Univerza v Ljubljani
Fakulteta za matematiko in fiziko
Fakulteta za računalništvo in informatiko

Lazar Milinković

Implementacija algoritmov za probleme najkrajših poti v geometrijskih grafih

magistrsko delo

univerzitetni študijski program druge stopnje
računalništvo in matematika

Mentor: dr. Sergio Cabello

Ljubljana 2016

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

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

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

Izjava o avtorstvu magistrskega dela

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

Implementacija algoritmov za probleme najkrajših poti v geometrijskih grafih

S svojim podpisom zagotavljam, da:

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

V Ljubljani, dne 11. januarja 2016 Podpis avtorja:

Kazalo

Povzetek

Abstract

Poglavje 1

Uvod

V poglavju 2 je na kratko opisana programska knjižnica CGAL. Osredotočili smo se na osrednje ideje, uporabljene pri jedru knjižnice, na katerem temeljijo vsi ostali deli paketa, ter tiste strukture, ki so bile uporabljene pri implementaciji našega algoritma. V poglavju 3 je predstavljen teoretični del algoritma. Podrobno so opisane vse podatkovne strukture, celotni potek algoritma ter njegova časovna in prostorska kompleksnost. V poglavju 4 je predstavljena implementacija algoritma. Ponovno so opisane vse uporabljene podatkovne strukture in nekatere spremembe v header datotekah CGAL-a z razredi, ki predstavljajo te strukture. Podrobno so opisani tudi deli algoritma, kjer se pojavijo razlike med teoretičnim opisom in implementacijo. V poglavju 5 so predstavljeni rezultati; prikazani so časi izvajanja in prostorska poraba celotnega algoritma za različno število vhodnih točk. Enako smo storili posebej za nekatere podatkovne strukture, npr. Kd drevesa, Voronoijev diagram in SSSP drevesa.

Poglavje 2

CGAL in uporabljene matematične strukture

V tem poglavju so na kratko opisani programska knjižnica CGAL in podatkovne strukture, uporabljene v naših 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.

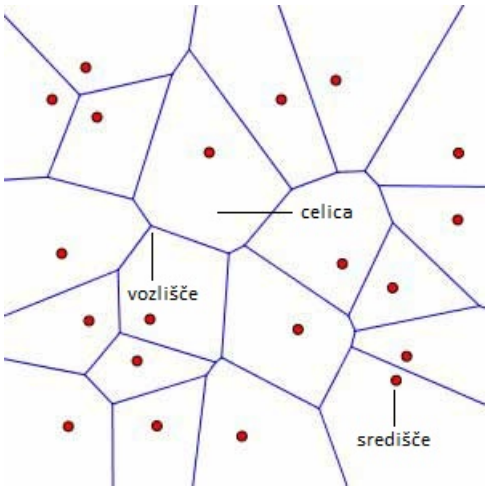
??? natancnost: cartesian, kaj pa leda, homogenous? povej kaj o traits, templated strukturami ($Point_2$)

2.2 BFS drevesa

2.3 SSSP drevesa

2.4 Graf kvadratne mreže

$https://en.wikipedia.org/wiki/Lattice_graph_Square_grid_graph$



Slika 2.1: Evklidski Voronoijev diagram brez degeneracij.

2.5 Delaunayeva triangulacija

2.5.1 Implementacija v CGAL

2.6 Voronoijev diagram

Voronoijev diagram [1] 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:

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

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

$$V_i = \bigcap_{i \neq j} V_{ij}. \tag{2.2}$$

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 2.1), ki ga uporabljamo v napem algoritmu.

