno, dodajanje ali brisanje podatkov iz njega ni več mogoče.

Drevo je parametrizirano s tremi predlogami: podatki, oknom in geometrijskimi značilnostmi. Prvi definira vrsto podatkov, drugi poizvedbeno območje, tretji pa podajajo implementacijo metod, ki jih drevo uporablja za dostop do podatkov. Da bi se izognili uporabi vgnezdenih predlog za argumente (do česar pride, če bi drevo na vsakem nivoju imelo predlogo za tip poddrevesa), so drevesa definirana s pomočjo objektnega programiranja. CGAL je definiral virtualni bazni razred *Tree_base*, iz katerega je izpeljan razred *Range_tree_d*. Konstrukturju slednjega moramo kot argument podati prototip poddrevesa tipa *Tree_base*, ki je lahko zopet objekt razreda *Range_tree_d*. Za zaustavitev rekurzije je iz *Tree_base* izpeljan še razred *Tree_anchor*, čigar konstruktor ne pričakuje nobenega argumenta. Tako opisana konstrukcija sledi modelu prototipa in v njej je poddrevo definirano s pomočjo objektnega programiranja v času izvajanja programa, kar pa predstavlja zanemarljiv časovni strošek.



Slika 2.3: Dvodimenzionalno območno drevo.

Poglavje 3

Teoretični opis algoritmov

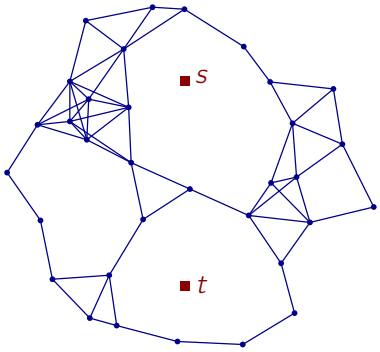
Naj bo P množica točk, ki predstavljajo središča krogov $\mathcal{D}$. P predstavlja vhod našega algoritma in vse operacije ter uporabljene podatkovne strukture se vrtijo okoli te množice. Kardinalnost množice je enaka n.

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

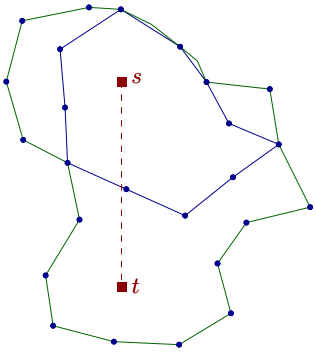
V nadaljevanju bomo namesto $G(P)$ pisali preprosto G, brez izgube splošnosti pa predpostavili, da je $z = (0, 0)$ in $z' = (0, s)$. zz' je torej daljica, v nadaljevanju imenovana σ, ki je vsebovana v koordinatni osi y. Če dana vhodna daljica ne leži na osi y, lahko naredimo translacijo nad σ in P. Ce daljica ni niti navpična, lahko naredimo tudi rotacijo daljice in točk, kjer pa je treba upoštevati, da pri tem pride do numeričnih napak in se prave razdalje med točkami izgubijo.

3.1 Drevo SSSP

3.1.1 Eksplicitna uporaba preiskovanja v širino (BFS)



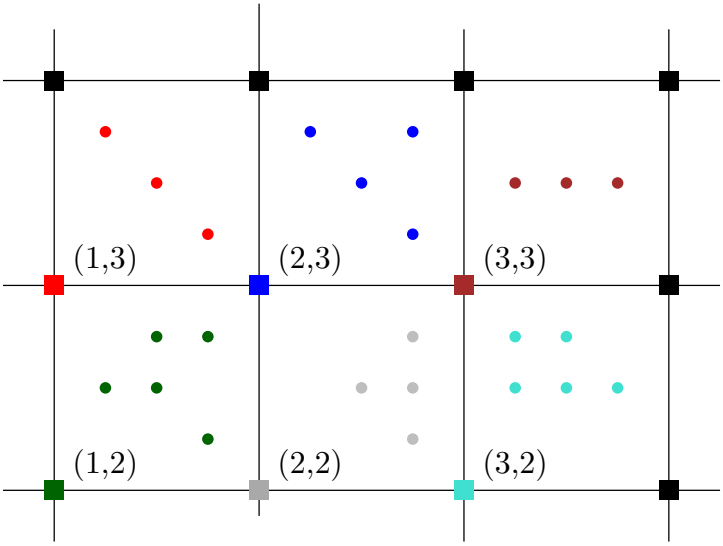
Slika 3.1: Graf $G(P)$, zgrajen nad množico P v sliki 1.1.



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.

```
BFS(G, s)
 1  // za vsako povezavo e ∈ G.E velja dist(e) ≤ 1
 2  for u ∈ G.V − {s}
 3        u.visited = false
 4        u.dist = ∞
 5        u.π = NIL
 6  s.visited = true
 7  u.dist = 0
 8  u.π = NIL
 9  Q = ∅
10  enqueue(Q, s)
11  while Q ≠ ∅
12        u = dequeue(Q)
13        for (u, v) ∈ G.E
14              if v.visited == false
15                    v.visited = true
16                    v.dist = u.dist + 1
17                    v.π = u
18                    enqueue(Q, v)
```

Slika 3.3: Psevdokoda eksplicitnega (splošnega) algoritma za preiskovanje v širino in posledično izgradnjo drevesa najkrajših poti.



Slika 3.4: .

3.1.2 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 razdalj. 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 3.4). 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.

3.1.3 Prilagojen algoritem za enotske diske

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).

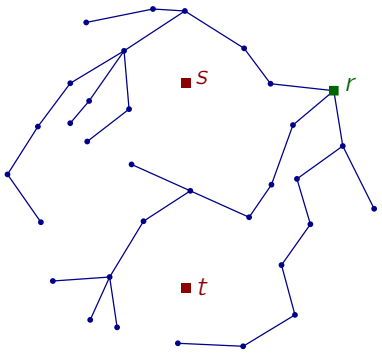
Algoritem dobi kot vhod poleg množice P izvorno točko $s \in P$ in nato inkrementalno v vsaki iteraciji dodaja točke h grafu. Množico točk, dodanih k drevesu v i-ti iteraciji, označimo kot

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

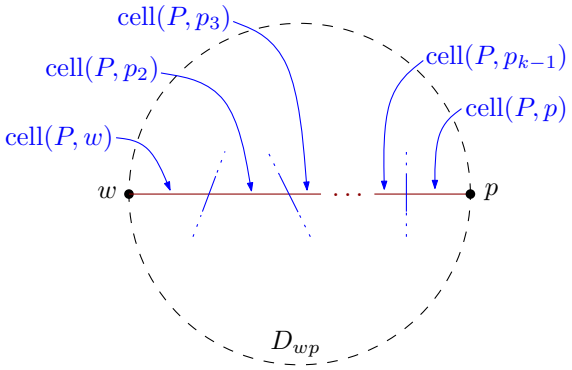
Velja torej $W_0 = \{s\}$. Za izgradnjo W_i ne potrebujemo celotnega grafa, zgrajenega do $i-1$-te iteracije, temveč samo množico W_{i-1}. Za vsak $q \in W_i$ velja, da je povezan s točko $p = NN(q, W_{i-1})$. p je torej izmed točk v W_{i-1} najbližje q.

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 $DT(P)$



Slika 3.5: Drevo najbližjega soseda, ki hrani 6 točk, predstavljeno v obliki seznama. Za vsako celico so prikazani indeksi točk, ki jih hrani objekt v njej.



Slika 3.6: aa

- točke sosednje (do tega trenutka zgrajeni) W_i v $DT(P)$.

Lema 3.1. *Naj bo p točka v $P\backslash\{s\}$, 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. Pobliže si poglejmo zaporedje Voronoijevih celic $cell(p_1, P), ..., cell(p_k, P)$, 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 $\pi = p_1 p_2 ... p_k$ 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 $wp \cap cell(p_j, P)$. 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 $d(s, p_j) = i$. □

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

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

Poleg tega za vsako točko $p \in P \backslash \{s\}$ velja $dist[p] = d(s, p)$ 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 $W_0, W_1, ...$ parno disjunktne. To je razvidno tudi iz psevdokode. Točka p je v vrstici 21 dodana v nek W_i, po tem, ko nastavimo $dist[p] = i$ 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

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

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 $W_0, W_1, ...$ sledi $d(s,p) = i$. Sklenemo, da

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

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 $\pi = p_1...p_k$, 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_1p_2 obravnavana v vrstici 15 in točka $p2$ je dodana v W_i in Q. Po $indukcijskitezi$(!!!) so potem vse točke $p_3, ..., p_k$ dodane v W_i in Q (po možnosti v različnem vrstnem redu). Sledi, da je $p_k = p$ dodan v W_i in zato

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

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 $\pi[p]p$. □

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.

Celotni algoritem je podan na sliki 3.7.

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.

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 $dist[p] \leq i < \infty$, 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

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

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

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

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

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

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

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 korenom 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 korenom s in $dist[\cdot]$ pravilno opisuje razdalje najkrajših poti v $G(P)$. □

3.2 Minimalna ločitev enotskih diskov

Cabello in Giannopoulos [2] sta predstavila splošen algoritem za problem minimalne ločitve, ki se v najslabšem scenariju izvede v kubičnem času. Algoritem deluje za poljubne smiselne oblike, kot so recimo daljice ali elipse, in ne samo za enotske diske. Njegova splošnost je sicer prednost, a po drugi strani ne izkorišča nobenih lastnosti enotskih diskov.

V tem poglavju je opisan algoritem, ki problem minimalne ločitve z enotskimi diski reši v skoraj kvadratičnem času. Izboljšava temelji na treh sestavinah:

- reinterpretacija generičnega algoritma za diske. V izvirnem algoritmu moramo označiti točko znotraj vsake oblike. Pri diskih je izbira očitna - njihovo središče. S tem se opis in interpretacija algoritma poenostavita.
- učinkovit algoritem za drevesa najkrajših poti nad grafom G. Primeren kandidat je algoritem, opisan v poglavju 3.1.3, ker izkorišča lastnosti grafa G.
- kompaktna obravnava povezav v G z uporabo sledečih orodij iz računske geometrije: območnih dreves, iskanja najbližjega soseda in dualnosti med točko in premico.

3.2.1 Splošni algoritem za ločevanje z diski

```
UnweightedShortestPath(P, r)
 1  zgradi Delaunayevo triangulacijo DT(P)
 2  for p ∈ P
 3       dist[p] = ∞
 4       π[p] = NIL
 5  dist[r] = 0
 6  W_0 = {r}
 7  i = 1
 8  while W_{i-1} ≠ ∅
 9       zgradi pod. strukturo za poizvedbe
          najbližjega soseda v W_{i-1}
10       Q = W_{i-1} // generator točk kandidatk
11       W_i = ∅
12       while Q ≠ ∅
13            q poljubna točka v Q
14            odstrani q iz Q
15            for qp povezava v DT(P)
16                 if dist[p] = ∞
17                      w = najbližji sosed p v W_{i-1}
18                      if |pw| ≤ 1
19                           dist[p] = i
20                           π[p] = w
21                           dodaj p v Q
22                           dodaj p v W_i
23       i = i + 1
24  return dist[·] in π[·]
```

Slika 3.7: Algoritem iz [3] za izračun neuteženega drevesa najkrajših poti.

```
GenericMinimumSeparation(P, s, t)
 1  best = ∞ // dolžina trenutno najboljše ločitve
 2  for r ∈ P
 3       (dist[ ], π[ ]) = drevo najkrajših poti iz r v G(P)
          // izračunaj N[ ]
 4       N[r] = 0
 5       for p ∈ P \ {r} v nepadajočih vrednostih od dist[p]
 6            N[p] = N[π[p]] + cr_2(st, pπ[p]) (mod 2)
 7       for pq ∈ E(G(P))
 8            if N[p] + N[q] + cr_2(pq, st) (mod 2) = 1
 9                 best = min{best, dist[p] + dist[q] + 1}
10  return best
```

Slika 3.8: Prilagoditev splošnega algoritma za izračun minimalne ločitve za enotske diske.

