Odabrane teme računalne 2D geometrije

Sabolić, Fran

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Fran Sabolić

ODABRANE TEME RAČUNALNE 2D GEOMETRIJE

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fran Sabolić

Matični broj: 0016141404

Studij: Informacijski sustavi

ODABRANE TEME RAČUNALNE 2D GEOMETRIJE

ZAVRŠNI RAD

Mentor :

Doc. dr. sc. Marcel Maretić

Varaždin, rujan 2022.

Fran Sabolić

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Računalna geometrija bavi se proučavanjem geometrijskih problema i primjenom matematičkih i algoritamskih metoda za njihovo rješavanje. Iako je za neke probleme jednostavno osmisliti algoritam za rješavanje, potrebno je osmisliti algoritme koji mogu pouzdano i efikasno riješiti zadane probleme u što kraćem vremenu. Cilj rada je proći kroz osnovne probleme računalne geometrije i načine njihovog rješavanja. Zbog jednostavnosti, svi problemi su definirani u dvije dimenzije. Svaki problem u radu se prvo definira nakon čega se prikazuje nekoliko algoritama za njegovo rješavanje. Za svaki problem je implementiran barem jedan od opisanih algoritama u programskom jeziku Python, a svaka implementacija ima uključenu i vizualizaciju rješenja pomoću biblioteke Matplotlib. Obrađeni problemi su: presjek segmenata, problem točke u poligonu, određivanje konveksne ljuske, Booleove operacije nad poligonima, problem triangulacije skupa točaka i izrada Voronojevog dijagrama.

Ključne riječi: konveksna ljuska, poligoni, dvodimenzionalni prostor, algoritmi, računalna geometrija

Sadržaj

1. Uvod

Računalna geometrija je disciplina koja se bavi efikasnim rješavanjem problema geometrije pomoću računala. Kako računala "ne vide" geometrijske objekte, potrebno je osmisliti algoritme i računalne objekte koji ih mogu predstaviti na računalu razumljiv način. Osim što računalo mora moći raditi sa geometrijskim podacima, mora ih efikasno koristiti i obrađivati. Implementacije tih algoritama danas omogućavaju brzo, praktično i lako snalaženje u svijetu. Tako na primjer, svaki put kada u novom gradu pokušavamo pronaći put do restorana ili kafića, računalo mora odrediti našu poziciju u prostoru i kako najbrže doći do željene lokacije koristeći pri tom algoritme koji su proizašli iz računalne geometrije. Očito, brzina izvršavanja i preciznost tih algoritama određuje naše zadovoljstvo s aplikacijom koju koristimo stoga nije dovoljno napraviti algoritam po principu "bitno da radi". Zato se računalna geometrija bavi pronalaženjem najoptimalnijih algoritama za određene probleme. Osim brzine izvođenja, potrebno je paziti na preciznost pojedinih algoritama. To znači da je uz nisku složenost potrebno uzeti u obzir koliko će računalo izgubiti na preciznosti izvodeći određene korake algoritma. Iz tih razloga se često izbjegavaju trigonometrijske funkcije i dijeljenje bez obzira što je trigonometrija vrlo koristan alat za rješavanje raznih problema, a dijeljenje je osnovna operacija nad brojevima. Nekada je teško pogoditi dobar omjer između brzine i preciznosti stoga uvijek treba uzeti u obzir koliko je određeni algoritam praktičan unatoč određenim manama. Neki osnovni problemi koji mogu dati uvid u način razmišljanja računalne geometrije su određivanje konveksne ljuske, izrada Voronojevog dijagrama ili možda jednostavnije traženje sjecišta dužina. Svaki od ovih problema ima barem nekoliko rješenja u obliku algoritama s različitim složenostima, preciznošću i težinom implementacije. Nekada će odabir algoritma za praktičnu primjenu biti očit, kao na primjer ako imamo naivni algoritam čija će brzina izvođenja eksponencijalno rasti s porastom unosa i sofisticiran algoritam koji razdvaja glavni problem na više manjih kako bi ih prije riješio u manje vremena.

Danas je računalna geometrija omogućila razvoj računalnog vida, robotike, geografskih informacijskih sustava i mnogih drugih područja čiji je temelj geometrijska obrada podataka. Zbog razvoja računalne geometrije danas postoje autonomni automobili, automatizirana skladišta, 3D printanje i mnoge tehnologije koje su nekada izgledale nemoguće. Obrađeni algoritmi i problemi u radu imaju za svrhu prikazati način razmišljanja unutar računalne geometrije kao uvod u računalnu 2D geometriju. Određeni algoritmi su implementirani u programskom jeziku Python, a vizualizacije su napravljene koristeći biblioteku Matplotlib. Sve implementacije se nalaze na GitHubu na linku: `https://bit.ly/3cRe1Qe`. Algoritmi su opisani i dodatno prikazani slikama izrađenim u GeoGebri, a sve slike u radu je izradio autor završnog rada.

2. Računalna geometrija

Geometrija kao područje matematike postoji od antike, a odatle i dolazi naziv „geometrija“ što znači „mjerenje zemlje“. Glavni zadatak geometrije je proučavanje i opisivanje svojstava i odnosa objekata koji se nalaze u n-dimenzionalnom prostoru. S razvojem matematike, razvijale su se i metode za rješavanje geometrijskih problema. U počecima, oni su se rješavali vizualno na intuitivnoj razini, no formalizacijom matematičkih metoda, dolazi se do algoritama za njihovo rješavanje. U posljednjih nekoliko desetljeća, dolazi do potrebe za računalnom obradom geometrijskih objekata kao što su točke, dužine, poligoni i slično. Kao odgovor na tu potrebu, 70-ih godina se javlja računalna geometrija. Računalna geometrija je polje računarstva koje proučava, opisuje i razvija algoritme za efikasno rješavanje geometrijskih problema koji kao ulaz primaju i kao rezultat vraćaju geometrijske podatke [1].

Računalna geometrija ima nekoliko podjela, no glavne dvije su kombinatorna računalna geometrija i numerička računalna geometrija.

Kombinatorna računalna geometrija se još naziva i algoritamskom geometrijom. Ovdje je naglasak na diskretnim geometrijskim strukturama [1]. Diskretne strukture su objekti koji poprimaju različite vrijednosti ili im se može dodijeliti različita vrijednost, najčešće prirodni broj. Ovo područje računalne geometrije se naziva kombinatornom računalnom geometrijom zato što je kombinatorika područje matematike koje se bavi odnosima između diskretnih struktura [2]. Svi obrađeni problemi (presjek segmenata, problem točke u poligonu, određivanje konveksne ljuske, Booleove operacije nad poligonima, problem triangulacije skupa točaka, Voronojev dijagram) spadaju u kombinatornu računalnu geometriju, a čine samo mali dio svih problema koji se u njoj rješavaju.

Numerička računalna geometrija odnosi se na prikazivanje objekata iz stvarnog svijeta u oblicima pogodnim za CAD sustave. Za razliku od kombinatorne računalne geometrije koja se bavi diskretnim strukturama i većinom radi sa poligonima i politopima, numerička računalna geometrija je usmjerena na prikaz krivulja i površina. U njoj je glavni naglasak na razvoju numerički stabilnih algoritama koji će proizvesti rezultate sa što manjom pogreškom [3].

Neke prednosti razvoja računalne geometrije su razvoj geometrijskih alata koji omogućavaju efikasnije i lakše rješavanje određenih problema. Prije sistematskog proučavanja postojećih algoritama, mnogi od njih su davali kriva rješenja za određene rubne slučajeve ili su jednostavno bili neefikasni. Druga prednost je što je razvoj računalne geometrije povezan s razvojem diskretne kombinatorne geometrije pa su napredci u jednom području ujedno i napredci u drugom. Na kraju, računalna geometrija stavlja naglasak na dokazivanje složenosti pojedinog algoritma, ali i na numeričkoj stabilnosti tog algoritma. Tako mogu nastati vrlo pouzdani algoritmi sa visokom efikasnošću, a ponekada mogu nastati „savršeni“ algoritmi čija složenost ne može biti manja za dani problem [4].

Iako je računalna geometrija vrlo raznolika i korisna disciplina, ipak ima određena ograničenja. Kako se računalna geometrija najviše fokusira na rješavanje problema koristeći diskretnu geometriju, teško je riješiti probleme koji nisu diskretni. Osim toga, računalna geometrija se najviše bazira na rješavanju problema koji se odnose na „plosnate“ objekte jer je za njih

mnogo lakše odrediti sve slučajeve korištenja pa tako i složenost samog algoritma za rješavanje problema. Na kraju, u računalnoj geometriji se dimenzija promatranog prostora često uzima kao neki manji prirodni broj kako bi se uopće mogli konstruirati algoritmi koji rješavaju dani problem. Što je dimenzija prostora veća, to je teže naći algoritme koji će rješavati probleme u tom prostoru. Iz tog razloga, velika većina algoritama koji su proizašli iz računalne geometrije rješava probleme u dvodimenzionalnom ili trodimenzionalnom prostoru [4].

Računalna geometrija ima mnoge primjene unutar različitih polja računarstva. Tako na primjer u računalnoj grafici na temelju presjeka kursora miša i određene površine na monitoru, računalo može odrediti na koju točku monitora je korisnik pritisnuo mišem, u robotici roboti mogu pronaći određeni put do nekog objekta i pritom izbjegavati razne prepreke, u geografskim informacijskim sustavima možemo pronaći presjeke različitih vrsta geografskih karata kako bi dobili potpunu sliku nekog područja, u CAD-u se određeni objekti mogu simulirati kao presjeci poligona i tako dalje. Samo iz ovih par navedenih primjera, lako je za zaključiti koliko značenje računalna grafika ima u svakodnevnom životu, a da nismo ni svjesni [5].

3. Problem presjeka skupa dužina

Za dani skup dužina S koji se sastoji od n dužina $\overline{D_1}, \overline{D_2}, ..., \overline{D_n}$ u ravnini, žele se pronaći sva njihova sjecišta. Pretpostavlja se da su sve dužine zadane kao parovi dviju različitih točaka koje nazivamo krajnjim točkama A i B [6].