2.6.1 Voronoijev diagram v CGAL

Knjižnica CGAL ponuja 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 primerek, ki predstavlja koncept razreda DelaunayGraph_2. Primeri takih struktur so Delaunayeva triangulacija, navadna triangulacija in Apollonov graf. Druga predloga, AT, predstavlja lastnosti prilagoditve ter definira tipe struktur in funktorje, ki jih razred potrebuje za dostop do geometrijskih lastnosti Delaunayeve triangulacije. Funktor mora biti recimo definiran za konstrukcijo

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

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

2.7 Območna drevesa

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

2.8 Kd-drevesa

(2.3)

Poglavje 3

Teorija

3.1 Predstavitev problema

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

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

$$\begin{aligned} \min \quad & |\mathcal{D}'| \\ \text{tako da} \quad & \mathcal{D}' \subset \mathcal{D} \\ & \mathcal{D}' \text{ ločuje } z \text{ in } z'. \end{aligned}$$

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

Naj bo P množica točk, ki predstavljajo središča krogov $\mathcal{D}$. P predstavlja vhod našega algoritma in vse operacije ter uporabljene podatkovne strukture

se vrtijo okoli te množice. Kardinalnost množice je enaka n.

Naj bo $G(P)$ graf z množico vozlišč P s povezavo med točkama $p, q \in P$, če velja $|pq| <= 1$ 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; če poleg tega tudi vertikalna ni, naredimo še rotacijo (kjer pa lahko pride do numeričnih napak).

3.2 SSSP drevo

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

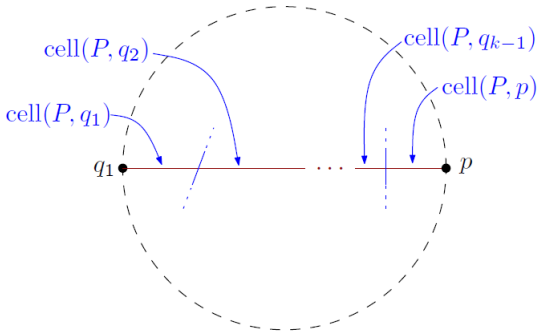
Algoritem dobi kot vhod poleg množice P izvorno točko $s \in P$ in nato inkrementalno v vsaki iteraciji dodaja točke h grafu. Množico točk, dodanih k drevesu v i-ti iteraciji, označimo kot

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

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

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

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



Slika 3.1: aa

Lema 3.1. *Naj bo $p \in W_i$. V $G(P) \cap DT(P)$ obstaja pot $q_1, ..., q_k = p$, kjer $q_1 \in W_{i-1}$ in $q_2, ..., q_k \in W_i$.*

Dokaz. ??? □

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

Celotna psevdokoda algoritma za izgradnjo drevesa je opisana spodaj.

Lema 3.2. *Za izgradnjo drevesa SSSP potrebuje algoritem $O(n)$ prostora in $O(n \log n)$ časa.*

Dokaz. ??? □

3.3 Drevo najbližjega soseda

Uravnoteženo binarno iskalno drevo. Opis vzami iz "main", namesto VD pa reci vd oz drugo primerno strukturo za iskanje najbližjega soseda. Omeni, da je bil izbran kd-tree.

Lema 3.3. *Naj bo P množica uteženih točk na ravnini. V času $O(n \log n)$ lahko konstruiramo tâko podatkovno strukturo, ki v času $O(\log^2 n)$ za točko*

Algorithm 1 Algoritem za izgradnjo SSSP drevesa

```
1:  procedure buildTree
2:      for p ∈ P do
3:          dist[p] ← ∞
4:      dist[r] ← 0
5:      zgradi Delaunayevo triangulacijo DT(P)
6:      W_0 ← {s}
7:      i ← 1
8:      while W_{i−1} ≠ ∅ do
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 ≠ ∅ do
13:             q naj bo poljubna točka v Q
14:             odstrani q iz Q
15:             for povezava qp v DT(P) do
16:                 if dist[p] = ∞ then
17:                     w ← NN(W_{i−1}, p)
18:                     if |pw| ≤ 1 then
19:                         dist[p] ← i
20:                         dodaj p v Q in W_i
21:         i ← i + 1
22:     return dist[·]
```

poizvedbe q najde točko v

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

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

Za vsako vozlišče ν v $\mathcal{T}$:

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

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

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

Potrebna je še analiza časa poizvedbe. Za točko poizvedbe q preverimo, če je vsebovana v $U(r)$, kjer je r koren drevesa. Če ni, potem nobena točka iz P ni dovolj blizu q. Sicer označimo $\nu = r$ in nadaljujemo z iskanjem od vrha navzdol po T, dokler ne pridemo do listov drevesa. Ko ν ni list,

označimo njegovega levega in desnega otroka z ν_ℓ in ν_r. Če se q nahaja v $U(\nu_\ell)$, označimo $\nu = \nu_\ell$ in nadaljujemo navzdol po poddrevesu otroka ν_ℓ, sicer to storimo za desnega otroka. V vsakem trenutku poizvedbe ohranjamo sledečo invarianto:

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

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

3.3.1 Optimizacijski problem 1

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

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

Lema 3.4. *Naj bosta A and B množici največ n uteženih točk v ravnini. $\Phi(A,B)$ lahko izračunamo v času $O(n \log^2 n)$.*

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

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

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

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

3.3.2 Optimizacijski problem 2

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

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

Za vsako točko $a \in A$ definiramo množici

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

in optimizacijski problem

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

Če združimo oba optimizacijska problema skupaj, dobimo

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

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

Dualnost in dualni prostor

Opiši iz poglavja knjige.

Lema 3.5. *Obstaja družina* $\{B_1, \ldots, B_t\}$ *podmnožic množice* B *in podatkovna struktura* $\mathcal{D}(B)$ *z naslednjimi lastnostmi*

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

Dokaz. Za potrebe dokaza uporabimo dualnost, opisano zgoraj.

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

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

V dualnem prostoru je množica σ^* *horizontalslab*

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

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

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

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

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

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:

$$b_1 \in \{(x,y) \mid (x,y) \in B, y < 0\} \Rightarrow \varphi_1(b), \varphi_2(b) < 0 \text{ and } \varphi_1(b) > \varphi_2(b)$$
$$b_2 \in \{(x,y) \mid (x,y) \in B, 0 <= y < s\} \Rightarrow \varphi_1(b) > 0,\ \varphi_2(b) < 0$$
$$b_3 \in \{(x,y) \mid (x,y) \in B, y >= s\} \Rightarrow \varphi_1(b) > 0,\ \varphi_2(b) >= 0 \text{ and } \varphi_1(b) > \varphi_2(b)$$

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 $\varphi_1(b) >= \varphi_2(b)$ 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:

$$ab \cap \sigma \neq \emptyset \iff (\varphi_1(a) - \varphi_1(b)) \cdot (\varphi_2(a) - \varphi_2(b)) < 0$$
$$\varphi_1(a) <= \varphi_1(b)$$
$$\varphi_2(a) >= \varphi_2(b).$$

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 $\varphi_1(b) >= \varphi_2(b)$. Glej sliko.)

Za shranjevanje množice točk $\varphi(B)$, kjer je vsaka točka $b \in B$ asociirana s točko preslikave $\varphi(b)$, lahko uporabimo dvodimenzionalno območno drevo. Za vsako točko poizvedbe $a \in A$ lahko točke $b \in B$, kjer ab seka σ, dobimo s poizvedbo na območnem drevesu, ki vrne točke $\varphi(B)$ v kvadrantu

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

Območna drevesa so bolj podrobno opisana v poglavju x.

□

Poglavje 4

Implementacija algoritma

4.1 SSSP drevo

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

Za implementacijo smo uporabili razred $SSSPTree$ s štirimi zasebnimi atributi. Vsi so 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 Dodatki v razredu Point_2

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

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

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

4.1.2 Voronoijev diagram za iskanje najbližjega soseda

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

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

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

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

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

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

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

4.1.3 Izgradnja drevesa

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

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

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

- $p.setDist(i)$
- $Q.insert(p)$
- $Wi.insert(p)$
- $p.setParent(shared\_ptr\langle Point\_2\rangle\ w)$
- $p.setNr(updateNr(w,\ p,\ st))$
- $categorize(p,\ st)$

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

4.2 Drevo najbližjega soseda

Drevo najbližjega soseda, opisano v poglavju 3.3, bi lahko implementirali s kazalci ali seznamom. Odločili smo se slednjega. Seznam hrani 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 $O(n)$, 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.