Naj sprehod W v grafu $G = G_{\leq 1}(P)$ definira ravninsko poligonsko krivuljo na očiten način: točke v P povežemo med seboj z daljicami v danem zaporedju iz W. Notacijo bomo naredili ohlapnejšo in z W označili kar samo krivuljo. Za poljubno vpeto drevo T iz G in poljubno povezavo $e \in E(G) \setminus E(T)$ naj bo $cycle(T, e)$ edinstven cikel v $T + e$. Za poljubni sprehod v G naj bo $\mathtt{cr}_2(st, W)$ število presečišč med daljico st in krivuljo W po modulu 2. Naslednja lastnost je implicitna v [2]:

> Naj bo T vpeto drevo iz G. Množica enotskih diskov s središči v P ločuje s in t natanko takrat, ko obstaja taka povezava $e \in E(G) \setminus E(T)$, da velja $\mathtt{cr}_2(st, cycle(T, e)) = 1$.

Posledica tega je, da najti minimalno ločitev pomeni isto kot najti najkrajši cikel v G, ki ima liho število presečišč z daljico st. Še več, iskanje lahko omejimo na konkretno družino ciklov. Naj bo W^* poljuben optimalen cikel in naj bo r^* poljubno vozlišče iz W^*. Fiksirajmo drevo najkrajših poti T_{r^*} v G s korenom r^*. Potem množica ciklov $\{cycle(T_{r^*}, e) \mid e \in E(G) \setminus E(T_{r^*})\}$ vsebuje optimalno rešitev. To sledi iz tako imenovanega tri-potnega pogoja (ideja v [2], naj dodamo njen opis in/ali dokaz???). Ker je r^* neznan, preverimo kar vse korene (v primerih, kjer je optimalna rešitev velika, to pripelje do možnosti uporabe randomiziranega algoritma, kjer nekatere korene izberemo naključno). Velikost optimalne rešitve je dana z

$$\min\{1 + d_G(r, p) + d_G(r, q) \mid \\ r \in P,\ pq \in E(G) \setminus E(T_r),\ \mathtt{cr}_2(st, cycle(T_r, pq)) = 1\}.$$

Vrednosti $\mathtt{cr}_2(st, cycle(T_r, e))$ lahko za posamezno povezavo e izračunamo v konstantnem amortiziranem času s preprostim knjigovodstvom. Poglejmo fiksno drevo T_r. Za vsako točko $p \in P$ shranimo $N[p]$ kot pariteto števila presečišč poti v T_r iz r do p. Ko p ni koren drevesa, lahko vrednost $N[p]$ izračunamo s pomočjo njegovega starša kot $N[p] = N[\pi[p]] + \mathtt{cr}_2(st, p\pi[p])$. V psevdokodi algoritma (vrstice 4-6) je postopek enako opisan, čeprav lahko vrednosti $N[p]$ računamo tudi hkrati z računanjem drevesa najkrajših poti T_r.

Za vsako drevo najkrajših poti T_r imamo potem

$$\begin{aligned} &\forall pq \in E(G) \setminus E(T_r) : \\ &\mathtt{cr}_2(st, cycle(T_r, pq)) = N[p] + N[q] + \mathtt{cr}_2(st, pq) \pmod 2 \\ &\forall pq \in E(T_r) : \\ &0 = N[p] + N[q] + \mathtt{cr}_2(st, pq) \pmod 2 \end{aligned}$$

ker se presečišča, ki smo jih šteli dvakrat, izničijo po modulu 2. Podrobneje, pot v T_r iz r do najmanjšega skupnega prednika p in q je upoštevana dvakrat. Iz tega sledi, da je dovolj za vse povezave pq v G preveriti, če je vsota $N[p] + N[q] + \mathtt{cr}_2(st, pq)$ enaka 0 po modulu 2. Končni algoritem je podan na sliki 3.8.

Poglejmo si časovno kompleksnost algoritma. Za vsako točko $r \in P$ moramo izračunati drevo najkrajših poti v G. Kot smo omenili v poglavju 3.1.3, je to izvedljivo v času $\mathcal{O}(n \log n)$. Za vsako povezavo pq potem porabimo $\mathcal{O}(1)$ časa. Za vsako točko r to skupaj znese $O(n \log n + |E(G)|)$ časa, kar v najslabšem primeru pomeni kubično časovno zahtevnost. Boljši čas lahko dobimo s kompaktno obravnavo vseh povezav v G, kar je opisano v poglavju 3.2.2.

3.2.1.1 Dualnost in dualni prostor

Točka v ravnini ima dva parametra: koordinati x in y. Nenavpična premica v ravnini ima prav tako dva parametra: smerni koeficient in odsek na ordinatni osi. To pomeni, da lahko naredimo bijektivno preslikavo množice točk v množico premic. Pri nekaterih se lahko nekatere lastnosti množice točk prenesejo v določene lastnosti množice premic. Lep primer so tri točke na premici, ki se preslikajo v tri premice, ki grejo skozi isto točko. Preslikavam, ki to omogočajo, pravimo dualne transformacije. Preprost primer dualne transformacije je sledeč. Naj bo $p = (p_x, p_y)$ točka v ravnini. Njen dvojnik p^* je premica, definirana kot $p^* = (y = p_x x - p_y)$. Dvojnik premice $l : y = mx + b$ je taka točka p, da za njo velja $p^* = l$. Ali drugače rečeno, $l^* = (m, -b)$.

Za dualno transformacijo pravimo, da objekte preslika iz primarnega v dualni prostor. Sliki objekta v dualnem prostoru pravimo *dvojnik*. Določene lastnosti, ki držijo v primarnem, držijo tudi v dualnem prostoru. Naj bo p točka, l pa premica v ravnini. Dualna transformacija $o \mapsto o^*$ ima naslednje lastnosti:

- ohranja vsebovanost: $p \in l$ natanko takrat, ko velja $l^* \in p^*$
- ohranja vrstni red: p leži nad l natanko takrat,ko l^* leži nad p^*.

Dualno transformacijo se lahko uporabi tudi na drugih objektih, kot so recimo daljice. Smiselna izbira za s^*, kjer je s daljica $\overline{pq}$, bi bila unija vseh dvojnikov točk na s. Rezultat je neskončna množica premic, ki se sekajo v eni točki. Grafično si to lahko predstavljamo kot unijo dveh območij s^*, ki sta omejeni s premicama p^* in q^* in se stikata v točki njunega presečišča (glej tudi sliko) [1].

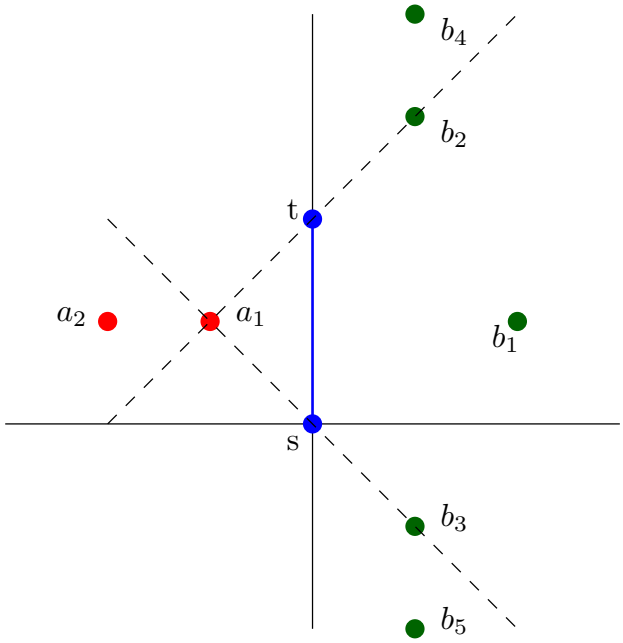
3.2.2 Hitrejši algoritem

Uporabili bomo podatkovno strukturo v naslednji lemi. V osnovi gre za večnivojsko podatkovno strukturo, ki vsebuje dvodimenzionalno območno drevo T, slednje pa na vsakem vozlišču sekundarnega drevesa hrani podatkovno strukturo za iskanje najbližjega soseda.

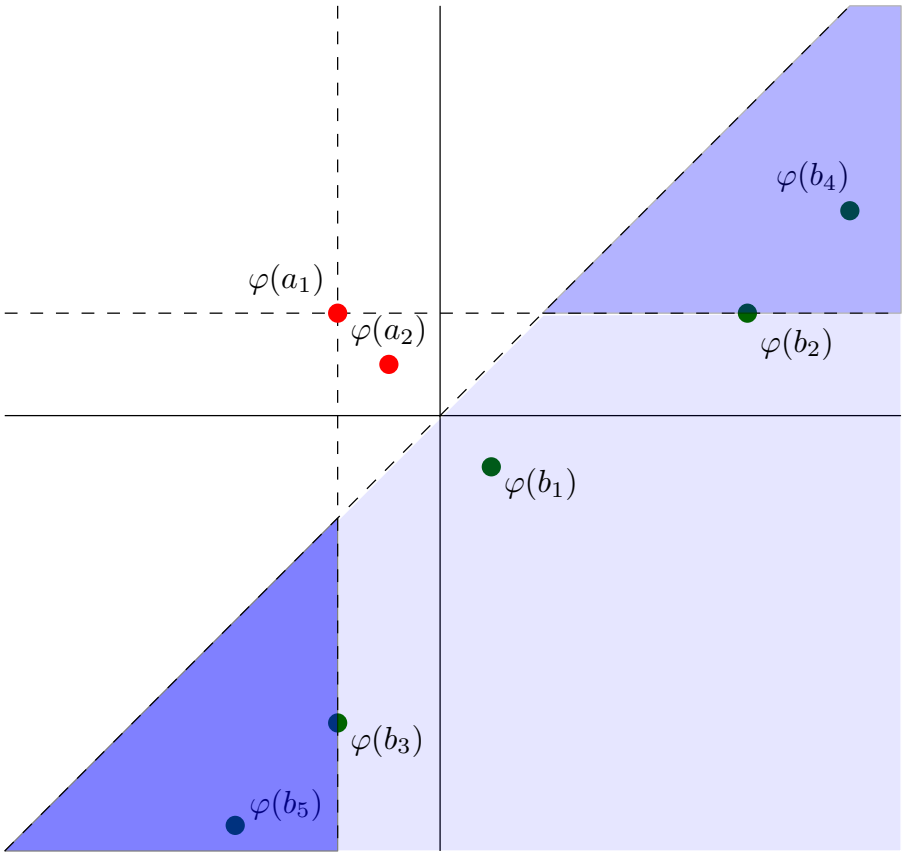
Lema 3.5. *Naj bo B množica točk s pozitivnimi koordinatami x. B lahko predprocesiramo v času $\mathcal{O}(n \log^3 n)$, tako da za vsako točko $a \in A$ z negativno koordinato x lahko v času $\mathcal{O}(\log^3 n)$ določimo, če je množica $\{b \in B \mid ab \text{ seka } \sigma \text{ and } |ab| \leq 1\}$ prazna. Z isto podatkovno strukturo lahko obravnavamo poizvedbe za ugotavljanje, ali je množica $\{b \in B \mid ab \text{ ne seka } \sigma \text{ and } |ab| \leq 1\}$ prazna.*

Dokaz. Za potrebe dokaza uporabimo dualnost, opisano v prejšnjem poglavju, in območna drevesa.

Naj bo $\mathbb{L}$ množica nenavpičnih premic, σ^* pa množica točk dualnih nevertikalnim daljicam, ki sekajo daljico $\sigma = st$:



Slika 3.9: 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.



Slika 3.10: Spodaj so prikazane točke iz slike 3.9 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.

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

V dualnem prostoru je množica σ^* *horizontalslab*

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

ker velja predpostavka $s = (0, 0)$ and $t = (0, \tau)$, kjer $\tau \geq 0$.

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 σ^* in ima krajišči $(\varphi_1(b), 0)$ in $(\varphi_2(b), -\tau)$ na obeh njegovih mejah. $\varphi_1(b)$ predstavlja smerni koeficient premice, ki seka točki $(0, 0)$ in b, $\varphi_2(b)$ pa smerni koeficient premice, ki seka točki $(0, \tau)$ in b.

Definirajmo točko preslikave $\varphi(b) = (\varphi_1(b), \varphi_2(b))$. Funkcija preslikave φ torej preslika točke z nenegativnimi koordinatami x v točke v ravnini.