3.1. Određivanje sjecišta dviju dužina

Prije rješavanja problema presjeka skupa dužina, potrebno je osmisliti način otkrivanja sjecišta između samo dviju dužina. Nakon toga, vrlo lako se može doći barem do naivnog algoritma za rješavanje problema presjeka skupa dužina.

Za razliku od traženja sjecišta između pravaca, traženje sjecišta dužina ima nekoliko dodatnih rubnih slučajeva. Kako su pravci beskonačne duljine, jedini rubni slučaj kada se ne sijeku je kada su paralelni. Ako nisu, tada se samo traži rješenje sustava dviju jednadžbi pravaca. Dvije dužine imaju beskonačno mnogo pozicija u kojima se ne sijeku stoga nije dovoljno samo provjeriti jesu li paralelne. Za dužine ne postoji eksplicitna jednadžba, no može se izvesti parametarska jednadžba dužine preko vektorske jednadžbe pravca. Pogledajmo prvo vektorsku jednadžbu pravca:

$$\vec{p} = \vec{p_0} + t \cdot \vec{v}.$$

$\vec{p_0}$ je pozicijski vektor neke točke na pravcu, a $\vec{v}$ je vektor smjera. Ako su zadane dvije točke nekog pravca, T_1 i T_2, tada već imamo pozicijski vektor jedne točke, a oduzimanjem pozicijskih vektora koje te dvije točke predstavljaju možemo dobiti i vektor smjera pravca. To znači da $\vec{v}$ možemo prikazati kao razliku $\vec{p}$ i $\vec{p_0}$. U tom slučaju možemo zamisliti da te dvije točke omeđuju neku dužinu $\overline{D}$ koja se nalazi na pravcu. Vektorska jednadžba pravca će vraćati točke koje se nalaze na toj dužinu za svaki t gdje je $0 \leq t \leq 1$. Ako je $t = 0$ ili $t = 1$ tada dobivamo krajnje točke T_1 i T_2 dužine $\overline{D}$. Stoga parametarska jednadžba dužine glasi:

$$\vec{p_i} = \vec{p_0} + t \cdot (\vec{p} - \vec{p_0}), \ \ 0 \leq t \leq 1, \ t \in \mathbb{R}$$

gdje je $\vec{p_i}$ pozicijski vektor neke točke na dužini, a $\vec{p_0}$ i $\vec{p}$ su pozicijski vektori krajnjih točaka dužine. Pretpostavimo da imamo dvije dužine $\overline{P_1P_2}$ i $\overline{P_3P_4}$. Jednadžbe tih dužina tada glase:

$$\vec{p_i} = \vec{p_1} + t \cdot (\vec{p_2} - \vec{p_1}), \ \ 0 \leq t \leq 1, \ t \in \mathbb{R}$$
$$\vec{p_i} = \vec{p_3} + s \cdot (\vec{p_4} - \vec{p_3}), \ \ 0 \leq s \leq 1, \ s \in \mathbb{R}$$

gdje su $\vec{p_1}$, $\vec{p_2}$, $\vec{p_3}$ i $\vec{p_4}$ pozicijski vektori predstavljeni točkama P_1, P_2, P_3 i P_4 . Nakon toga, kao i sa pravcima, pokušavamo naći rješenje sustava dviju jednadžbi dužina gdje su nepoznanice parametri t i s. Traženje sjecišta dužina počinjemo s pretpostavkom da postoji neka točka $\vec{p_i}$ koja pripada objema dužinama. Odnosno,

$$\vec{p_1} + t \cdot (\vec{p_2} - \vec{p_1}) = \vec{p_3} + s \cdot (\vec{p_4} - \vec{p_3}) \, .$$

Ako je $0 \leq t \leq 1$ i $0 \leq s \leq 1$ tada znamo da se te dvije dužine sijeku (odnosno imaju zajedničku točku). Prvo moramo izraziti t i s:

$$s = \frac{(\vec{p_1} - \vec{p_3}) + t \cdot (\vec{p_2} - \vec{p_1})}{\vec{p_4} - \vec{p_3}}$$

$$t = \frac{(\vec{p_3} - \vec{p_1}) + s \cdot (\vec{p_4} - \vec{p_3})}{\vec{p_2} - \vec{p_1}}$$

Kako bi dobili koeficijent s, prvo moramo prikazati koeficijent t preko koordinata krajnjih točaka. Izjednačavanjem tih dviju jednadžbi, dolazimo do jednadžbe za koeficijent s:

$$t = \frac{(x_3 - x_1) + s \cdot (x_4 - x_3)}{x_2 - x_1}$$

$$t = \frac{(y_3 - y_1) + s \cdot (y_4 - y_3)}{y_2 - y_1}$$

$$\frac{(x_3 - x_1) + s \cdot (x_4 - x_3)}{x_2 - x_1} = \frac{(y_3 - y_1) + s \cdot (y_4 - y_3)}{y_2 - y_1} \; / \; \cdot (x_2 - x_1)(y_2 - y_1)$$

$$(x_3 - x_1)(y_2 - y_1) + s \cdot (x_4 - x_3)(y_2 - y_1) = (y_3 - y_1)(x_2 - x_1) + s \cdot (y_4 - y_3)(x_2 - x_1)$$

$$s \cdot (x_4 - x_3)(y_2 - y_1) - s \cdot (y_4 - y_3)(x_2 - x_1) = (y_3 - y_1)(x_2 - x_1) - (x_3 - x_1)(y_2 - y_1)$$

$$s \cdot [(x_4 - x_3)(y_2 - y_1) - (y_4 - y_3)(x_2 - x_1)] = x_2(y_3 - y_1) - x_1(y_3 - y_1) - x_3(y_2 - y_1) + x_1(y_2 - y_1)$$

$$s \cdot [(x_4 - x_3)(y_2 - y_1) - (y_4 - y_3)(x_2 - x_1)] = x_2y_3 - x_2y_1 - x_1y_3 + x_1y_1 - x_3y_2 - x_3y_1 + x_1y_2 - x_1y_1$$

$$s \cdot [(x_4 - x_3)(y_2 - y_1) - (y_4 - y_3)(x_2 - x_1)] = x_2y_3 - x_2y_1 - x_1y_3 - x_3y_2 - x_3y_1 + x_1y_2 \mathbf{- x_1y_1 + x_1y_1}$$

$$s \cdot [(x_4 - x_3)(y_2 - y_1) - (y_4 - y_3)(x_2 - x_1)] = x_1(y_2 - y_3) + x_2(y_3 - y_1) - x_3(y_2 - y_1)$$
$$/ : (x_4 - x_3)(y_2 - y_1) - (y_4 - y_3)(x_2 - x_1)$$

$$s = \frac{x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)}{(x_4 - x_3)(y_2 - y_1) - (y_4 - y_3)(x_2 - x_1)} \; / \; \cdot \frac{-1}{-1}$$

$$s = \frac{x_1(y_3 - y_2) + x_2(y_1 - y_3) + x_3(y_2 - y_1)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \, .$$

Na sličan način dolazimo do koeficijenta t:

$$s \; = \; \frac{(x_1 - x_3) + t \cdot (x_2 - x_1)}{x_4 - x_3}$$

$$s \; = \; \frac{(y_1 - y_3) + t \cdot (y_2 - y_1)}{y_4 - y_3}$$

$$\frac{(x_1 - x_3) + t \cdot (x_2 - x_1)}{x_4 - x_3} \; = \; \frac{(y_1 - y_3) + t \cdot (y_2 - y_1)}{y_4 - y_3} \; / \; \cdot (x_4 - x_3)(y_4 - y_3)$$

$$(x_1 - x_3)(y_4 - y_3) + t \cdot (x_2 - x_1)(y_4 - y_3) = (y_1 - y_3)(x_4 - x_3) + t \cdot (y_2 - y_1)(x_4 - x_3)$$

$$t \cdot (x_2 - x_1)(y_4 - y_3) - s \cdot (y_2 - y_1)(x_4 - x_3) = (y_1 - y_3)(x_4 - x_3) - (x_1 - x_3)(y_4 - y_3)$$

$$t \cdot [(x_2 - x_1)(y_4 - y_3) - (y_2 - y_1)(x_4 - x_3)] = x_4(y_1 - y_3) - x_3(y_1 - y_3) - x_1(y_4 - y_3) + x_3(y_4 - y_3)$$

$$t \cdot [(x_2 - x_1)(y_4 - y_3) - (y_2 - y_1)(x_4 - x_3)] = x_1(y_3 - y_4) + x_4(y_1 - y_3) - x_3y_1 + x_3y_4 \boldsymbol{- x_3y_3 + x_3y_3}$$

$$t \cdot [(x_2 - x_1)(y_4 - y_3) - (y_2 - y_1)(x_4 - x_3)] = x_1(y_3 - y_4) + x_4(y_1 - y_3) + x_3(y_4 - y_1)$$
$$/ : (x_2 - x_1)(y_4 - y_3) - (y_2 - y_1)(x_4 - x_3)$$

$$t = \frac{x_1(y_3 - y_4) + x_4(y_1 - y_3) + x_3(y_4 - y_1)}{(x_2 - x_1)(y_4 - y_3) - (y_2 - y_1)(x_4 - x_3)} \; / \; \cdot \frac{-1}{-1}$$

$$t = \frac{x_1(y_3 - y_4) + x_4(y_1 - y_3) + x_3(y_4 - y_1)}{(y_2 - y_1)(x_4 - x_3) - (x_2 - x_1)(y_4 - y_3)} \, .$$

Uvrštavanjem vrijednosti točaka P_1, P_2, P_3 i P_4 u jednadžbe koeficijenata dobiti ćemo

neke realne brojeve. Kao što je ranije rečeno, ako su ti realni brojevi između 0 ili 1 tada postoji sjecište dužina, odnosno ako su $0 \leq t \leq 1$ i $0 \leq s \leq 1$ tada znamo da se dvije dužine sijeku. Ukoliko želimo saznati koordinate sjecišta, potrebno je samo uvrstiti jedan od koeficijenata u odgovarajuću jednadžbu [7].