4.2.1 Kd drevo

Kot osnovo smo za poizvedbe najbližjega soseda uporabili funkcijo *search* v razredu *Kd_tree*, ki kot argument sprejme *OutputIterator*, kamor se shranjujejo objekti, ki jih poizvedba vrne, in *FuzzyQueryItem*, ki je v dvodimenzionalnem primeru lahko krog ali pravokotnik in določa območje iskanja. Kot argument med drugim sprejme vrednost ϵ, ki določa stopnjo mehkosti

(ang. fuzzyness) in se jo uporablja pri aproksimacijskih poizvedbah. Časovna kompleksnost take poizvedbe je enaka $O(\log n + k)$, kjer je k število vrnjenih točk znotraj območja iskanja. Ker naš algoritem zahteva $O(\log n)$, smo razredu *Kd_tree* dodali podobno funkcijo, ki pa iskanje zaključi v istem trenutku, ko ugotovi, da se vsaj ena točka nahaja znotraj območja iskanja. Če na primer za neko notranje vozlišče Kd drevesa ugotovi, da vse točke v njegovem poddrevesu ustrezajo kriterijem iskanja, jih ne doda enega po enega v *OutputIterator*, temveč doda samo "dummy"točko in zaključi. Če najde samo eno točko (ko pride do lista Kd drevesa), potem tako točko tudi vrne. To si lahko privoščimo, ker nas v notranjih vozliščih drevesa najbližjega soseda ne zanimajo konkretne točke, ampak samo informacija, ali se vsaj ena točka nahaja v območju iskanja. Šele ko pridemo do lista NN drevesa, nas zanima konkretna točka. S tem, ko je vedno vrnjena ena točka, se časovna kompleksnost spremeni v $O(\log n)$.

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

```
21          *it = Point_d(0, 0); ++it;
22          //it = node->lower()->tree_items(it);
23       }
24       else
25           if (q.inner_range_intersects(b))
26             it = node->lower()->search_exists(it, q, b);
27
28       if (q.outer_range_contains(b_upper)) {
29          *it = Point_d(0,0); ++it;
30          //it = node->upper()->tree_items(it);
31       }
32       else
33          if (q.inner_range_intersects(b_upper))
34             it = node->upper()->search_exists(it, q, b_upper);
35    };
36    return it;
37 }
```

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

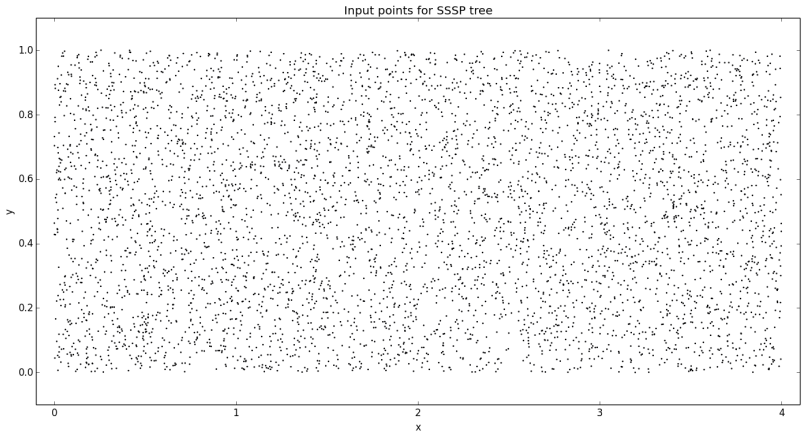
```
28      }
29      else {
30          Point_2 first = result.front();
31          return tuple<bool, Point_2> {true, first};
32      }
33  }
```

Poglavje 5

Rezultati

5.1 Drevo najkrajših poti

Algoritem za izgradnjo drevesa smo pognali na različnem številu vhodnih točk (glej tabelo). V prvem primeru so točke naključno generirane znotraj pravokotnika dimenzij 4x1. S tem



Slika 5.1: Točke v 4x1 pravokotniku.

Poglavje 6

Sklepne ugotovitve

Literatura

[1] CGAL 4.6 - 2D Voronoi Diagram Adaptor: User Manual. Dostopno na: http://doc.cgal.org/latest/Voronoi_diagram_2/index.html