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:

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

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:

$$ab \cap \sigma \neq \emptyset \iff \varphi_1(a) \leq \varphi_1(b)) \text{ in } \varphi_2(a) \geq \varphi_2(b)).$$

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 3.10.)

Za shranjevanje množice točk $\varphi(B)$, kjer je vsaka točka $b \in B$ asociirana s točko preslikave $\varphi(b)$, lahko uporabimo dvodimenzionalno območno drevo. Na vsakem vozlišču v sekundarnega drevesa shranimo podatkovno strukturo za poizvedbe najbližjega soseda nad kanonično množico $P(v)$, ki vsebuje točke shranjene pod v v sekundarnem drevesu.

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

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

To pomeni, da množico $\{b \in B \mid ab \text{ seka } \sigma\}$ dobimo kot unijo kanoničnih podmnožic $P(v_1), ..., P(v_k)$ za $k = \mathcal{O}(\log^2 n)$ vozlišč v sekundarnem drevesu. Za vsako tako podmnožico $P(v_i)$ naredimo poizvedbo najbližjega soseda za točko a. Če za nek v_i najdemo najbližjega soseda, ki je za razdaljo največ 1 oddaljen od a, potem vemo, da je množica $\{b \in B \mid ab \text{ seka } \sigma \text{ in } |ab| \leq 1\}$ neprazna. V nasprotnem primeru je množica prazna.

Čas zgraditve dvodimenzionalnega območnega drevesa je $\mathcal{O}(n \log n)$. Vsaka točka se pojavi v $\mathcal{O}(\log^2 n)$ kanoničnih podmnožicah $P(v)$. To pomeni, da $\sum_v |P(v)| = O(n \log^2 n)$, kjer vsota iterira skozi vsa vozlišča v v sekundarnem drevesu. Ker za vsak v zgradimo podatkovno strukturo za iskanje

najbližjega soseda, kar vzame $O(|P(v)| \log |P(v)|)$ časa, je skupna časovna zahtevnost zgraditve podatkovne strukture $\mathcal{O}(n \log^3 n)$.

Za poizvedbe dvodimenzionalno območno drevo vzame $\mathcal{O}(\log^2)$ časa, da najde $\mathcal{O}(\log^2)$ vozlišč $v_1, ..., v_k$, tako da velja

$$\bigcup_{i=1}^{k} P(v_i) \;=\; \{b \in B \mid ab \text{ seka } \sigma\},$$

potem pa dodatno potrebujemo $\mathcal{O}(\log n)$ časa za vsako vozlišče, da naredimo poizvedbo najbližjega soseda. Poizvedbe za $\{b \in B \mid ab$ ne seka σ and $|ab| \leq 1\}$ naredimo z isto podatkovno strukturo nad točkami v drugih dveh kvadrantih (glej sliko 3.10). □

Podatkovna struktura za poizvedbe najbližjega soseda v lemi 3.5 ima časovno zahtevnost zgraditve $\mathcal{O}(n \log n)$ in časovno zahtevnost poizvedbe $(\log n)$. Če bi uporabili neko drugo podatkovno strukturo za poizvedbe najbližjega soseda s časom zgraditve $T_c(n)$ in časom poizvedbe $T_q(n)$, bi čas zgraditve v lemi 3.5 postal $O(T_c(n \log^2 n))$, čas poizvedbe pa $O(T_q(n) \cdot \log^2 n)$.

Iz teoretičnega vidika bi bilo bolj učinkovito izračunati unijo

$$\bigcup_{b \in B} \{(x, y) \in \mathbb{R}^2 \mid x < 0,\ |(x, y)b| \leq 1,\ (x, y) \text{ seka } \sigma\}$$

in tam narediti point location(???). Ker območja ne morejo imeti veliko presečišč z daljico σ, lahko dobimo dobre asimptotične meje. V praksi pa se izkaže, da je izboljšava rezultata zanemarljiva.

Obravnavajmo zdaj fiksni koren r. Predpostavimo, da so drevo najkrajših poti T_r ter tabele $\pi[\,]$, $dist[\,]$ and $N[\,]$ že izračunane. Točke združimo glede na njihovo razdaljo od r:

$$W_i \;=\; \{p \in P \mid dist[p] = i\}, \quad i = 0, 1, \ldots$$

Standardna lastnost dreves BFS, ki drži tudi tukaj,je da se razdalji od r dveh sosednih vozlišč razlikujeta za največ 1. To pomeni, da so sosedi točke

$p \in P$ v G vsebovani v $W_{dist[p]-1} \cup W_{dist[p]} \cup W_{dist[p]+1}$. To lastnost bomo uporabili pri našem algoritmu.

Naredimo skupine L_i^j in R_i^j (kjer L pomeni "levo"in R pomeni "desno"), definirane kot

$$L_i^j = \{p \in P \mid dist[p] = i,\ p.x < 0,\ N[p] = j\},\ \text{kjer } j = 0,1 \text{ in } i = 0,1,\ldots$$
$$R_i^j = \{p \in P \mid dist[p] = i,\ p.x > 0,\ N[p] = j\},\ \text{kjer } j = 0,1 \text{ in } i = 0,1,\ldots$$

Zanimajo nas povezave pq v G, za katere velja $N[p] + N[q] + \mathtt{cr}_2(st, pq) = 1 \pmod 2$. Z upoštevanjem simetrije je to ekvivalentno parom točk (p, q) v enem od naslednjih dveh primerov:

- za nek $i \in \mathbb{N}$ in nek $j \in \{0,1\}$ imamo $p \in L_i^j \cup R_i^j$, $q \in L_i^{1-j} \cup R_i^{1-j} \cup L_{i-1}^{1-j} \cup R_{i-1}^{1-j}$, $|pq| \leq 1$, in pq ne seka st;
- za nek $i \in \mathbb{N}$ in nek $j \in \{0,1\}$ imamo $p \in L_i^j \cup R_i^j$, $q \in L_i^j \cup R_i^j \cup L_{i-1}^j \cup R_{i-1}^j$, $|pq| \leq 1$, in pq seka st.

Oba primera se da učinkovito rešiti z enim od naslednjih primerov:

- Če želimo iskati kandidate $(p, q) \in L_i^j \times L_{i'}^{1-j}$ (ki ne morejo sekati st, ker ležijo na isti strani osi y), najprej preprocesiramo $L_{i'}^{1-j}$ za poizvedbe najbližjega soseda. Potem za za vsako točko p v L_i^j naredimo poizvedbo nad podatkovno strukturo, da najdemo njenega najbližjega soseda q_p v L_i^j. Če za nek p velja $|pq_p| \leq 1$, smo našli povezavo pq_p v G, za katero velja $\mathtt{cr}_2(cycle(T_r, pq_p)) = 1$ in $dist[p] + dist[q_p] + 1 = i + i' + 1$. Če za vsak p velja $|pq_p| > 1$, potem $L_i^j \times L_{i'}^{1-j}$ ne vsebuje nobene povezave iz G. Če $m = |L_i^j| + |L_{i'}^{1-j}|$, je skupni čas izvajanja $\mathcal{O}(m \log m)$.
- Če želimo iskati kandidate $(p, q) \in L_i^j \times R_{i'}^j$, ki sekajo st, najprej preprocesiramo $R_{i'}^j$ v podatkovno strukturo, kot smo opisali v lemi 3.5. Nato za vsako točko $p \in L_i^j$ naredimo poizvedbo na podatkovni strukturi za take točke q, da pq seka st. Če kot rezultat dobimo neprazno množico, potem obstaja pq v G, za katero velja $p \in L_i^j$, $q \in R_{i'}^j$, $\mathtt{cr}_2(cycle(T_r, pq)) = 1$ in $dist[p] + dist[q] + 1 = i + i' + 1$. V nasprotnem

primeru ne obstaja nobena povezava $pq \in L_i^j \times R_{i'}^j$, ki bi sekala st. Za $m = |L_i^j| + |R_{i'}^j|$ je skupni čas izvajanja $O(m \log^3 m)$.

- Če želimo iskati kandidate $(p, q) \in L_i^j \times R_{i'}^{1-j}$, ki ne sekajo st, najprej preprocesiramo $R_{i'}^{1-j}$ v podatkovno strukturo, kot smo opisali v lemi 3.5. Nato za vsako točko $p \in L_i^j$ naredimo poizvedbo na podatkovni strukturi za take točke q, da pq ne seka st. Preostali potek je podoben kot v prejšnji točki.

Sklenemo lahko, da vsakega od primerov lahko rešimo v času $\mathcal{O}(m \log^3 m)$ v najslabšem primeru, kjer je m število točk udeleženih v primeru. Z iteracijo čez vse možne vrednosti i je zdaj to preprosto pretvoriti v algoritem, ki porabi $\mathcal{O}(n \log^3 n)$ časa za vsak koren r. Za primer z enotskimi diski ta algoritem izboljša čas splošnega algoritma.

Izrek 3.6. *Problem minimalne ločitve n enotskih krogov lahko rešimo v času $\mathcal{O}(n^2 \log^3 n)$.*

Dokaz. Naj bodo P središča krogov in - tako kot prej - obravnavajmo graf $G = G_{\leq 1}(P)$. Za vsak $r \in P$ zgradimo drevo najkrajših poti in množice $W_i, L_i^0, L_i^1, R_i^0, L_i^1$ za vsak i v času $O(n \log n)$. Nato imamo največ n iteracij, kjer za vsako iteracijo i porabimo $O(|W_i \cup W_{i-1}| \log^3 |W_i \cup W_{i-1}|)$ časa. Ker so množice w_i disjunktne in če seštejemo čase vseh iteracij, je čas izvajanja za posamezen koren $r \in P$ enak $\mathcal{O}(n \log^3 n)$.

Pravilnost algoritma sledi iz dejstva, da računa isto kot splošni algoritem. □

Opis postopka za posamezen koren je shematično prikazan na sliki 3.11. (Appendix: celotni algoritem???) Kot pri splošnem algoritmu spremenljivka *best* hrani dolžino trenutno najkrajšega cikla. Na začetku lahko njeno vrednost nastavimo na $n+1$. Če algoritem konča z isto vrednostjo spremenljivke, potem za dane točke ne obstaja rešitev za problem minimalne ločitve. Ko obravnavamo določen koren r, nas zanimajo samo cikli s korenom r in dolžino največ *best*. Ker ima poljuben cikel, ki gre skozi vozlišče v W_i, dolžino najmanj $2i$, lahko obravnavamo samo indekse i, za katere velja $2i < best$. Poleg

tega lahko dodatno omejimo iskanje (kar sicer ni opisano v algoritmu, je pa uporabljeno pri implementaciji) tako, da najprej obravnavamo pare, ki tvorijo cikel z dolžino $2i$, na primer $L_i^0 \times L_{i-1}^1$, in šele potem tiste z dolžino $2i+1$, na primer $L_i^0 \times L_i^1$. Z uporabo takega zaporedja lahko končamo z iskanjem za koren r takoj, ko najdemo povezavo v *while* zanki, in preidemo na naslednji koren.