Implementacija traženja sjecišta

Implementacija traženja sjecišta je samo primjena ranije izvedenih formula. Metoda `sjeciste` je metoda klase `Duzina`. Parametri `self` i `duzina` su objekti klase `Duzina`, a njihovo sjecište tražimo. Prije samog traženja sjecišta, potrebno je provjeriti jesu li dvije dužine paralelne. Ukoliko jesu, potrebno je provjeriti preklapaju li se te dvije dužine ili ne. Ako se ne preklapaju, tada sjecište ne postoji. Ako nisu paralelne, vrijednosti krajnjih točaka danih dužina se unose u formule koeficijenata i ako postoji sjecište, ono se računa uvrštavanjem vrijednosti koeficijenta t u jednadžbu dužine te se dobiva neka točka (sjecište).

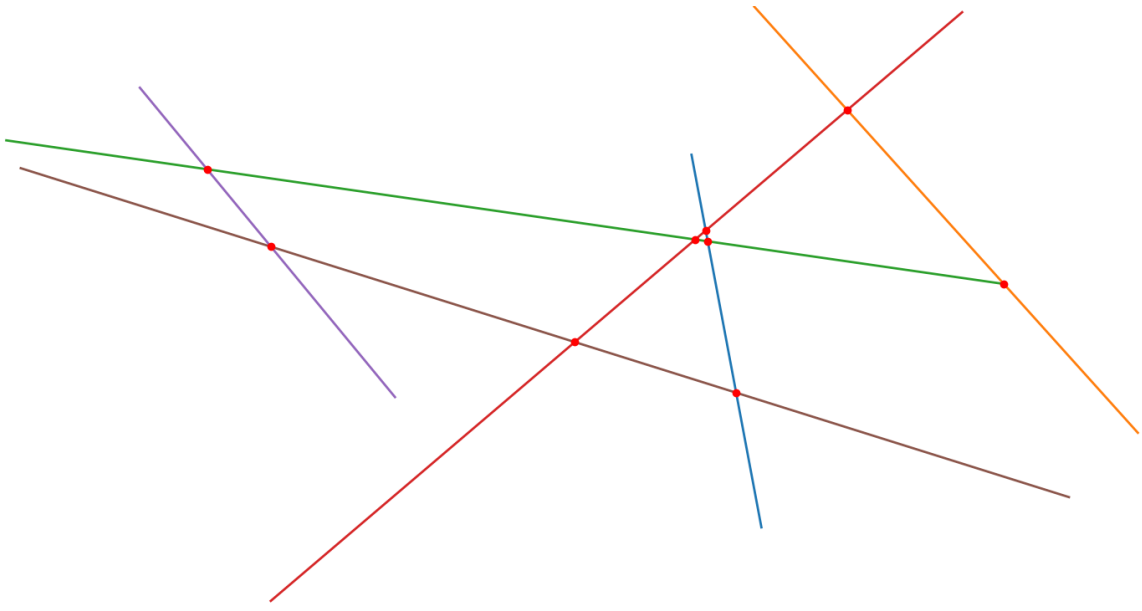
```
def sjeciste(self, duzina):
    vektor_self = self.u_vektor()
    vektor_other = duzina.u_vektor()
    if vektor_self // vektor_other:
        if duzina.A.pripada_duzini(self):
            return duzina.A
        elif duzina.B.pripada_duzini(self):
            return duzina.B
        elif self.A.pripada_duzini(duzina):
            return self.A
        elif self.B.pripada_duzini(duzina):
            return self.B
        else:
            return Tocka(None, None)

    t = ((self.A.x * (duzina.B.y - duzina.A.y)
          + duzina.A.x * (self.A.y - duzina.B.y)
          + duzina.B.x * (duzina.A.y - self.A.y))
         / (vektor_other.vektorski_produkt(vektor_self)))

    s = ((self.A.x * (duzina.A.y - self.B.y)
          + self.B.x * (self.A.y - duzina.A.y)
          + duzina.A.x * (self.B.y - self.A.y))
         / (vektor_self.vektorski_produkt(vektor_other)))

    if 0 <= t <= 1 and 0 <= s <= 1:
        return self.A + (self.B - self.A).mnozenje_skalarom(t)
```

```
        return Tocka(None, None)
```

3.2. *Brute Force* algoritam

Brute Force algoritam za traženje sjecišta skupa dužina je najjednostavniji za implementirati. Potrebno je usporediti svaku dužinu u skupu sa svakom drugom dužinom u skupu i probati pronaći sjecište tih dužina. Ukratko, korištenjem ugniježđene for petlje, prolazimo kroz sve dužine i svaki put kada naiđemo na sjecište, pohranjujemo ga u listu sjecišta. Očito, kako bi usporedili sve dužine potrebno je $O(n^2)$ vremena jer kroz listu dužina moramo proći jednom za svaku od dužina. Zbog velike složenosti, *Brute Force* algoritam nema previše praktičnih primjena. Postoje mnogo optimalniji algoritmi od kojih je jedan *Sweep Line* algoritam.

```
def presjek_segmenata(duzine):
    sjecista = []
    br_duzina = len(duzine)
    for i in range(br_duzina):
        for j in range(i + 1, br_duzina):
            sjec = duzine[i].sjeciste(duzine[j])
            if not sjec.prazna():
                sjecista.append(sjec)
    return sjecista
```

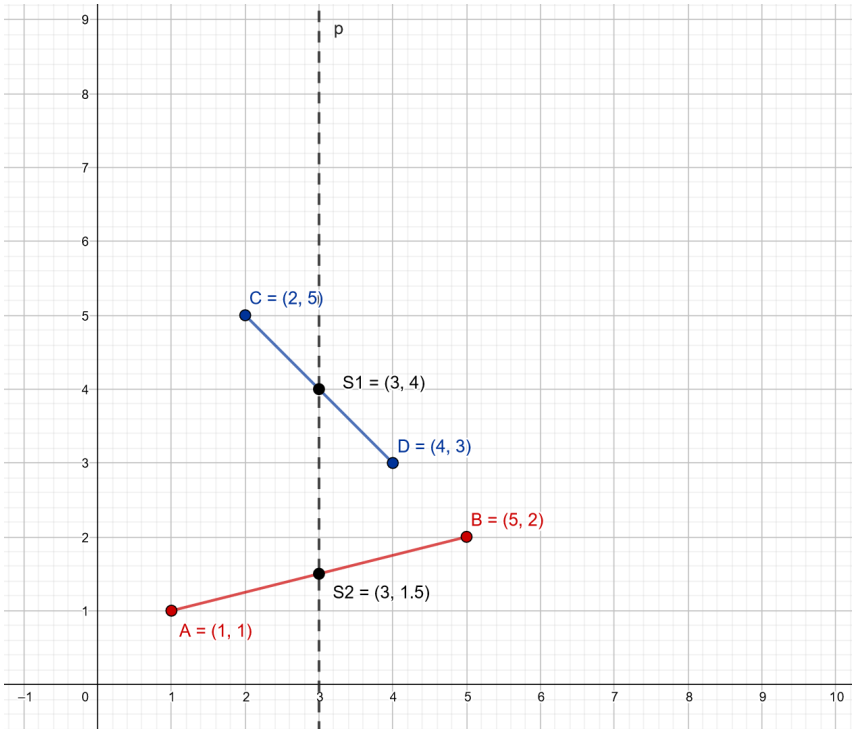


Slika 1: Prikaz implemetacije presjeka skupa dužina

3.3. *Sweep Line* algoritam

Sweep Line algoritam se temelji na poopćenju algoritma za traženje preklapanja intervala realnih brojeva u jednoj dimenziji. Taj algoritam radi sa skupom intervala realnih brojeva $X = \{S_1, S_2, \ldots, S_n\}$ gdje je svaki interval ograničen sa svojim najmanjim brojem, $S_{n_{\text{lijevo}}}$, i najvećim brojem $S_{n_{\text{desno}}}$. Ukratko, problem u jednoj dimenziji se rješava tako da se uzlazno sortiraju sve krajnje vrijednosti intervala iz skupa X. Nakon toga, iterira se kroz sortirani skup, a u zasebnu strukturu podataka se unosi ili briše interval, ovisno je li se naišlo na najmanji ili najveći broj intervala. Ako se kroz iteriranje sortiranog skupa naiđe na najmanji broj $S_{n_{\text{lijevo}}}$ nekog intervala, taj interval se pohranjuje u strukturu. Ukoliko se naiđe na najveći broj $S_{n_{\text{desno}}}$, interval se izbacuje iz strukture. Znamo da je došlo do preklapanja skupova ako pokušamo pohraniti neki interval u strukturu, a ona nije prazna. Očito, ovo rješenje zahtjeva nekoliko preinaka prije nego što se može primijeniti na dvije dimenzije [6].

U dvije dimenzije, cilj algoritma je smanjiti broj potencijalnih dužina za usporedbu kako bi pronašli sva sjecišta. Za početak, potrebno je definirati pojam usporedivosti dviju dužina u ravnini. Za dvije dužine $\overline{D_1}$ i $\overline{D_2}$ kažemo da su usporedive ako postoji pravac p paralelan s y-osi koji ih sječe po istoj apscisi x. Ovisno o udaljenosti sjecišta S_1 i S_2 i pravca p, kažemo da je dužina $\overline{D_1}$ „iznad“ dužine $\overline{D_2}$ ako sjecište S_1 ima veću ordinatu nego sjecište S_2.



Slika 2: Prikaz dvije dužine koje ispunjavaju uvjet usporedivosti za *Sweep Line* algoritam. U ovom slučaju, dužina $\overline{CD}$ je iznad dužine $\overline{AB}$

Ova usporedivost je razlog zbog kojeg se algoritam zove *Sweep Line* (linija „pročešljavanja“). Naime, kako bi smanjili moguće kandidate za dužine koje se sijeku, potrebno je „premesti“ sve dužine iz nekog smjera (u ovom slučaju s lijeva na desno) pomoću pravca p i provjeriti ispunjavaju li one uvjet usporedivosti jedna s drugom. Ako ne, tada ne moramo tražiti

njihovo sjecište [6].