// Delo za koren $r \in P$

1 $(dist[\,], \pi[\,])$ = drevo najkrajših poti iz r v $G_{\leq 1}(P)$
2 Izračunaj nivoje $W_0, W_1, \ldots$
3 **for** $i = 0 \ldots n$
4 Izračunaj $N[p]$ za vsak p v W_i
5 Izračunaj L_i^0, L_i^1, R_i^0, R_i^1
6 $i = 1$
7 **while** $2i <$ *best* in $W_i \neq \emptyset$
// znotraj obeh strani osi y
8 išči kandidate v
$L_i^0 \times L_{i-1}^1$, $L_i^1 \times L_{i-1}^0$, $R_i^0 \times R_{i-1}^1$,
$R_i^1 \times R_{i-1}^0$, $L_i^0 \times L_i^1$ in $R_i^0 \times R_i^1$
// preko osi y skozi σ
9 išči kandidate, ki sekajo σ v
$L_i^0 \times R_{i-1}^0$, $L_i^1 \times R_{i-1}^1$, $L_{i-1}^0 \times R_i^0$,
$L_{i-1}^1 \times R_i^1$, $L_i^0 \times R_i^0$ in $L_i^1 \times R_i^1$
// preko osi y mimo σ
10 išči kandidate, ki ne sekajo σ v
$L_i^0 \times R_{i-1}^1$, $L_i^1 \times R_{i-1}^0$, $L_{i-1}^1 \times R_i^0$,
$L_{i-1}^0 \times R_i^1$, $L_i^0 \times R_i^1$ in $L_i^1 \times R_i^0$
11 $i = i + 1$

Slika 3.11: Work for each vertex in the new algorithm for minimum separation with unit disks.

```
SeparationUnitDisksCompact(P, s, t)
 1  best = n + 1 // length of the best separation so far
 2  for r ∈ P
 3      (dist[ ], π[ ]) = shortest path tree from r in G_≤1(P)
        // Compute the levels W_i
 4      for i = 0 ... n
 5          W_i = new empty list
 6      for p ∈ P
 7          add p to W_dist[p]
        // Compute N[ ] for the elements of W_i and
        // and construct L_i^0, L_i^1, R_i^0, R_i^1
 8      N[r] = 0
 9      for i = 1 ... n
10          for p ∈ W_i
11              N[p] = N[π[p]] + cr_2(st, pπ[p]) (mod 2)
12              if p to the left of the y-axis
13                  add p to L_i^N[p]
14              if p to the right of the y-axis
15                  add p to R_i^N[p]
```

```
1       i = 1
2       while 2i < best and W_i ≠ ∅
            // length 2i; within each side of the y-axis
3           search candidates in L_i^0 × L_{i-1}^1, L_i^1 × L_{i-1}^0, R_i^0 × R_{i-1}^1, R_i^1 × R_{i-1}^0
            // length 2i; across y-axis crosing σ
4           search candidates crossing σ in
5           L_i^0 × R_{i-1}^0, L_i^1 × R_{i-1}^1, L_{i-1}^0 × R_i^0, L_{i-1}^1 × R_i^1
            // length 2i; across y-axis not crosing σ
6           search candidates not crossing σ in
7           L_i^0 × R_{i-1}^1 , L_i^1 × R_{i-1}^0, L_{i-1}^0 × R_i^1,L_{i-1}^1 × R_i^0
            // length 2i + 1; within each side of the y-axis
8           search candidates in L_i^0 × L_i^1, R_i^0 × R_i^1
            // length 2i + 1; across y-axis crosing σ
9           search candidates crossing σ in L_i^0 × R_i^0, L_i^1 × R_i^1
            // length 2i + 1; across y-axis not crosing σ
10          search candidates not crossing σ in L_i^0 × R_i^1, L_i^1 × R_i^0
11          i = i + 1
12  return best
```

Slika 3.12: New algorithm for minimum separation with unit disks.

Poglavje 4

Implementacija algoritmov

4.1 Drevo SSSP

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 seznami 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 atribute.

4.1.1 Podatkovna struktura za točko

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. Koren drevesa ima za razdaljo vrednost 0, točke, ki jih ni moč doseči iz korena, pa največjo možno vrednost števila tipa *int*, tj. $numeric\_limits < int >:: max()$

parent hrani deljen kazalec ($shared\_ptr < Point\_2 >$) na svojega starša v drevesu

visited hrani logično vrednost (*true* ali $false$), ki pove, če je vozlišče bilo že dodano v drevo. To bi sicer lahko izvedeli tudi preko atributa *dist*

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 Podatkovna struktura za iskanje najbližjega soseda

Kot smo omenili pri teoretičnem 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.

V praksi se pri uporabi knjižnice CGAL izkaže, da je za iskanje najbližjega soseda bolje uporabiti Delaunayevo triangulacijo. Do tega sklepa smo prišli z analizo implementacije obeh struktur:

> Za iskanje najbližjega soseda z Voronoijevim diagramom uporabimo funkcijo $locate(Point\ q)$, ki za rezultat vrne Voronoijevo središče, vozlišče, ali rob. V našem primeru vedno iščemo najbližje Voronoijevo središče, tako da v primeru, da je rezultat vozlišče ali rob, vrnemo enega od enako oddaljenih središč. Ker v CGAL-u objekt Voronoijevega diagrama interno hrani strukturo Delaunayeve triangulacije, v praksi potemtakem pravzaprav

iščemo vozlišče v Delaunayevi triangulaciji. Funkcija *locate* za izračun uporablja funktor za iskanje najbližjega Delaunayevega vozlišča, ki je definiran z drugo predlogo Voronoijevega diagrama. Imenuje se $Delaunay\_triangulation\_nearest\_site\_2$. Ta nato kliče funkcijo $nearest\_vertex(Point\ q)$ v Delaunayevi triangulaciji.

Rezultat slednje je pravzaprav tisto, kar iščemo od samega začetka, zato se lahko znebimo Voronoijevega diagrama kot vmesnika in direktno uporabljamo Delaunayevo triangulacijo. Poleg poenostavitve kode in rešitve pa je pomemben faktor tudi optimizacija. Teoretična časovna kompleksnost iskanja najbližjega soseda je $\mathcal{O}(\log n)$, povprečna časovna zahtevnost funkcije $nearest\_vertex(Point\ q)$, ki za najdbo najbližjega vozlišča uporablja sprehod po povezavah triangulacije (ang. *line walk*), pa je $\mathcal{O}(\sqrt{n})$. Delaunayeva triangulacija v CGAL-u vsebuje tudi funkcijo $nearest\_vertex(Point\ q,\ Face\_handle\ hint)$, kateri kot drugi argument podamo lice v Delaunayevi triangulaciji, pri katerem se iskanje začne. Če znamo podati dobro začetno lice, se časovna kompleksnost bistveno zmanjša (na k korakov, kjer $k << n$). Problem pri uporabi Voronoijevega diagrama je potem tudi ta, da njena funkcija *locate* ne podpira dodatnega argumenta za začetno lice. Tudi če bi tako funkcijo napisali sami, bi nastal problem pri tem, kako najti primerno lice oziroma kako dostopati do njega. Sama izbira lica v DT pa je očitna. Za točko q, ki jo dobimo iz vrste, in točko p, s katero tvorita povezavo v DT(P) (glej psevdokodo 3.7), imamo za poizvedbo točki p najbližjega vozlišča dve možnosti za dobrega kandidata za začetno lice. Če $q \in DT(W_{i-1})$, je dober kandidat katerokoli lice vozlišča q, sicer pa je to katerokoli lice starša točke q v drevesu SSSP, ker je starš zagotovo vsebovan v $DT(W_{i-1})$.

Za razliko od Voronoijevega diagrama, ki ima samo metodo za vstavljanje ene točke v strukturo, Delaunayeva triangulacija vse-

buje tudi metodo, ki kot argument sprejme seznam točk. Te nato prostorsko sortira, preden jih eno po eno doda v strukturo [4]. Prednost sortiranja je v tem, da so bili deli strukture, ki so obravnavani med vstavljanjem, z veliko verjetnostjo obravnavani tudi pri prejšnjem vstavljanju, in se zato z večjo verjetnostjo nahajajo v pomnilniku *cache* kot v glavnem pomnilniku. Uporaba sortiranja ni optimalna izbira na vseh primerih vhodnih podatkov. Pri našem algoritmu se je izkazalo, da algoritem deluje počasneje, zato smo metodo prilagodili tako, da točk ne sortira. To niti ni toliko presenetljivo: vhodne točke, ki jih DT v i-ti iteraciji dobi iz seznama W_{i-1}, so deloma že smiselno sortirane. Z analizo si poglejmo, kako je zgrajen seznam W_1, ki vsebuje točke z razdaljo 1 od korena r. Za iskanje sosedov vozlišča r uporabimo krožni iterator, ki se začne pri poljubnem sosedu in se nato s pomočjo lic premika do naslednjega. Sosedi si zato sledijo v smiselnem zaporedju, v katerem so tudi shranjeni v seznam W_1. Ko obiščemo vse sosede, moramo obiskati še vse sosede sosedov. Začnemo pri prvem obiskanem sosedu, in zopet s krožnim iteratorjem pogledamo vse sosede, in tako naprej do ostalih. V splošnem velja, da bodo sosedje soseda i bolj blizu sosedom soseda $i+1$ kot soseda $i+2$, in s prvimi bolj verjetno tvorili povezavo v $DT(W_1)$ kot z drugimi. Drugače povedano: za novonastali trikotnik, ki je dodan v triangulacijo, je velika verjetnost, da meji na trikotnik, dodan v prejšnjem koraku.

Za učinkovito iskanje začetnega lica smo napisali razširitev razreda DT, imenovano $DTWithFaceMap$. Hrani zasebni slovar $pointsToFaceHandlers$, ki kot ključ hrani kazalec na objekt tipa $Point\_2$, kot vrednost pa objekt tipa DT_Face_handle. Vsebuje tudi dodatno metodo za vstavljanje, ki kot argument sprejme seznam vozlišč tipa $DTvertex_handle$. Iz vsakega objekta vozlišča dobi ven točko tipa $Point\_2$, ki jo vstavi v DT z metodo $insert(Point\_2q)$, ki je že definirana v razredu DT. Ta metoda vrne nov objekt vozlišča, ki je bilo

dodano v strukturo. Objekt hrani informacijo o točki q in licu f, na katerega vozlišče meji. Par $< q, f >$ nato shranimo v slovar $pointsToFaceHandlers$.

Pomembno dejstvo, ki ga je potrebno izpostaviti, je sledeče. Teoretično gledano je množica točk v DT(W_i) podmnožica točk v DT(P). Ker CGAL razlikuje vozlišča in točke, triangulaciji za vozlišče v, ki ga obe vsebujeta, hranita vsak svoj lasten objekt. Oba objekta vozlišča hranita referenco na isto instanco točke.

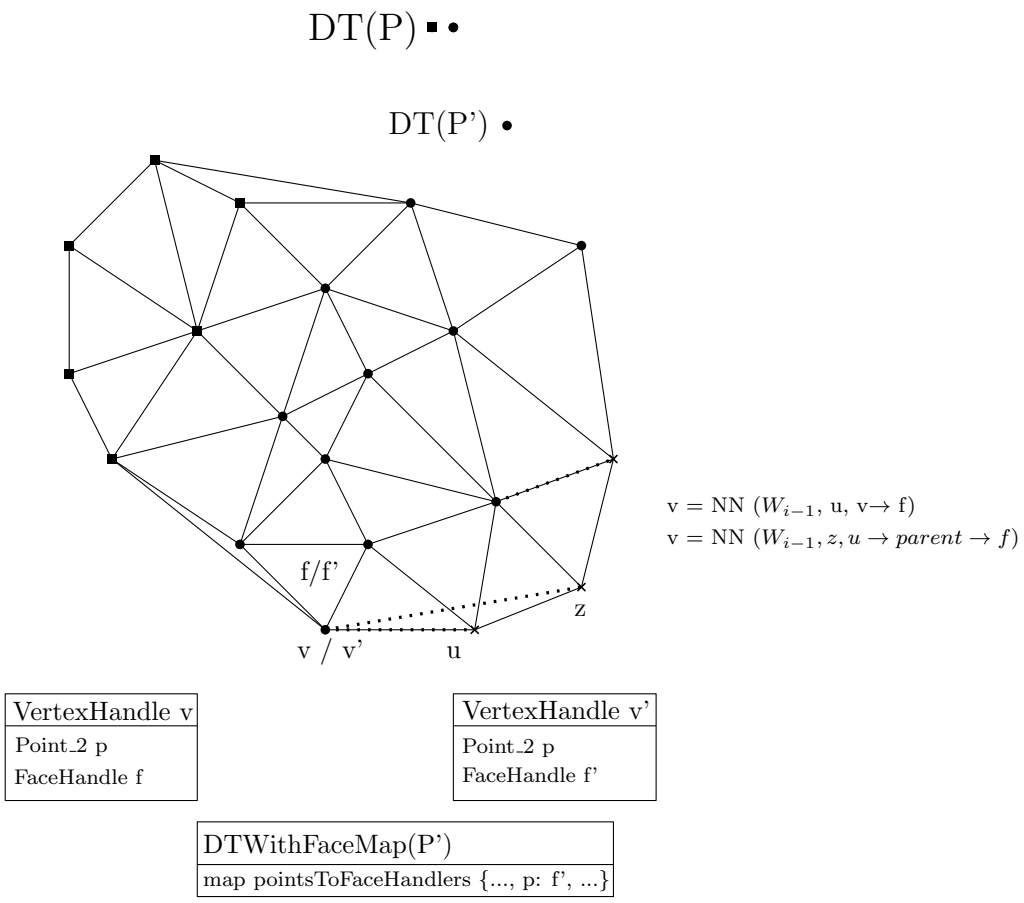
Koda, ki se izvede za posamezno vozlišče p, je sledeča:

```
f = DT::insert((*p)->point(), f)-> face();
pointsToFaceHandlers[&((*p)->point())] = f;
```

```
1  Face_handle getFaceFromPoint(const Point_2* p) {
2   return pointsToFaceHandlers[p];
3  }
4
5  ptrdiff_t insert(vector<DH_vertex_handle> points) {
6   size_type n = this->number_of_vertices();
7   Face_handle f;
8   for (vector<DH_vertex_handle>::iterator p = points.
        begin(); p != points.end(); ++p) {
9    f = DT::insert((*p)->point(), f)->face();
10   pointsToFaceHandlers[&((*p)->point())] = f;
11  }
12  return this->number_of_vertices() - n;
13 }
```

Listing 4.1: sample code

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


DT(P)
DT(P')
f/f'
z
v / v'
u
v = NN (W_{i-1}, u, v→ f)
v = NN ($W_{i-1}, z, u \rightarrow parent \rightarrow f$)
VertexHandle v
Point_2 p
FaceHandle f
VertexHandle v'
Point_2 p
FaceHandle f'
DTWithFaceMap(P')
map pointsToFaceHandlers {..., p: f', ...}

Slika 4.1: .

4.1.3 Izgradnja drevesa

Za drevo SSSP smo napisali razred SSSPTree. Lahko je uporabljen samostojno ali za potrebe problema ločitve. Ponuja dva konstruktorja; prvi je SSSPTree(Iterator begin, Iterator end), drugi pa kot tretji argument sprejme še daljico *st*. Metoda $createTreeFromRoot(Point\_2\ r)$ zgradi drevo s podanim korenom. Če je bil uporabljen drugi konstruktur, se bodo v tej metodi poleg izgradnje drevesa izvedle še dodatne operacije, ki so opisane v naslednjem poglavju.

Konstruktor najprej zgradi DT nad P. Nato za izvorno 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 $\forall p \in P : p.dist = \infty$, $p.parent = nullptr$.

Za generatorja točk kandidatk uporabljamo objekt tipa *deque*, primeren za hranjenje elementov v vrsti. V teoretičnem opisu algoritma smo za $q \in Q$ rekli, da je poljubna točka v Q. V implementaciji je q vedno prva točka v vrsti, kar je bolj optimalno, kot smo omenili v poglavju 4.1.2. Za W_i in W_{i-1} v zanki hranimo seznam Delaunayevih vozlišč (objekte tipa DT_Vertex_Handle). Seznam W_{i-1} vstavimo v strukturo $DTWithFaceMap$.

Za vsak $q \in Q$ poiščemo lice f, ki ga bomo uporabili kot namig pri iskanju najbližjega soseda. Če je razdalja q enaka i, pomeni, da q ni vsebovan v W_{i-1} oziroma strukturi $DTWithFaceMap$. Ker pa je q že v drevesu SSSP, lahko dostopamo do točke qs (tipa $Point\_2$), ki je njen starš. Lice f dobimo tako, da poiščemo vrednost objekta v mapi $pointsToFaceHandlers$, kot ključ pa uporabimo kazalec na starša qs. Če je razdalja q manjša od i, kot ključ uporabimo kar kazalec na točko, ki jo interno hrani q.

Povezave oziroma sosede q najdemo s krožnim iteratorjem, ki ga vrne metoda $incident\_vertices(q)$ v DT. Ker ima v CGAL implementaciji DT poleg standardnih vozlišč še eno neskončno vozlišče, ki je sosedno vsem ostalim,

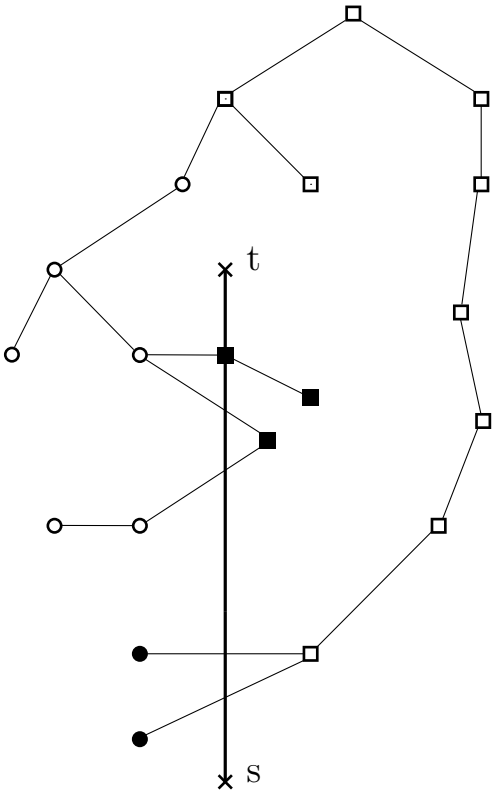
moramo takega soseda ignorirati. Za ostale sosede p najprej preverimo, če je njena točka bila že obiskana. Če ni, poiščemo njenega najbližjega soseda w v W_{i-1} z metodo $nearestVertex$, ki ji kot argument podamo p in lice f. Če $|pw| <= 1$, usvarimo objekt tipa $shared\_ptr < Point\_2 >$, ki hrani kazalec na w. Ta objekt nato nastavimo kot starša točki v p. Točko tudi označimo kot obiskano in ji nastavimo razdaljo i, vozlišče p pa vstavimo v seznam W_i. Na koncu iteracije i seznam W_{i-1} zamenjamo z W_i.

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.

4.2 Minimalna ločitev

4.2.1 Prilagoditev drevesa SSSP

Ko je SSSPTree uporabljen izključno za izgradnjo drevesa, je vse, kar pravzaprav naredimo, to da vsem danim točkam v DT nastavimo kazalec na njihovega starša, ki ga hranijo kot atribut. Če za inicializacijo objekta razreda SSSPTree uporabimo konstruktor, ki poleg točk sprejme kot tretji argument še daljico st, bo metoda za izgradnjo drevesa poleg staršev točkam nastavila še vrednosti atributov nr in jih kategorizirala tako, da jih bo dodala v enega od štirih seznamov tipa $vector$, ki jih razred SSSPTree hrani kot zasebne atribute. Seznami sovpadajo z razredi $L0, L1, R0, R1$, kjer seznam $R0$ hrani točke, ki ležijo desno od daljice st in imajo vrednost atributa nr enako 0. Seznami v resnici kot elemente hranijo zopet sezname, šele ti pa dejansko hranijo točke glede na njihovo razdaljo od korena drevesa. Seznam $R0$ tako na i-tem mestu hrani seznam, ki vsebuje vse točke drevesa z razdaljo i in ostalimi že omenjenimi lastnostmi.



Slika 4.2: Vozlišča drevesa SSSP, ki so označena glede na skupino, v katero so uvrščena glede na daljico st. S krogi so označena vozlišča $L0$, s polnimi krogi vozlišča $L1$, s kvadrati vozlišča $R0$ in s polnimi kvadrati vozlišča $R1$.

Za točko p izračunamo nr ali število presečišč med daljico st in potjo pr (kjer je r koren drevesa) po modulu 2 tako, da prevzamemo vrednost nr njegovega starša q in jo spremenimo samo v primeru, če daljica qp seka st. Za to uporabimo funkcijo v CGAL-u $do\_intersect(seg1, seg2)$. Ker obstaja možnost, da p ali q ležita na daljici st ali sta kolinearna z njenima krajiščema (ker je st vedno navpična daljica, to pomeni, da imajo vse tri enako vrednost koordinate y), je potrebno definirati, kaj storiti v takem primeru. Za točke s tako lego vedno določimo, da so desno od st. V primeru, da p leži na st, moramo zato za spremembo njenega atributa nr poleg presečišč preveriti tudi, da se q in p nahajata na različni strani. $p.nr \neq q.nr$, kjer $p \in st$, velja torej natanko takrat, ko se q nahaja levo od p. S tem dodatnim pogojem preprečimo nepričakovano obnašanje algoritma (glej sliko. NARIŠI PRIMER TREH TOČK qps, KJER JE p NA ST, in pokaži, da imata brez dodatnega pogoja q in s enak nr).

Časovna zahtevnost izračuna vrednosti nr je konstantna. Enako velja za kategorizacijo, ki poleg nr izračuna še orientacijo s predikatom on_left. Za dostop do podseznama, kamor vstavimo točke glede na kategorizacijo in razdaljo od korena drevesa, uporabimo operator [], za vstavljanje točke pa metodo $push\_back(p)$. Oba zagotavljata amortizirano časovno zahtevnost $\mathcal{O}(1)$. S tem, ko sproti izračunamo $N[p]$ za vsako točko drevesa ter podsezname L_i^0, L_i^1, R_i^0 in L_i^1, se znebimo bloka $3-5$ v teoretičnem opisu algoritma, ki bi nam sicer vzel $O(n)$ časa.

4.2.2 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 **??**. Otroka vozlišča, ki se v seznamu nahaja 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 hoteli 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 soseda, 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.

4.2.3 Kd drevo

Za podatkovno strukturo opisano v lemi 3.5 kot primarno strukturo uporabljamo območna drevesa. Za strukturo, ki jo v sekundarnem drevesu območnega drevesa uporabljamo za iskanje najbližjega soseda, smo se namesto VD odločili uporabiti kd drecesa. O slabostih implementacije VD smo govorili že v poglavju 4.1.2. Obe strukturi smo tudi direktno primerjali na podlagi dejanske časovne in prostorske porabe in združili rezultate v tabelah **??** in **??**. DT prav tako ne pride v poštev, ker brez dobrega začetnega lica, ki ga v tem primeru ne moremo zagotoviti, poizvedba s sprehodom vzame $O(\sqrt{n})$ časa.

Kot osnovo smo za poizvedbe najbližjega soseda uporabili funkcijo *search* v razredu Kd_tree, ki je namenjena območnemu iskanju. Razlog za to je, da nas po lemi 3.5 v resnici ne zanima najbližji sosed, ampak poljubna točka, ki je od poizvedbene točke oddaljena največ 1. 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. Podamo mu lahko vrednost ϵ, ki določa stopnjo mehkosti (ang. fuzzyness) in se jo uporablja pri aproksimacijskih poizvedbah, ki pa jih v našem algoritmu nismo potrebovali. Tip, ki smo ga uporabili, je $Fuzzy\_circle$ s središčem, ki ga določa točka poizvedbe in polmerom 1. Časovna kompleksnost funkcije *search* oziroma območnega iskanja v kd drevesu enaka $\mathcal{O}(\sqrt{n}+k)$, kjer je k število vrnjenih točk znotraj območja iskanja. Ker naš algoritem zahteva čas $\mathcal{O}(\log n)$, smo v razredu Kd_tree napisali podobno funkcijo, ki vrne par $(boolean, Point\_2)$. Kot smo omenili, nas v resnici namesto najbližje točke zanima le, če območje iskanja vsebuje kakšno točko. To pomeni, da lahko iskanje po kd drevesu zaključimo v tistem trenutku, ko ugotovimo, da je ta pogoj izpolnjen. Konkretno je ta pogoj izpolnjen takrat, ko za neko poddrevo ugotovimo, da se njegove točke v celoti nahajajo v območju iskanja. Naša metoda v tem primeru vrne par *true* in prvo točko v takem poddrevesu. S tem za poizvedbe zagotovimo povprečno časovno zahtevnost $\mathcal{O}(\log n)$.