Za izvršavanje algoritma potrebne su dvije strukture podataka, jedna u koju ćemo spremati dužine koje potencijalno imaju sjecište i druga u koju ćemo pohranjivati točke događaja po kojima će se kretati pravac p. Nazovimo prvu strukturu LS (*Line Segments*), a drugu EP (*Event Points*). Sve dužine unutar strukture LS su sortirane prema njihovom položaju u odnosu sa ostalim dužinama u strukturi. To znači da u strukturi LS dužina može biti „ispod“ ili „iznad“ neke druge dužine prema pravilu usporedivosti. Druga struktura EP sastoji se od točaka događaja koje su sortirane uzlazno prema apscisama. Točka događaja je točka u kojoj izvršavamo određene operacije nad skupom dužina, ovisno o vrsti točke događaja. Slično kao u jednoj dimenziji u kojoj postoje dvije točke događaja ($S_{n_{\text{lijevo}}}$ i $S_{n_{\text{desno}}}$), u *Sweep Line* algoritmu postoje tri vrste točaka događaja: lijeva, desna i sjecište. Lijeva točka događaja je krajnja točka neke dužine čija apscisa je manja od druge krajnje točke (ona koja je više “lijevo“ u koordinatnom sustavu). Druga krajnja točka te dužine je tada desna točka događaja. Treća točka događaja, sjecište, je svaka točka u kojoj se sijeku dvije dužine. Svaki put kada se u strukturu EP unese nova točka događaja, mora biti dodana tako da točke u strukturi ostanu sortirane.

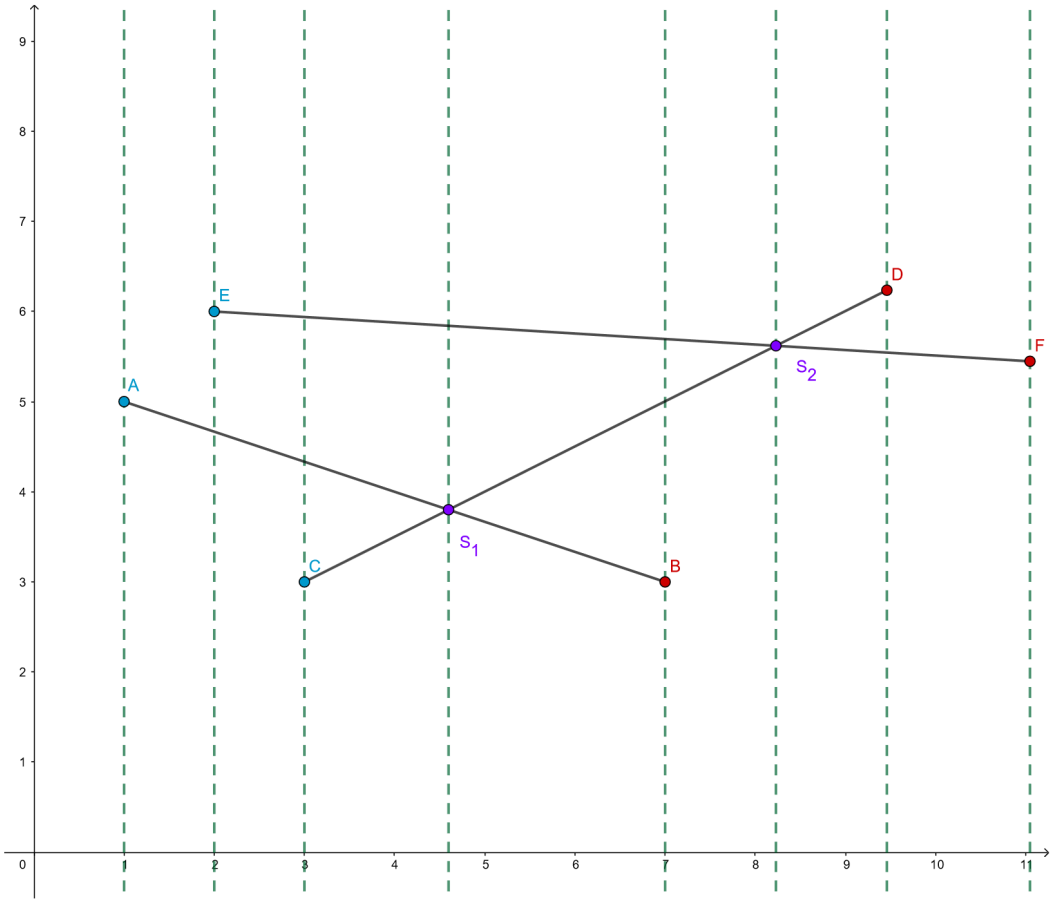
Svaka točka događaja ima skup operacija koje se moraju izvršiti kada se na nju naiđe:

Lijeva - u strukturu LS pohranjujemo dužinu $\overline{D}$ kojoj pripada ta lijeva točka. Nakon toga, tražimo presjek unesene dužine $\overline{D}$ i njenih dviju susjednih dužina: $\overline{D_{\text{iz}}}$, one „iznad“ unesene dužine, i $\overline{D_{\text{is}}}$, one „ispod“ unesene dužine. Ako je pronađeno sjecište ili sjecišta, ona se pohranjuju u strukturu EP na odgovarajuće mjesto, ovisno o apscisama sjecišta.

Desna - Prvo je potrebno pronaći dužinu $\overline{D}$ kojoj desna točka pripada. Nakon toga, pokušavamo pronaći sjecište dužine $\overline{D_{\text{iz}}}$ (koja je odmah iznad dužine $\overline{D}$) i dužine $\overline{D_{\text{is}}}$ (koja je odmah ispod dužine $\overline{D}$). Ako to sjecište postoji, pohranjuje se u strukturu EP. Na kraju, potrebno je izbaciti dužinu $\overline{D}$ iz strukture LS.

Sjecište - Nailaskom na sjecište S, potrebno je pronaći dužine $\overline{D_1}$ i $\overline{D_2}$ čije sjecište je S te dužinu $\overline{D_{1_{\text{iz}}}}$ i dužinu $\overline{D_{2_{\text{is}}}}$. Dužina $\overline{D_{1_{\text{iz}}}}$ je prva dužina iznad dužine $\overline{D_1}$, a dužina $\overline{D_{2_{\text{is}}}}$ je prva dužina ispod dužine $\overline{D_2}$. Dužina $\overline{D_1}$ mora biti iznad dužine $\overline{D_2}$. Zatim dužinama $\overline{D_1}$ i $\overline{D_2}$ zamijenimo mjesta u strukturi LS jer su nakon sjecišta njihovi položaji zamijenjeni ($\overline{D_1}$ je sada ispod $\overline{D_2}$). Na kraju, potrebno je probati pronaći sjecište dužine $\overline{D_1}$ i $\overline{D_{2_{\text{is}}}}$ te dužine $\overline{D_2}$ i $\overline{D_{1_{\text{iz}}}}$. Ako su pronađena sjecišta, pohranjuju se u strukturu EP i algoritam može nastaviti dalje [6].

Algoritam se izvršava tako da se na početku popunjava struktura EP sa krajnjim točkama svih zadanih dužina. Nakon toga, po redu se iterira kroz svaku točku strukture EP i ovisno o vrsti točke događaja izvršavaju se operacije za tu točku. Algoritam završava kada se prođe kroz sve točke iz strukture EP. Slika 3 prikazuje izvršavanje *Sweep Line* algoritma sa 3 dužine. Svaki redak tablice prikazuje stanja LS i EP struktura nakon izvršenih operacija za određenu točku događaja te pronađena sjecišta u tom koraku.



Trenutna točka	*LS*	*EP*	*Sjecišta*
A	$\overline{AB}$	E, C, B, D, F	$\emptyset$
E	$\overline{EF}, \overline{AB}$	C, B, D, F	$\emptyset$
C	$\overline{EF}, \overline{AB}, \overline{CD}$	S_1, B, D, F	S_1
S_1	$\overline{EF}, \overline{CD}, \overline{AB}$	B, S_2, D, F	S_1, S_2
B	$\overline{EF}, \overline{CD}$	S_2, D, F	S_1, S_2
S_2	$\overline{EF}, \overline{CD}$	D, F	S_1, S_2
D	$\overline{EF}$	F	S_1, S_2
F	$\emptyset$	$\emptyset$	S_1, S_2

Slika 3: Prikaz *Sweep Line* algoritma. Svaka isprekidana zelena linija predstavlja pravac p za određenu točku događaja. Plave točke predstavljaju lijeve točke događaja, ljubičaste sjecišta i crvene desne točke događaja

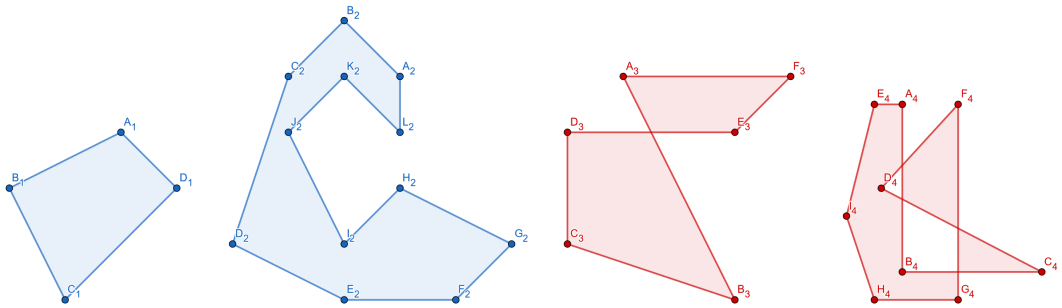
Za razliku od *Brute Force* algoritma, *Sweep Line* algoritam je mnogo brži. Točnije, njegova složenost je $O(n \log n)$ naspram složenosti *Brute Force* algoritma koja je $O(n^2)$. Glavna prednost *Brute Force* algoritma je njegova lakša implementacija, no rijetko će se koristiti za velike skupove dužina [6].

4. Problem točke u poligonu

Za neku točku T i poligon P pokušavamo odrediti nalazi li se točka T unutar poligona P [8].