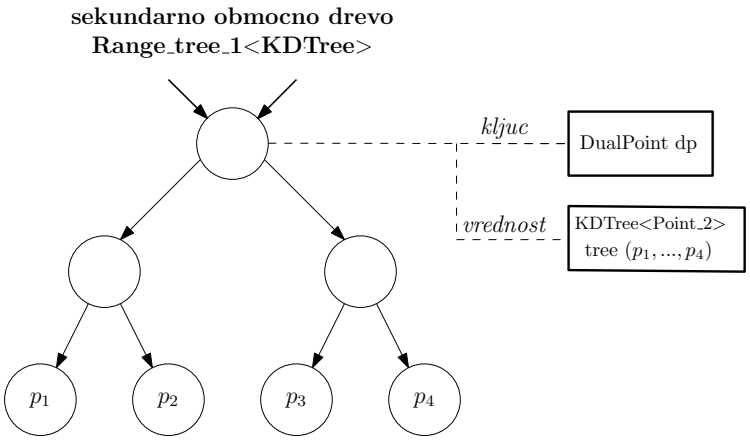
4.2.4 DualPoint - implementacija dualne točke

Odločili smo se razlikovati po tipu med navadnimi točkami in točkami v dualnem prostoru, ki izhajajo iz premic v primarnem prostoru. Za slednje smo naredili razred $DualPoint$, ki 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 **??**), potem $originalPoint$ predstavlja točko b. Konstruktor razreda kot argumenta sprejme točko b ter daljico σ in s pomočjo funkcije $imagePoint(Point\_2b, Segment\_2\sigma)$ izračuna točko preslikave. Kazalec na točko v primarnem prostoru hranimo zato, da ko delamo poizvedbo nad območnim drevesom, ki hrani dualne točke, lahko iz rezultata enostavno dostopamo do primarne točke, ki nas v resnici zanima.

4.2.5 Območno drevo

Za dvodimenzionalna območna drevesa smo uporabili implementacijo v knjižnici CGAL, ki jo predstavlja razred *Range_tree_2*. Vsako vozlišče hrani par *kljuc* in *vrednost*. *Vrednost* dodatno opisuje vozlišče in je lahko tudi prazna, ključ pa se uporablja pri izgradnji drevesa in poizvedbah. Oba argumenta sta predlogi, kar pomeni, da moramo pri konstrukciji drevesa zanju podati konkretna tipa. Za ključ smo uporabili naš razred *DualPoint*, kot vrednost pa vsako vozlišče hrani svoje kd drevo. Zaradi tega smo morali vnesti nekaj sprememb v CGAL-u. Razred za značilnosti območnih dreves *Range_tree_map_traits_2* je parametriziran z modelom predstavitve dvodimenzionalnih točk tipa *Point_2* in s tipom ta vrednost. Drevo torej kot ključ vedno uporabi objekt tipa $Point\_2 < K >$, kjer je K model predstavitve (v naši implementaciji je ta vedno $Cartesian < double >$). Iz razreda značilnosti smo zato odstranili prvi parameter in v njem zamenjali tip ključa na *DualPoint*. Spremeniti smo morali tudi operatorja za primerjavo ključev v razredih *C_Compare_1* (za primarno drevo) in *C_Compare_2* (za sekundarno drevo). Funkciji kličeta predikata *compare_x* oziroma *compare_y* neposredno nad objekti obeh ključev. Ker so objekti v našem primeru tipa *DualPoint*, smo funkciji spremenili tako, da predikatoma poda atribut ključev *point*, ki je tipa *Point_2* in hrani točko v dualnem prostoru.

Pri inicializaciji drevesu podamo seznam parov dualnih točk in praznih kd dreves. Nato se iterativno sprehodimo (tako kot pri preiskovanju v širino) po prvem nivoju drevesa - primarnem drevesu - s funkcijo *traverse_and_populate_with_data*. Za vsako vozlišče pokličemo funkcijo *build_Kd_on_layer*2, ki rekurzivno od spodaj navzgor za vsako vozlišče v sekundarnem drevesu, na katerega kaže vozlišče v primarnem drevesu, vstavi v (do tistega trenutka prazno) njegovo kd drevo točke, ki jih hranita kd drevesi njegovih otrok. Pri listih je v kd drevo vstavljena točka, čigar dualna točka je uporabljena kot ključ vozlišča. Rekurzivni klic funkcije se torej kliče pred vstavljanjem točk v kd drevo. S pomočjo rekurzije je vsakemu notranjemu vozlišču v sekundarnem drevesu seznam točk v njegovem poddrevesu, ki ga



Slika 4.3: .

potrebujemo za izgradnjo njegovega kd drevesa, direktno podan preko kd dreves njegovih otrok. Če bi kd drevesa gradili od zgoraj navzdol v sekundarnem drevesu, bi morali za vsako vozlišče narediti sprehod po njegovem poddrevesu. Primarno drevo za razliko od sekundarnih ne hrani nobenih dreves kot vrednosti v svojih vozliščih.

4.2.5.1 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 vstavljajo 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 soseda v kd drevesih. 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 pogojem 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-

damo vso poddrevo v rezultat, kot je to storjeno v izvirni metodi, temveč opravimo še poizvedbo najbližjega soseda (z območnim iskanjem) točke a v kd drevesu vozlišča v. Če poizvedba vrne neprazen rezultat, lahko iskanje v območnem drevesu zaključimo. 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, vrnemo kot rezultat. Enako velja za obiskana vozlišča v sekundarnem drevesu.

Interval I je odvisen od točke poizvedbe a in tega, ali v drevesu iščemo tako točko b, da ab seka st ali ne. Če ja, potem je območje iskanja drugi kvadrant v dualnem prostoru (glej sliko 3.10), sicer sta to prvi in tretji kvadrant. Za te potrebe smo napisali metodi $rangeTree\_query$ in $rangeTree\_query\_complement$. Pri prvi je iz točke poizvedbe a interval konstruiran kot pravokotnik s spodnjim levim ogliščem $(a.x, -\infty)$ in zgornjim desnim ogliščem $(\infty, a.y)$. Pri drugi metodi sta konstruirana dva intervala: prvi ima za levo krajišče točko $(-\infty, -\infty)$ in za desno krajišče točko a, drugi pa za levo krajišče točko a, za desno pa točko (∞, ∞). Najprej se naredi poizvedba s prvim intervalom in le če slednja ne vrne nobenega rezultata, se poizvedba naredi tudi z drugim intervalom.

4.2.6 Celotna struktura algoritma

Program kot prvi argument sprejme tekstovno datoteko, ki vsebuje po eno točko v vsaki vrstici kot par koordinat. Kot drugi argument je podano začetno krajišče daljice σ, točka s, kot tretji pa točka t kot končno krajišče daljice. Nato se pokliče glavna metoda programa, v kateri se izvedejo vsi izračuni in se okvirno ujema z 3.12.

Najprej se inicializira objekt SSSPTree, ki v konstruktorju zgradi DT. Za vsako točko $r \in P$ se nato zgradi drevo s korenom r. Iz drevesa dobimo ven štiri sezname $L0, L1, R0, R1$. V *while* zanki nato preverimo kandidate v istem vrstnem redu, kot so nanizani v 3.12. Za kandidate na isti strani daljice st se pokliče metoda $nnSeparation$, za ostale pa $rangeSeparation$.

V metodi $nnSeparation$ za vse pare množic točk (S_1, S_2) na isti strani

daljice st in ki tvorijo cikel dolžine $2i$ zgradimo kd drevo nad S_2, za vsako točko $p \in S_1$ pa v drevesu z območnim iskanjem poiščemo prvo točko q, ki je od p oddaljena za največ 1. To naredimo z metodo v razredu Kd_tree, ki smo jo napisali sami. Preprosta optimizacija, ki smo jo dodali v algoritem, je ta, da vnaprej zavržemo tiste pare, pri katerih je ena od množic prazna. S tem bistveno pohitrimo algoritem (glej poglavje 5), saj nima smisla delati poizvedbe nad praznim drevesom ali zgraditi drevesa, nad katerim ne bomo naredili nobene poizvedbe.

V *rangeSeparation* preverimo pare v sledečem vrstnem redu: $L_i^0 \times R_{i-1}^0$, $L_i^1 \times R_{i-1}^0$, $L_i^1 \times R_{i-1}^1$, $L_i^0 \times R_{i-1}^1$, $L_{i-1}^0 \times R_i^0$, $L_{i-1}^0 \times R_i^1$, $L_{i-1}^1 \times R_i^0$, $L_{i-1}^1 \times R_i^1$, $L_i^0 \times R_i^0$, $L_i^0 \times R_i^1$, $L_i^1 \times R_i^0$, $L_i^1 \times R_i^1$. Območnih dreves ne gradimo za vsak par posebej, temveč enkrat za vsako od množic $R_{i-1}^0, R_{i-1}^1, R_i^0, R_i^1$, če je to potrebno. Tudi tukaj namreč ne preverjamo parov, pri katerih je ena od množic prazna. Za pare, kjer iščemo kandidate, ki sekajo st, uporabimo metodo $rangeTree\_query$, za kandidate, ki ne sekajo st, pa metodo $rangeTree\_query\_complement$. Na koncu z metodo *nnSeparation* preverimo še zadnja dva para $L_i^0 \times L_i^1$ in $R_i^0 \times R_i^1$. S takim zaporedjem parov lahko v trenutku, ko naletimo na prvi par, ki vrne neprazen rezultat, prenehamo z iskanjem z ostalimi pari, zaključimo zanko in s tem iskanje cikla za koren r.

4.2.7 Izgradnja minimalnega 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 $T_r^{-1}[a] + T_{r-1}[b]$, ki se vedno stikata v natanko eni točki, tj. korenu drevesa. Z drugimi besedami: najmanjši skupni prednik (ang. *lowest common ancestor*) točk a in b v drevesu najkrajših poti $SSSP(r, P)$ je vedno r. Če to ne bi veljalo in bi bil njun najmanjši skupni prednik točka $r^\star$, bi potem

obstajal krajši cikel v drevesu $SSSP(r^\star, P)$, kjer bi a in b določala minimalno zaprto pot in bi bil njun najmanjši skupni prednik zopet točka $r^\star$. Z združitvijo obeh poti dobimo pot $(a, ..., r, ...b)$. 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.

4.3 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)

```
PSEVDOKODA GLAVNEGA ALGORITMA(Iterator begin, Iterator end, Segment2 st)
 1  (Point2, Point2, int) best = (null, null, MAX VALUE)
    // izračunaj DT
 2  SSSPTree ssspTree(begin, end, st)
 3  for r ∈ P
 4      ssspTree.createTreeFromRoot(r)
 5      vector l0 = ssspTree.getL0()
        // podobno za l1, r0, r1
 6      i = 1
 7      int maxDist = l0.size()
 8      while 2i < best && 2i ≤ maxDist
 9          (Point2, Point2, int) result = nnSeparation(l0, l1, r0, r1, i)
10          if result[2] < best[2]
11              best = result
12              break
13          result = rangeSeparation(l0, l1, r0, r1, i, st)
14          if result[2] < best[2]
15              best = result
16              break
17          i++
        // resetiraj vrednosti atributov v točkah DT
18      ssspTree.resetTree()
19  return makeCycle(best)
```

Slika 4.4: Celotni algoritem minimalne ločitve

```
nnSeparation(vector l0, vector l1, vector r0, vector r1, int i, Segment2 st)
 1  (Point2, Point2, int) best = (null, null, MAX VALUE)
 2  vector L_i^0 = l0[i]
 3  vector L_{i-1}^0 = l0[i - 1]
    // podobno za l1, r0, r1
 4  vector groups = [L_{i-1}^0, L_i^1, L_i^0, L_{i-1}^1, R_{i-1}^0, R_i^1, R_i^0, R_{i-1}^1]
 5  vector pairs = [(0, 1), (2, 3), (4, 5), (6, 7), (1, 2), (5, 6)]
 6  for pair : pairs
 7        vector p0 = groups[pair.first]
 8        vector p1 = groups[pair.second]
 9        if (!p0.empty() && !p1.empty())
10              KDTree kdtree(p1.begin(), p1.end())
11              for q : p0
12                    (bool, Point2) result = kdtree.customSearch(q)
13                    if (result[0])
14                          Point2 nearest = result[1]
15                          best = (q, nearest, q.getDist() + nearest.getDist())
16                          return best
17  return best
```

Slika 4.5: Prilagoditev splošnega algoritma za izračun minimalne ločitve za enotske diske.

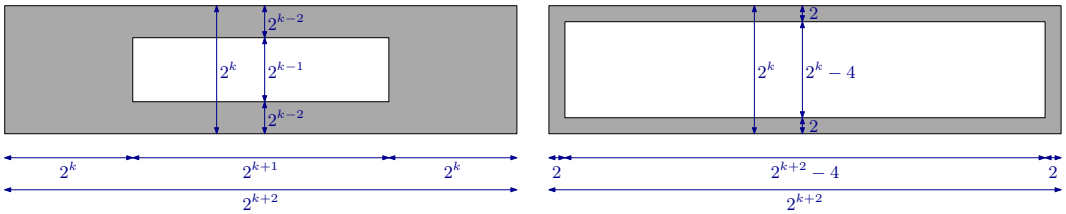
Poglavje 5

Eksperimenti in rezultati

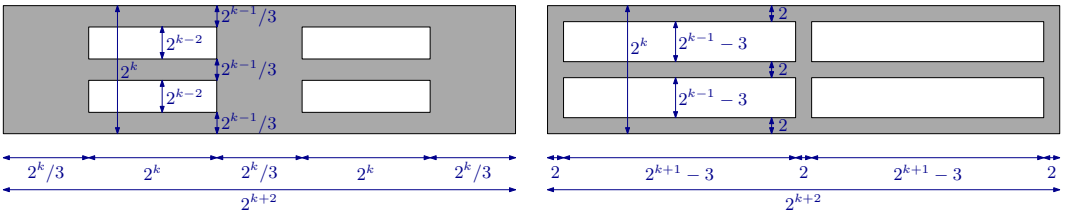
Eksperimente smo izvedli na prenosnem računalniku z operacijskim sistemom Windows 10, procesorjem CPU i5-5200U z 2.20 Ghz in 8GB spomina. Za vizualizacijo domen, vhodnih podatkov in rezultatov smo napisali skripte v Pythonu in pri tem uporabili knjižnico *Matplotlib*.

5.1 Generator vhodnih podatkov

Točke smo generirali naključno znotraj sledečih poligonskih domen: pravokotnik brez lukenj, pravototnik z manjšo luknjo, pravokotnik z večjo luknjo, pravokotnik s štirimi manjšimi luknjami in pravokotnik s štirimi večjimi luknjami. Natančne dimenzije domen so prikazane na slikah 5.1 in 5.2. Konkretno smo uporabili domene, kjer je imel zunanji pravokotnik dimenzije 4×1, $8\times2,\ldots,128\times32$. Velikost vhodnih podatkov je bila 1K, 2K, 5K, 10K, 20K



Slika 5.1: .



Slika 5.2: .

in 50K točk. Podatke smo generirali enkrat in nato zapisali v tekstovno datoteko. Pri testiranju problema minimalne ločitve smo točko s postavili na sredino luknje, t pa vertikalno nad s in zunanjim pravokotnikom. Prvi dve točki v tekstovni datoteki, ki hrani vhodne podatke, sta vedno obravnavani kot točki s in t. Domen s premajhnimi luknjami nismo uporabili za problem minimalne ločitve, ker je višina lukenj manjša od 1 in bi drevo najkrajših poti vsebovalo povezave med točkama na nasprotnih straneh luknje. S tem bi se vloga luknje izgubila in bi kot rešitev dobili trivialni cikel.

Pri večjih domenah in manjših vhodnih podatkih je bil izziv zapolniti prostor, ki je pogoj za obstoj rešitve minimalnega problema, saj so točke bolj razpršene. Za vhodne podatke sicer nismo zahtevali pogoja, da je njihov graf G povezan, smo pa želeli, da je velikost največje komponente G dovolj podobna velikosti celotnega grafa G. S tem smo povečali možnost obstoja rešitve in dosegli, da smo dejansko testirali želeno število vhodnih podatkov. Nekatere kombinacije (na primer 1000 točk na domeni z zunanjim pravokotnikom 128×32) smo zato izpustili, pri drugih pa smo problem rešili tako, da smo za začetnih m generiranih točk preverili, da imajo vsaj k sosedov (točk, od katerih so oddaljene za največ 1) med trenutnimi vhodnimi podatki. Če pogoj ni bil izpolnjen, smo generirano točko zavrgli. S tem smo generator na začetku spodbudili k zapolnjevanju prostora domene. Vrednosti m in k smo nastavili za vsako kombinacijo posebej s pomočjo kalibracije. k se je gibal med 3 in 5 (manjši k pomeni hitrejše zapolnjevanje prostora, zato smo ga uporabili pri večjih domenah), m pa okrog številke 500.

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.

5.2 Drevo najkrajših poti

Implementacijo algoritma SSSP smo primerjali z dvema očitnima alternativnima algoritmoma za drevesa najkrajših poti. Pri prvi eksplicitno zgradimo graf $G = G_{\leq 1}(P)$ tako, da za vse možne pare $p, q \in P$ dodamo povezavo pq v strukturo grafa, če je razdalja med p in q največ 1. Za dan koren r nato za izgradnjo drevesa uporabimo splošno iskanje v širino (BFS). Predprocesiranje (izgradnja grafa $G_{\leq 1}(P)$) ima kvadratno časovno zahtevnost, čas izgradnje drevesa pa je odvisen od gostote grafa. Za vsako točko iz vhodnega seznama, ki ga nespremenjenega hranimo ves čas, zgradimo vozlišče. Za vozlišče grafa smo napisali strukturo $GraphNode$, ki hrani točko tipa $Point\_2$, kazalec na starša in sosede kot seznam indeksov, ki določajo položaj soseda/točke v vhodnem seznamu. Za vsako vozlišče preverimo razdaljo do ostalih vozlišč (brez simetričnih primerov; skupno preverimo torej $\frac{n(n-1)}{2}$ parov) in za vsak par vozlišč z razdaljo največ 1 dodamo drug drugega v seznam sosedov. Drevo najkrajših poti potem zgradimo s klasičnim algoritmom za preiskovanje v

širino (glej 3.3).

Kot drugo alternativo uporabimo mrežo enotskih razdalj (glej poglavje 2.4). Dve točki (x, y) in (x', y') se nahajata v isti celici mreže natanko takrat, ko velja $(\lfloor x \rfloor, \lfloor y \rfloor) = (\lfloor x' \rfloor, \lfloor y' \rfloor)$. Vse točke v celici c shranimo v seznam $\ell(c)$. Vse neprazne sezname $\ell(c)$ shranimo v slovar, v katerem je kot ključ uporabljeno spodnje levo oglišče celice. S tako strukturo nato izvedemo nekakšno preiskovanje v širino. Seznam $\ell(c)$ celice c hrani točke, ki še niso bile obiskane z drevesom najkrajših poti. Ko procesiramo točko p v celici c, moramo kot kandidate obravnavati vse ostale točke v $\ell(c)$ in točke v seznamih osmih sosednih celic. Vsaka točka, ki je sosedna p, je nato odstranjena iz seznama svoje celice. Predprocesiranje (izgradnja slovarja) ima linearno časovno zahtevnost, čas izgradnje drevesa pa je odvisen od porazdelitve točk. Enostavno se da najti primere, kjer bi bil ta čas kvadratičen. Za vsako najkrajše drevo slovar zgradimo od začetka. Ker je čas, ki ga porabimo za to, zanemarljiv, čas predprocesiranja za mrežo ni prikazan med rezultati. Za seznam točk v celicah smo pri implementaciji uporabili strukturo *list* in ne *vector*. Razlog za to je ta, da je vsaka točka slej ko prej odstranjena iz seznama svoje celice. *vector* v C++ omogoča samo brisanje vsebine elementa na i-tem mestu, ne pa brisanja samega elementa oziroma je časovna kompleksnost tega $O(n)$. Po drugi strani *list* porabi nekaj več spomina, ker vsak element hrani še kazalca na prejšnji in naslednji element, ima enak čas vstavljanja elementa na konec seznama kot *vector*, vendar je časovna kompleksnost brisanja (in vstavljanja) elementa sredi seznama $O(1)$, ker samo preveže njegovega predhodnika in naslednika. Za slovar celic smo uporabili strukturo *unordered_map*, ki kot ključ hrani par celih števil, ki predstavljajo spodnje levo oglišče celice. Za ključ smo morali definirati tudi dobro *hash* funkcijo, odporno na trke (ang. *collisions*). Uporabili smo Cantorjevo funkcijo $\pi(k_1, k_2) = \frac{1}{2}(k_1 + k_2)(k_1 + k_2 + 1) + k_2$.

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.

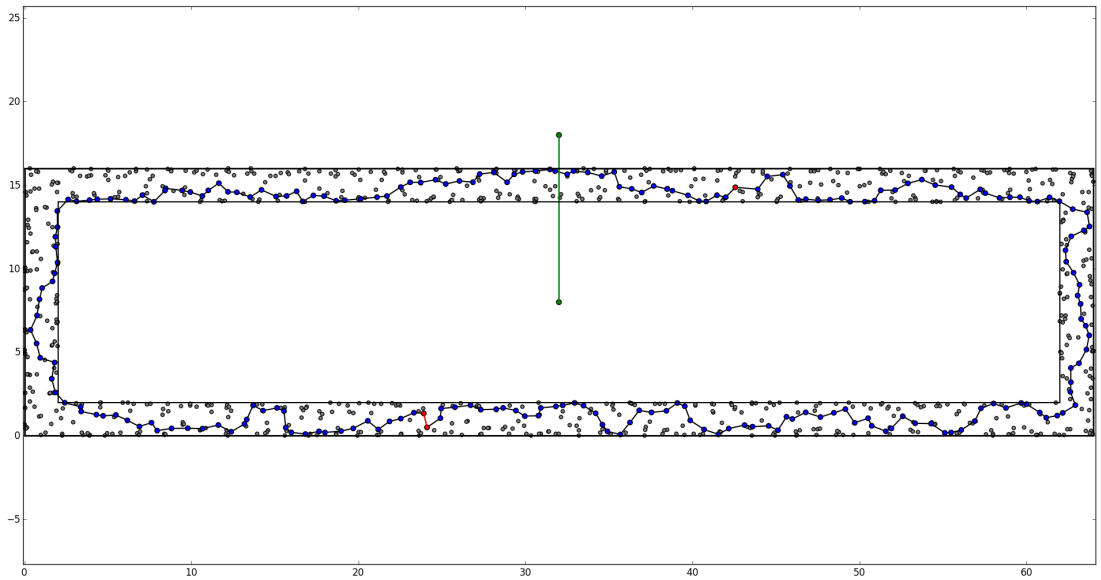
5.3 Minimalna ločitev

Kot merilo uspešnosti algoritma za minimalno ločitev smo njegove rezultate primerjali z rezultati splošnega algoritma za ločevanje, opisanega v poglavju 3.2.1. Za potrebe implementacije splošnega algoritma smo razredu $Point\_2$ dodali še atribut $neighbours$, ki za posamezno točko hrani seznam kazalcev na njene sosede v grafu G. Najprej z inicializacijo objekta $SSSPTree$ zgradimo DT(P). Nato se algoritem z iteratorjem sprehodi čez vozlišča triangulacije, pred katerih lahko dostopamo do objektov točk. Za vsako vozlišče u gremo ponovno skozi iterator vozlišč, le da ima ta začetek pri trenutnem vozlišču (skupno število iteracij je torej $\frac{n\times(n-1)}{2}$). Za vsako vozlišče v druge iteracije preverimo $dist(u.p, v.p)$ in če je ta največ 1, v seznam $u.p.neighbours$ dodamo kazalec na točko $v.p$. Ker se drugi iterator ne začne s prvim vozliščem, povezavo uv shranimo samo enkrat.