Poligon definiramo kao područje ravnine zatvoreno rubovima, odnosno dužinama $\overline{T_1T_2}$, ... , $\overline{T_nT_1}$ gdje su točke T_1,T_2,..., T_n vrhovi poligona. Poligoni mogu biti jednostavni ili složeni. Poligon je jednostavan ako ima barem 2 vrha i ako se rubovi sijeku samo sa susjednim rubovima u krajnjim točkama . Važno je napomenuti da će različiti algoritmi za problem točke u poligonu možda davati drugačije rezultate za složene poligone.

Dva najpoznatija algoritma za određivanje pozicije točke u odnosu na poligon su *Ray Casting* algoritam i *Winding Number* algoritam [9].

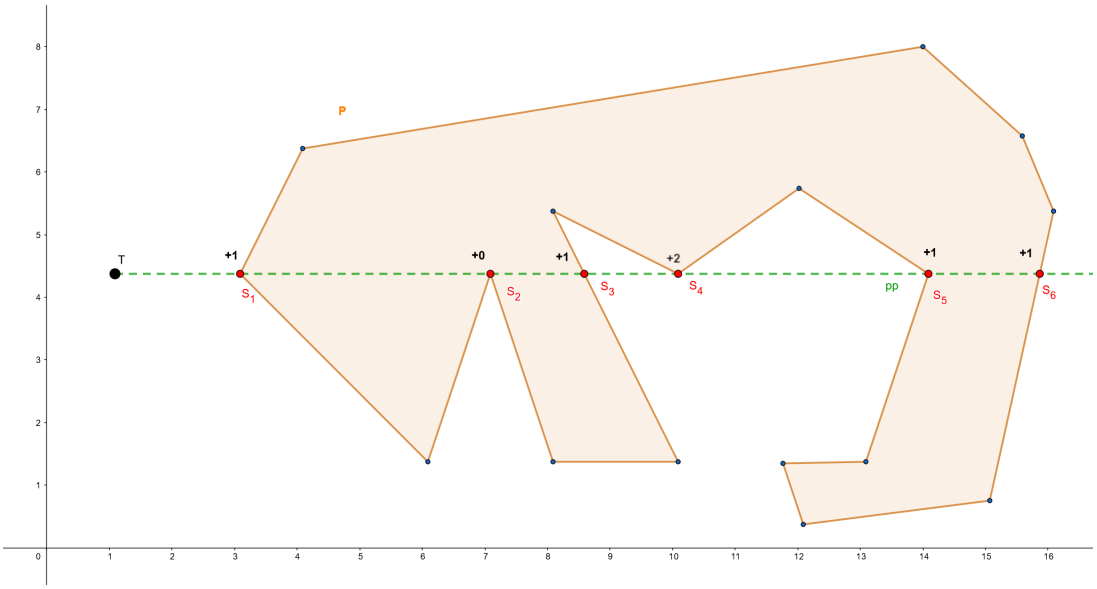


Slika 4: Jednostavni (plavi) i složeni (crveni) poligoni

4.1. *Ray Casting* algoritam

Ray Casting algoritam („algoritam provedene zrake“) se provodi tako da se prvo provuče polupravac p kroz zadanu točku T. Smjer polupravca nije bitan, bitno je samo da je s jedne strane omeđen točkom T. Zatim, računamo koliko puta polupravac p siječe poligon P, odnosno njegove rubove. Ako je taj broj paran, tada se točka nalazi izvan poligona, a ako je neparan, tada je unutar poligona. Ako ne postoji ni jedno sjecište, tada je točka očito izvan poligona. Ukratko, svaki put kada polupravac naiđe na rub poligona ulazimo ili izlazimo iz poligona. Ako pogodimo rub poligona samo jednom, tada smo izašli iz njega. Svaki drugi pogodak je ulazak u poligon, a svaki neparan nakon toga je ponovo izlazak. Rubni slučaj pri izvođenju algoritma je kada se naiđe na neki vrh poligona. Problem pri prolaženju kroz vrh je što se tehnički prolazi kroz dva ruba poligona stoga bi se to moglo brojati kao jedan ili dva „pogotka“. Jedno od rješenja tog problema je da se polupravac rotira oko točke T za neki vrlo malen kut. Nedostaci tog pristupa su što se tada ispočetka moraju tražiti sjecišta sa svim rubovima poligona, ali osim

toga postoji mogućnost da ćemo ponovo pogoditi neki drugi vrh poligona. Drugo rješenje je da se pronađeno sjecište broji kao sjecište samo ako je ordinata sjecišta jednaka ordinati krajnje točke ruba koji se siječe [6].

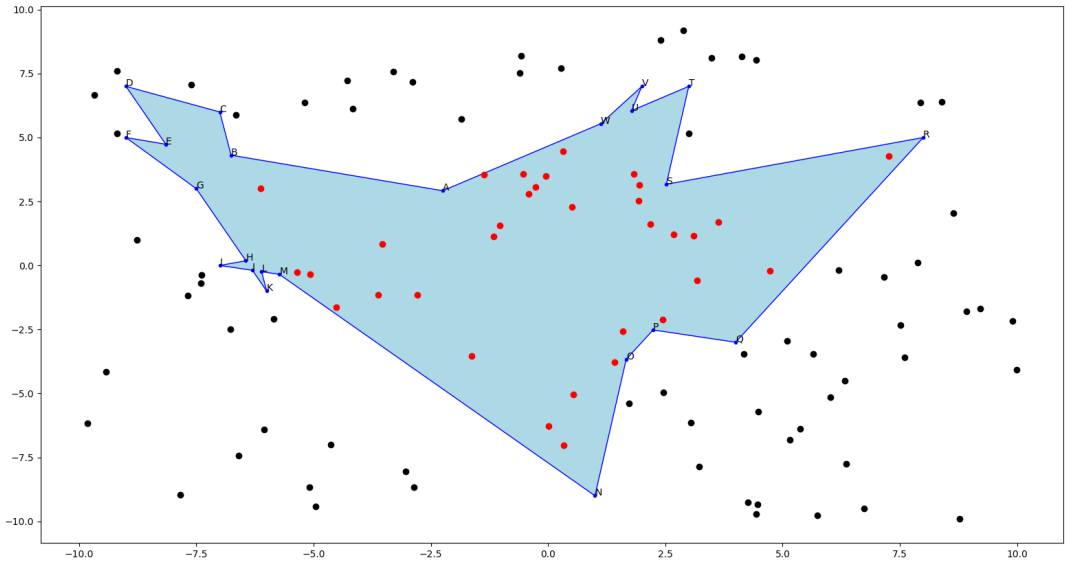


Slika 5: Prikaz *Ray Casting* algoritma za poligon P i točku T

Implementacija *Ray Casting* algoritma

U implementaciji *Ray Casting* algoritma, metoda prima objekt klase `Tocka`, `self`, i objekt klase `Poligon`, `poligon`. Sama metoda pripada klasi `Tocka`. Poligon je predstavljen kao lista točaka koje se mogu spojiti po redu kako bi se dobili rubovi poligona. Umjesto da koristimo polupravac p, stvaramo dužinu čija je jedna krajnja točka `self`, a druga točka je točka s ordinatom točke `self` i apscisom koja je malo veća od najveće apscise u poligonu. Tako smo osigurali da će dužina završavati izvan poligona te nećemo propustiti nijedno sjecište sa rubovima poligona. Osim toga, tada možemo ponovno koristiti funkciju za sjecište dužina i ne moramo pisati klasu za polupravac. Zatim, iteriramo kroz sve rubove poligona. Prvo provjeravamo pripada li točka `self` trenutnom rubu. Ako se točka nalazi na rubu ili je jedna od krajnjih točaka tog ruba, tada će funkcija vratili 0 kako bi označili da se točka tehnički ne nalazi ni unutar ni izvan poligona, već na njegovom rubu. U suprotnom, ukoliko dužina za presjek nije paralelna sa trenutnim rubom, tražimo sjecište između ruba i dužine za presjek. Ako su trenutni rub i dužina paralelni, tada se ili ne sijeku ili se preklapaju. U oba slučaja ne moramo brojati taj slučaj kao sjecište. Ako je pronađeno sjecište, moramo provjeriti je li to sjecište jedan od vrhova poligona. Ako je, provjeravamo je li ordinata sjecišta jednaka ordinati krajnje točke ruba koji se siječe i ako je, povećavamo broj pronađenih sjecišta za jedan. Na kraju, funkcija vraća 1 ili -1, ovisno o tome koliko je sjecišta pronađeno. Broj -1 označava da je točka izvan poligona, a vrijednost 1 označava da je točka unutar poligona [6].

```
def pripada_poligonu(self, poligon):
    najdesnija_tocka_plus = Tocka(poligon.max_x() + 1, self.y)
    if najdesnija_tocka_plus == self:
        return -1

    duzina_za_presjek = Duzina(self, najdesnija_tocka_plus)
    rubovi_poligona = poligon.rubovi()
    sjecista = 0
    for rub in rubovi_poligona:
        if self.pripada_duzini(rub):
            return 0
        if not rub.u_vektor() // duzina_za_presjek.u_vektor():
            sjec = duzina_za_presjek.sjeciste(rub)
            if not sjec.prazna():
                if (sjec in (rub.A, rub.B)
                        and sjec.y == rub.manja_oridnata()
                        or sjec not in (rub.A, rub.B)):
                    sjecista += 1

    return 1 + (sjecista % 2 == 0) * -2
```