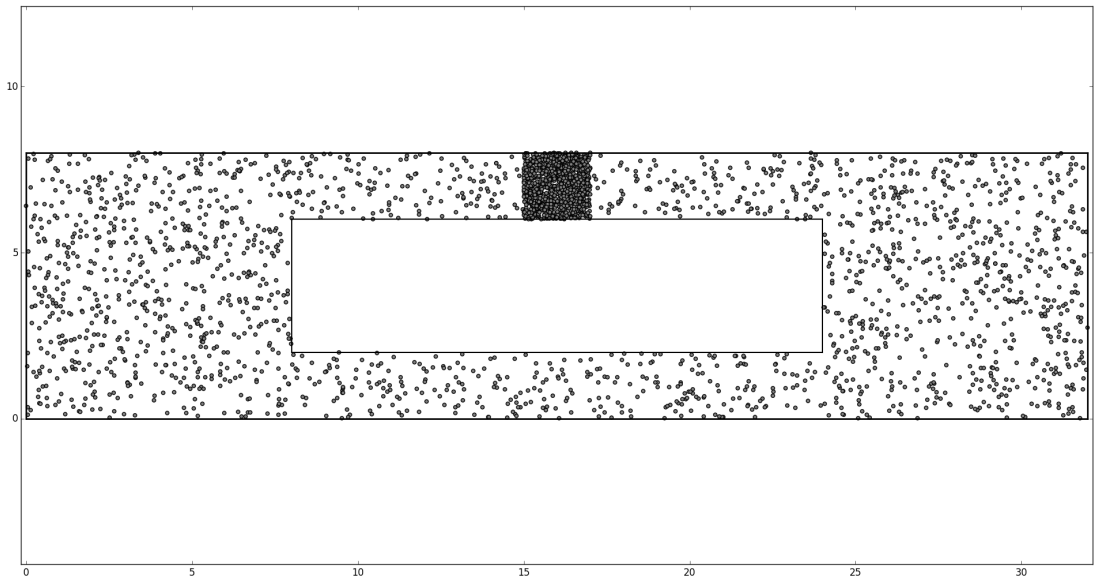
V glavni zanki se zopet sprehodimo skozi iterator vozlišč in za vsako njegovo točko r zgradimo drevo $SSSPTree(r)$. S tem se za vsako točko p hkrati izračuna tudi število presečišč poti rp z st po modulu 2. Z dodatno zastavico v konstruktorju drevesa smo izklopili kategorizacijo točk v štiri sezname, saj tukaj ni potrebna. Čez povezave $E(G)$ se sprehodimo tako, da gremo zopet z iteratorjem čez vsa vozlišča in za vsako točko vozlišča preverimo vse njegove sosede s pomočjo seznama kazalcev.

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

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



Slika 5.3: Primer rezultata - najkrajšega cikla - algoritma za minimalno ločitev. Modre točke so del poti cikla, samostojna rdeča točka predstavlja koren drevesa r, s katerim smo našli cikel, par rdečih točk p, q pa z rdečo povezavo najkrajši poti od r do p in r do q združi v cikel. Zelena daljica predstavlja daljico st.



Slika 5.4: Primer eksperimenta, kjer se nov algoritem za ločevanje odreže bistveno bolje od splošnega algoritma. 2000 izvirnim točkam smo dodali 1000 novih točk blizu st, ki s tem bistveno povečajo velikost množice $E(G(P))$.

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

Tabela 5.8: Časi za minimalno ločevanje s 4 luknjami.

Pravokotnik brez lukenj	50K točk		
dimenzije pravokotnika	4×1	8×2	16×4
SSSP	25.2	22.3	20.1
mreža	23.2	22.4	22.2
	32×8	64×16	128×32
SSSP	19.3	19	18.7
BFS	204.2	68.8	28.5
mreža	21.9	22	23

Tabela 5.9: Max ram brez lukenj.

5.3.1 Analiza

Za razliko od SSSP algoritma BFS in mreža nimata nikakršne garancije za najslabši primer. Za oba lahko zgradimo takšne vhodne množice P, pri katerih bo njuna izvedba zelo počasna. Kot primer smo testirali P s 10000 točkami v pravokotniku, ki ima dimenzije 32×8 in majhno luknjo. Med krajiščema daljice st smo nato v majhnem zgoščenem prostoru dodali še dodatnih 1000 točk. Širina tega prostora je 1, tako da se število povezav v grafu G bistveno poveča. Čas izvajanja je glede na prvotno instanco najmanj narastel za SSSP, največ pa za BFS (glej tabelo 5.12).

Pravokotnik s 4 velikimi luknjami	5K točk		
dimenzije pravokotnika	32×8	64×16	128×32
SSSP	3.1	2.8	2.5
BFS	4.2	2.9	2.3
mreža	2.7	2.7	2.9
	10K točk		
SSSP	5.6	5.0	4.5
BFS	12.6	7.6	5.3
mreža	4.9	4.9	5.1

Tabela 5.10: Max ram - velike luknje

Pravokotnik s 4 luknjami	2K točk, manjše luknje		
dimenzije pravokotnika	32×8	64×16	128×32
nov algoritem za ločevanje	5.4	3.3	2
splošni algoritem	2.9	2.1	1.8
	5K točk, večje luknje		
nov algoritem za ločevanje	12.9	8.1	5.8
splošni algoritem	8.7	6	4.2

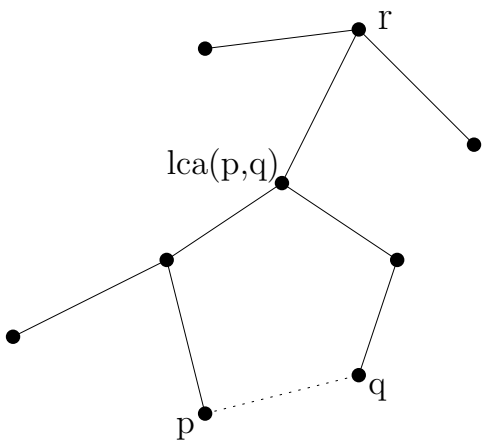
Tabela 5.11: Max ram - velike luknje

	Časovna porast za 11K točk (1K zgoščenih)
predprocesiranje SSSP	9.7%
povprečje za koren s SSSP	13.6%
predprocesiranje BFS	21.9%
povprečje za koren z BFS	56.5%
mreža	25%

Tabela 5.12: Časi za minimalno ločevanje s 4 luknjami.

Podobno smo storili z algoritmom za minimalno ločevanje. Uporabili smo enako območje s 500 in 1000 dodatnimi točkami. V prvem primeru nov algoritem porabi 94 sekund (skoraj trikrat več kot brez dodatnih točk), splošni algoritem pa 435 (šestkrat več kot brez dodatnih točk). V drugem primeru nov algoritem porabi 173 sekund, splošni algoritem pa več kot 15 minut. Podobno kot pri eksperimentih z drevesi najkrajših poti se tudi tukaj nov algoritem izkaže bolje v primerih, kjer so točke bolj gosto poseljene in je med njimi več povezav. Delež dreves, ki jih moramo zgraditi, da najdemo najkrajši cikel, je odvisen od velikosti lukenj oziroma širine območja, kjer se točke nahajajo, kar vpliva na dolžino cikla. Pri ožjih območjih je manjše število možnih poti in verjetnost, da naletimo na tako drevo, čigar koren je del najkrajšega cikla, je s tem večja. Zaradi večjih lukenj je tudi dolžina cikla večja in je tako razmerje med številom točk v ciklu in številom vseh točk večji.

Zaradi omenjenih lastnosti eksperimentov in enakomerne razporeditve točk v njih je najkrajši cikel najden praktično v trenutku. Ker je večina možnih ciklov podobne dolžine kot najkrajši cikel, se pri eksperimentih tudi ne izrazi optimizacijski doprinos pogoja $2i < best$ v glavni zanki algoritma. Razliko bi lahko ponazorili z eksperimentom brez lukenj, kjer bi zelo malo ciklov imelo enako dolžino. Zaradi velike variacije dolžin ciklov in dolžine najkrajšega cikla slednjega ne bi odkrili tako hitro, bi pa z omenjenim pogojem velik delež ciklov vnaprej zavrgli.



Slika 5.5: Primer drevesa s korenom r in ciklom, ki ga tvorita poti od p do r in q do r, kjer pa obstaja tudi krajši cikel v poddrevesu s korenom $lca(p, q)$.

Še ena možna optimizacija algoritma, ki je nismo naknadno implementirali, ker se ne obnese v naših eksperimentih, bi bila za najden trenutno najkrajši cikel, ki ga tvorita točki p in q, ugotoviti, če je koren drevesa enak najmanjšemu skupnemu predniku $lca(p, q)$ obeh točk v drevesu (glej sliko 5.5). Če temu ni tako, potem vemo, da obstaja krajši cikel, ki ga dobimo z drevesom, ki ima za koren točko $lca(p, q)$, zato tak cikel določimo kot trenutno najkrajšega. Do takega drevesa bi sicer prišli slej ko prej, a pri primerih, kjer imajo cikli zelo različne dolžine, lahko s pogojem $2i < best$ do takrat, ko pridemo do tega drevesa, zavržemo dosti ciklov vnaprej.

Pri porabi spomina se SSSP v povprečju odreže najbolje. Splošni BFS se odreže bistveno slabše pri manjših območjih, kjer je število povezav v grafu G večje. Rezultati mreže ne odstopajo znatno. Pri bolj gostih območjih se izkaže malenkost bolje kot SSSP, najmanjšo porabo pa ima pri srednje velikih območjih. Je edini od algoritmov, ki mu poraba spomina malenkost naraste pri največjem območju. To gre pripisati temu, da je zaradi enakomerne razporeditve točk potrebno zgraditi več celic v večjih območjih.

Poraba spomina novega algoritma za ločevanje je po drugi strani nekoliko večja kot pri splošnem algoritmu. Razmerje sicer malenkost pade pri 5K točkah glede na 2K točk. Porabo spomina smo približno izmerili tudi na

območju 32×8 z eno luknjo in 20K točk (približno zato, ker porabe nismo merili celotni čas izvajanja, ki bi bil precej velik). Novi algoritem porabi 45MB spomina, splošni pa 85MB. Razlog za večjo porabo spomina novega algoritma pri manj točkah najbrž leži v tem, da kljbu manjši prostorski kompleksnosti nekaj dodatnega spomina zasede sama izgradnja CGAL struktur območnih in kd dreves.

Poglavje 6

Sklepne ugotovitve

Čeprav se algoritem za minimalno ločevanje obnaša bistveno bolje od splošnega algoritma, ko so točke gosto poseljene, temu ni tako, ko je gostota točk majhna. To smo videli tudi pri rezultatih eksperimentov. Implementirali bi lahko tudi prilagojen algoritem, ki bi se obnašal različno glede na velikost množic L_i^j in R_i^j, ki je pogosto precej majhna. To smo deloma tudi storili tako, da smo izpustili obravnavo tistih parov, kjer je bila ena od množic prazna. Lahko pa bi šli tudi korak dlje. Z eksperimenti bi lahko našli točko preloma za velikost obeh množic, pri kateri se začne splačati uporabiti kar "brute force"metodo in preveriti vse možne pare $(p, q), p \in L_i^j, q \in R_i^j$. Velik vpliv na to ima predvsem velikost L_i^j, torej število poizvedb. Pohitritev, ki jo dobimo s takšnim algoritmom, bi lahko demonstrirali z eksperimenti, kjer so točke neenakomerno razporejene. S tako množico točk P bi imeli tako gosta kot redka območja točk in algoritem bi se obnašal glede na to, na kakšnem območju se nahajajo točke v L_i^j in R_i^j. Namesto iskanja točke preloma bi lahko uporabili tudi preprosto hevristiko s časovno zahtevnostjo obeh metod. Naj bo n velikost množice L in m velikost množice R. Uporaba kd dreves bi se s hevristiko potem splačala, če $(n + m) \log m < nm$ (kjer $(n + m) \log m$ predstavlja skupno časovno zahtevnost izgradnje kd drevesa z m elementi in n poizvedb, nm pa časovno zahtevnost *brute force* metode). Podobno bi se območna drevesa splačala, če $(n + m) \log^3 m < nm$.

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] S. Cabello and M. Jejčič. Shortest paths in intersection graphs of unit disks. *Comput. Geom.*, 48(4):360–367, 2015.

[4] C. Delage and O. Devillers. Spatial sorting. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 edition, 2015.

[5] O. Devillers. Improved incremental randomized delaunay triangulation. In *Proceedings of the 14th Annual ACM Symposium on Computational Geometry*, pages 106–115, 1998.

[6] 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.

[7] M. Karavelas. 2D voronoi diagram adaptor. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 edition, 2015.

[8] Y. Wang, J. Gao, and J. S. Mitchell. Boundary recognition in sensor networks by topological methods. In *Proceedings of the 12th Annual In-*

ternational Conference on Mobile Computing and Networking, MobiCom '06, pages 122–133, 2006.