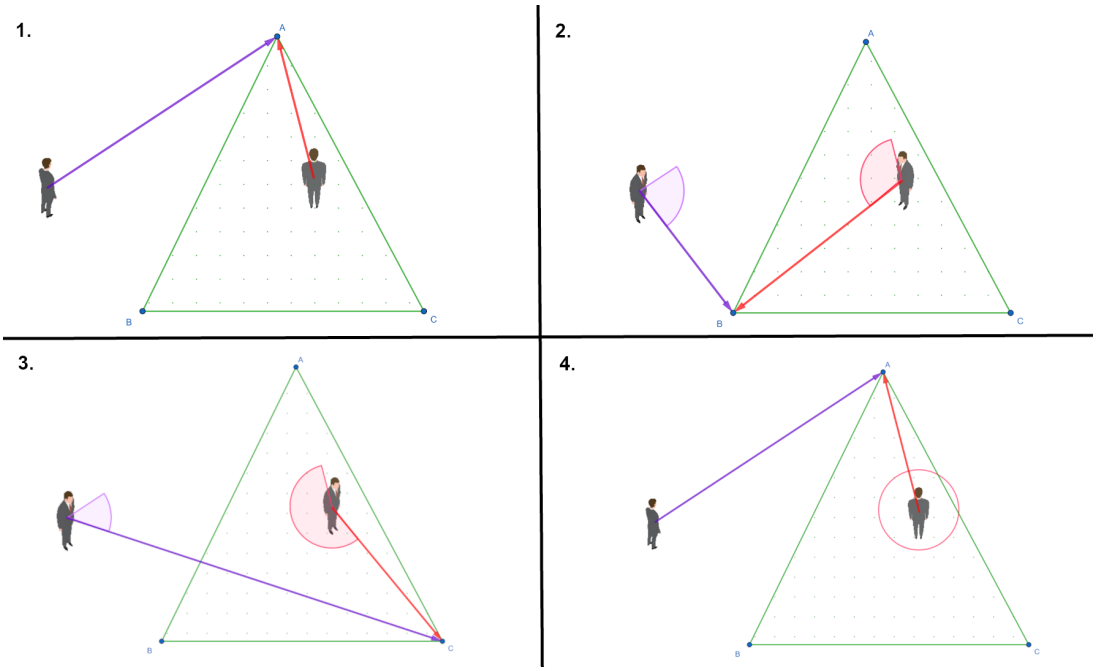


Slika 6: Prikaz implementacije *Ray Casting* algoritma u Pythonu

4.2. *Winding Number* algoritam

Sljedeći algoritam temelji se na broju navojaka. Ukratko, broj navojaka predstavlja koliko puta se točka T okrene oko svoje osi prateći rubove poligona P. Konceptualno to možemo zamisliti kao dvoje ljudi gdje jedna osoba stoji na nekom mjestu i promatra drugu osobu kako

hoda po nekom zamišljenom uzorku na podu. Svaki put kada se promatrač okrene oko svoje osi u određenom smjeru, povećava svoj broj namotaja. Tako je u *Winding Number* algoritmu točka T promatrač, a rubovi poligona su kretanja druge osobe. Broj namotaja možemo izračunati preko zbroja kutova između dužina $\overline{TP_n}$ i $\overline{TP_{n+1}}$ gdje su P_n i P_{n+1} uzastopni vrhovi poligona P. Ako je zbroj kutova 0, tada znamo da se točka nalazi izvan poligona jer se točka nije nijednom okrenula sama oko sebe. U suprotnom, točka je unutar poligona [8].

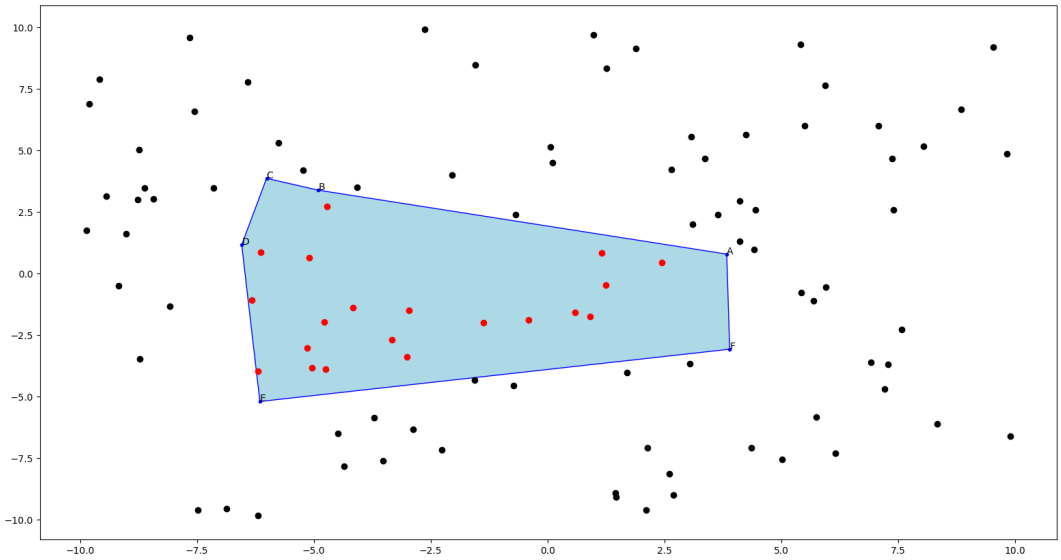


Slika 7: Prikaz *Winding Number* algoritma za dvije točke, jedne unutar poligona (crveno) i jedne izvan poligona (ljubičasto)

Implementacija *Winding Number* algoritma

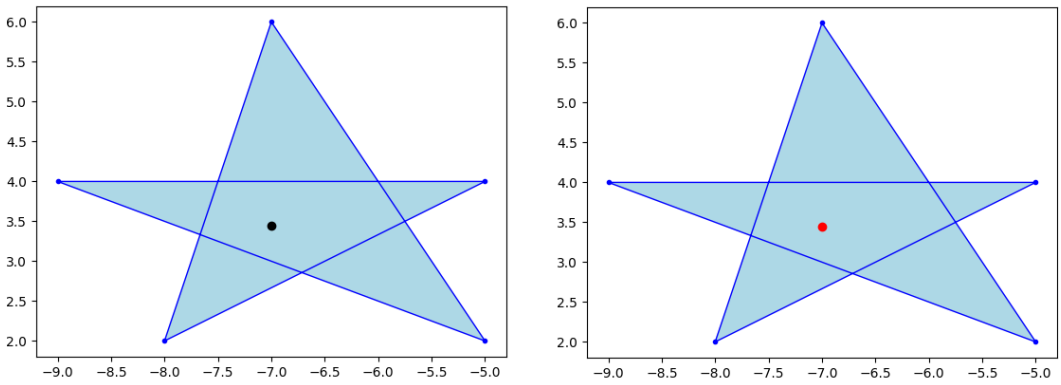
Winding Number algoritam implementiran je kao metoda unutar klase `Tocka`. Njeni parametri su kao i u prijašnoj implementaciji, objekt klase `Tocka`, `self`, i objekt klase `Poligon`, `poligon`. Algoritam kreće tako da iteriramo kroz sve vrhove danog poligona. U svakoj iteraciji prvo provjeravamo pripada li točka za provjeru dužini koju omeđuju trenutni vrh poligona i njegov susjedni vrh. Ukoliko pripada, znamo da se točka za provjeru nalazi na rubu poligona. Dalje, deklariramo tri vektora. Dva vektora, `vektor_tocka_t1` i `vektor_tocka_t2`, od točke za provjeru prema vrhovima koje provjeravamo i jedan vektor, `vektor_tocka_yos`, koji počinje od točke za provjeru, a paralelan je sa y-osi. Svaki vektor predstavljen je kao objek klase `Vektor`. Zatim oduzimamo kutove `vektor_tocka_yos`. Zbog pravilnosti algoritma, moramo postaviti taj kut u raspon od $-\pi$ do π . Tako dobivamo kut s predznakom kojim možemo odrediti za koliko smo se pomaknuli oko točke za provjeru. Dobiveni kut zbrajamo sa svim dobivenim kutovima u petlji, a na kraju provjeravamo je li taj zbroj vrlo blizu 0. Ukoliko je, znamo da se točka nalazi izvan poligona.

```
def pripada_poligonu_wn(self, poligon):
    vrhovi_poligona = poligon.vrhovi
    br_tocaka = poligon.broj_vrhova()

    zbroj_kuteva = 0
    for i in range(0, br_tocaka):
        tocka_n = vrhovi_poligona[i]
        tocka_n_1 = vrhovi_poligona[(i + 1) % br_tocaka]
        rub_poligona = Duzina(tocka_n, tocka_n_1)

        if self.pripada_duzini(rub_poligona):
            return 0

        vektor_tocka_n = Duzina(self, tocka_n).u_vektor()
        vektor_tocka_n_1 = Duzina(self, tocka_n_1).u_vektor()

        vektor_tocka_yos = Duzina(self, self + Tocka(0, 1)).u_vektor()

        kut = (vektor_tocka_n_1.kut_izmedu_vektora360(vektor_tocka_yos)
               - vektor_tocka_n.kut_izmedu_vektora360(vektor_tocka_yos))

        if kut > math.pi:
            kut -= 2 * math.pi

        if kut < -math.pi:
            kut += 2 * math.pi

        zbroj_kuteva += kut

    if abs(zbroj_kuteva) - EPSILON < 0 and abs(zbroj_kuteva) + EPSILON > 0:
        unutra = -1
    else:
        unutra = 1

    return unutra
```



Slika 8: Prikaz implementacije *Winding Number* algoritma u Pythonu

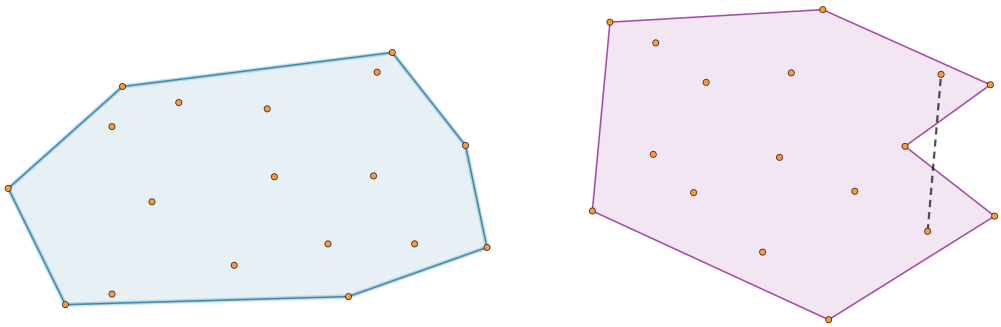
Ova implementacija ima složenost $O(n)$ jer samo moramo proći kroz sve vrhove poligona i izračunati određen kut. *Winding Number* i *Ray Casting* algoritam vraćati će iste rezultate za jednostavne poligone, no moguće je da će vraćati drugačija rješenja za složene poligone. Iako vraćaju drugačije rezultate, ni jedan ni drugi nisu pogrešni, ovisno o definiciji. Naime, u složenim poligonima koji se preklapaju, unutrašnjost poligona se može drugačije definirati za područje preklapanja [8].



Slika 9: Razlika u definicijama unutrašnjosti za *Ray Casting* i *Winding Number* algoritam. U RC algoritmu (lijevo) točka je izvan poligona, a u WN algoritmu (desno) točka pripada poligonu

5. Određivanje konveksne ljuske

Za skup točaka S kažemo da je konveksan ako su sve dužine definirane s bilo koje dvije točke T_1 i T_2 iz skupa potpuno sadržane u skupu točaka S. U suprotnome, skup je konkavan. Konveksna ljuska skupa S je najmanji konveksni skup točaka koji obuhvaća sve točke iz skupa S. Konceptualno, konveksnu ljusku možemo zamisliti kao podskup točaka nekog skupa koje čine poligon oko svih ostalih točaka iz skupa tako da su sve moguće dužine između točaka iz originalnog skupa potpuno sadržane u tom poligonu. Zanimljiva vizualizacija konveksne ljuske je ako točke iz skupa S zamislimo kao čavle zabijene u drvenoj ploči gdje svaki čavao predstavlja jednu točku iz skupa. Zatim, zamislimo da uzmemo gumicu i stavimo je oko tih čavli tako da se svi nalaze unutar gumice. Svi čavli koji diraju gumicu predstavljaju konveksnu ljusku [10].



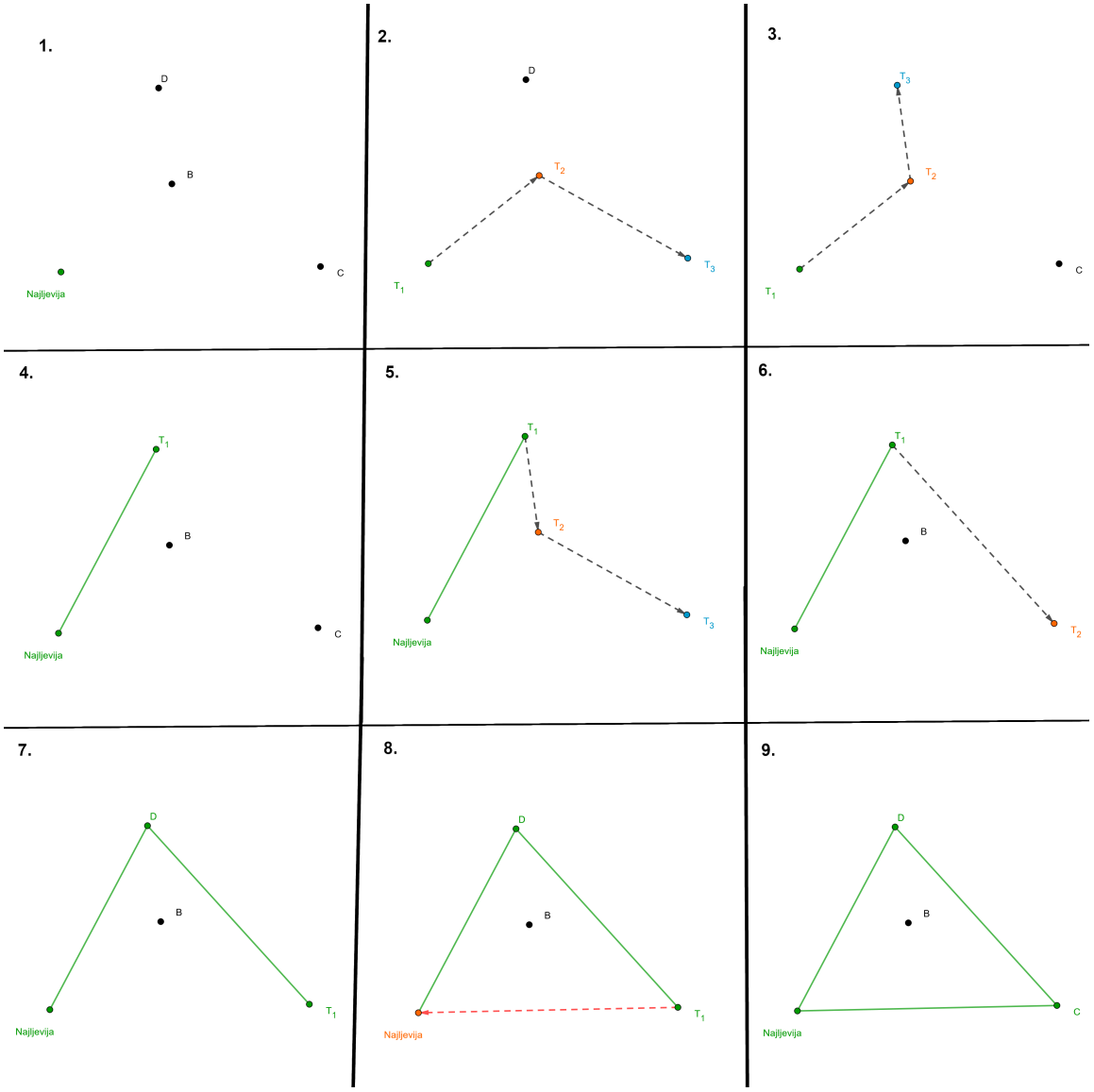
Slika 10: Konveksni (plavo) i konkavni (ljubičasto) skup točaka

Konveksne ljuske su vrlo korisne za praćenje i ocrtavanje objekata pa se često koriste u računalnom vidu. Osim toga, konveksne ljuske koriste se u izbjegavanju sudara, obradi slika, teoriji igara... Kako je konveksna ljuska jedna od najčešće korištenih struktura u računalnoj geometriji, postoji mnogo algoritama za njeno pronalaženje, a neki od najpoznatijih su *Jarvis March* algoritam, *Graham Scan* algoritam i *QuickHull* algoritam. Naravno, postoje i naivni algoritmi, no praktički su neiskoristivi zbog svoje vremenske složenost, a nisu mnogo lakši za implementirati od nekih optimalnijih algoritama [11].

5.1. *Jarvis March* algoritam

Jedan od najlakših algoritama za traženje konveksne ljuske skupa točaka je *Jarvis March* algoritam. Još se naziva i *Gift Wrapping* algoritam (algoritam zamatanja poklona) zbog načina na koji se izvodi. Prvi korak algoritma je traženje "najljevije" točke u zadanom skupu točaka jer će ona sigurno pripadati konveksnoj ljusci. Nju ćemo označiti kao točku T_1. Nakon toga, u drugom koraku biramo prvu sljedeću točku u skupu i označavamo je kao potencijalnu točku konveksne ljuske T_2. Zatim, u trećem koraku iteriramo kroz skup točaka dok ne pronađenom točku T_3 takvu da su točke T_1,T_2 i T_3 poredane u smjeru suprotnom od kazaljke na satu. Još jedan moguć način definiranja pravilne točke T_3 je ako je točka T_3 s lijeve strane vektora $T_1\vec{T}_2$. Kada naiđemo na takvu točku, T_2 poprima vrijednost točke T_3 i nastavljamo prolaziti kroz točke zadanog skupa tražeći točku novu točku T_3 za koju su točke T_1, T_2 i T_3 poredane u smjeru obrnutom od kazaljke na satu. Nakon što su pregledane sve točke u skupu, točku T_2 bilježimo kao točku u konveksnoj ljusci. Zatim, prije ponavljanja drugog i trećeg koraka, točka T_1 poprima vrijednost posljednje pronađene točke konveksne ljuske i završavamo sa trećim korakom. Drugi i treći korak ponavljamo sve dok se ne vratimo do najljevije točke.

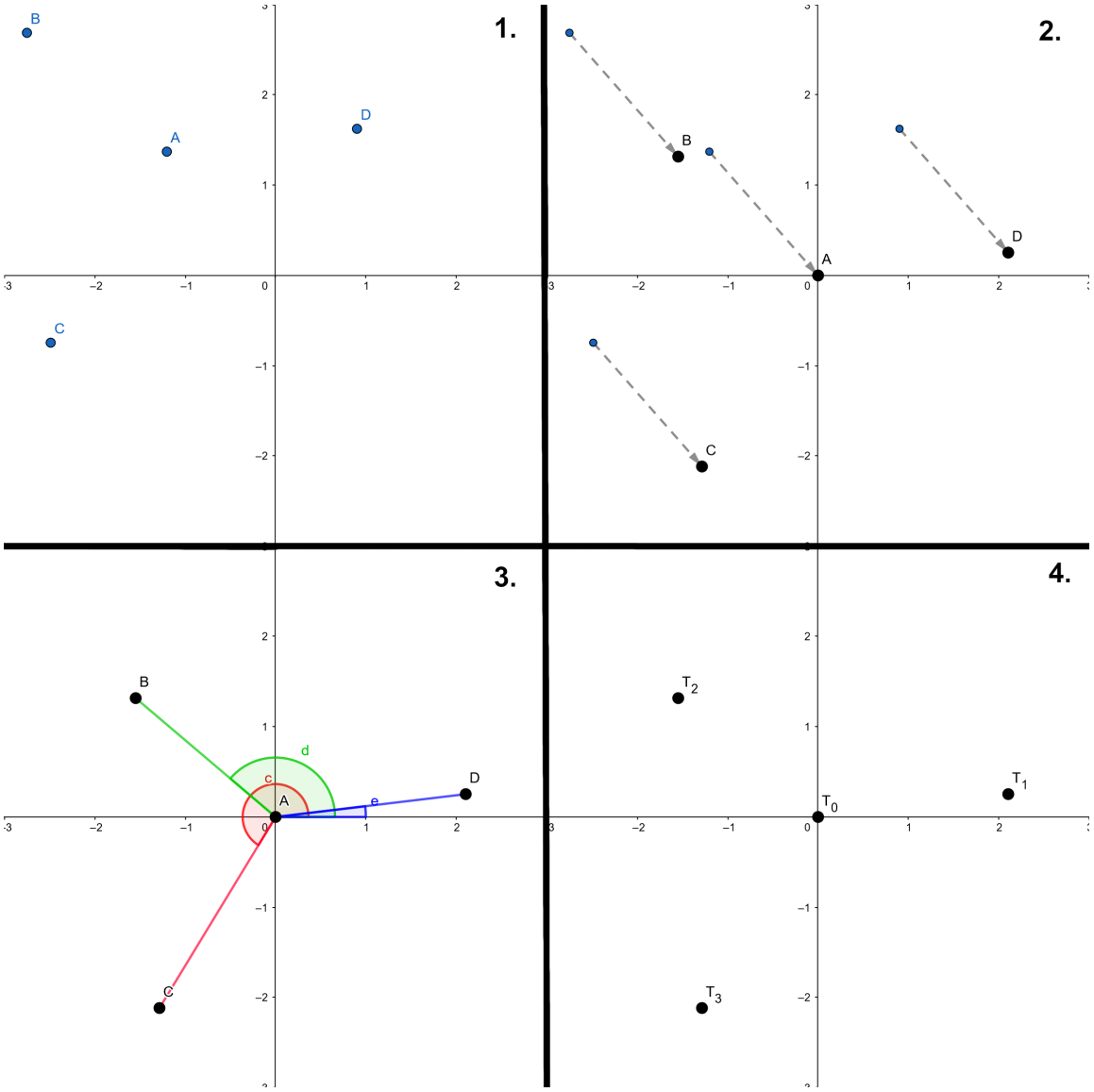
Algoritam se temelji na svojstvu konveksne ljuske da su svi unutarnji kutovi poligona kojeg čini konveksna ljuska manji ili jednaki $180°$. Ukoliko su tri točke, T_1, T_2 i T_3, poredane u smjeru kazaljke na satu, tada je kut između vektora $T_1\vec{T}_2$ i $T_2\vec{T}_3$ veći od $180°$. Tada znamo da druga točka T_2 u tom poretku ne pripada konveksnoj ljusci. Složenost algoritma je $O(nh)$ gdje je n broj točaka u zadanom skupu, a h je broj točaka konveksne ljuske. U najgorem slučaju kada su sve točke u skupu dio konveksne ljuske, složenost će biti $O(n^2)$. [11].



Slika 11: Prikaz izvođenja *Jarvis March* algoritma za 4 točke

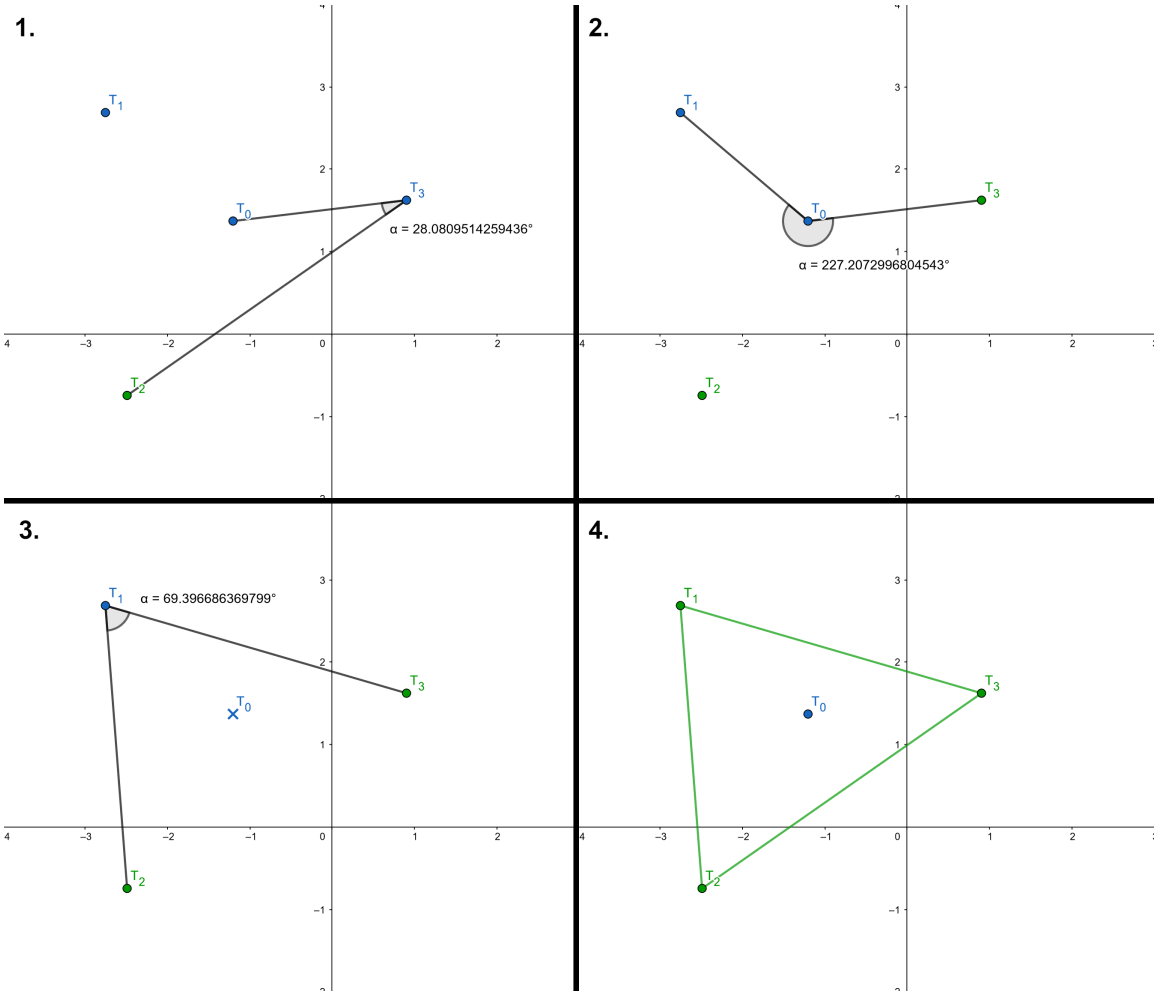
5.2. *Graham Scan* algoritam

Kako *Jarvis March* algoritam može postići složenost od $O(n^2)$, potrebno je naći optimalniji algoritam. *Graham Scan* je bolji algoritam za traženje konveksne ljuske i pronalazi ju u optimalnom vremenu. Njegova složenost za najgori slučaj je $O(n \log n)$.Osim toga, *Graham Scan* algoritam je vrlo jednostavan za implementirati jer se temelji na sortiranju i pretraživanju skupa točaka. Svejedno, *Graham Scan* algoritam ne mora biti najbrži za prosječne slučajeve izvođenja. Prvi korak algoritma je odabir točke T_0 prema kojoj ćemo izvršiti sortiranje skupa točaka prema polarnom kutu i udaljenosti od te točke. Polarni kut je kut koji neka točka zatvara s apscisom u smjeru suprotnom od kazaljke na satu. Kako bi ga dobili, prvo sve točke translatiramo tako da se točka T_0 nalazi u ishodištu koordinatnog sustava. Točka T_0 može biti bilo koja točka iz zadanog skupa. Zatim sve točke sortiramo prema kutu koji one zatvaraju s apscisom. Translacija se izvodi kako bi mogli povući dužinu $\overline{T_0T_n}$ gdje je T_n bilo koja točka iz zadanog skupa. Dužinu T_0T_n koristimo kako bi odredili kut između nje i apscise, odnosno između točke T_n i apscise. U samoj implementaciji ne mora dolaziti do translacije, ali to će biti objašnjeno u samoj implementaciji. Ako naiđemo na dvije točke koje imaju isti polarni kut, tada je drugi kriterij za sortiranje koja je točka udaljenija od točke T_0. Ovakvim sortiranjem smo zapravo točke poredali "u krug" oko točke T_0, a kretanjem tim krugom ćemo po redu dobiti točke koje pripadaju konveksnoj ljusci. Sortirane točke su pohranjene u dvostruko vezanu listu zbog lakšeg pretraživanja [6].



Slika 12: Prikaz sortiranja prema polarnom kutu

U drugom koraku, tražimo točku $T_{\min}$ koja ima najmanju ordinatu od svih točaka u zadanom skupu. Nakon sortiranja, kreće "skeniranje" sortiranog skupa. Za bilo koje tri uzastopne točke T_n, T_{n+1} i T_{n+2} definiramo "lijevo" i "desno skretanje". Kažemo da tri točke T_n, T_{n+1} i T_{n+2} čine "lijevo skretanje" ako je kut koji one zatvaraju manji od $180°$ (odnosno π). U suprotnom, točke čine "desno skretanje". Skeniranje se odvija provjeravanjem čine li neke tri uzastopne točke T_n, T_{n+1} i T_{n+2} lijevo ili desno skretanje. Ukoliko čine lijevo skretanje, pomičemo se za jednu točku unaprijed i dalje provjeravamo kakvo skretanje čine tri nove točke T_{n+1}, T_{n+2} i T_{n+3}. Ako naiđemo na desno skretanje, drugu točku u nizu, T_{n+1}, izbacujemo iz liste i dalje skeniramo točke T_{n-1}, T_n i T_{n+3}. Točke se "skeniraju" sve dok ne dođemo ponovno do točke $T_{\min}$. Skeniranje kreće od točke $T_{\min}$ i njenih dvaju sljedbenika iz skupa sortiranih točaka, T_1 i T_2. Na kraju skeniranja imamo poredane točke konveksne ljuske za zadani skup točaka. Kako se točke mogu sortirati sa složenosti $O(n \log n)$, a nakon toga se skeniranje izvodi jednom za svaku točku u sortiranom skupu, složenost ne može biti veća od $O(n \log n)$ [6].



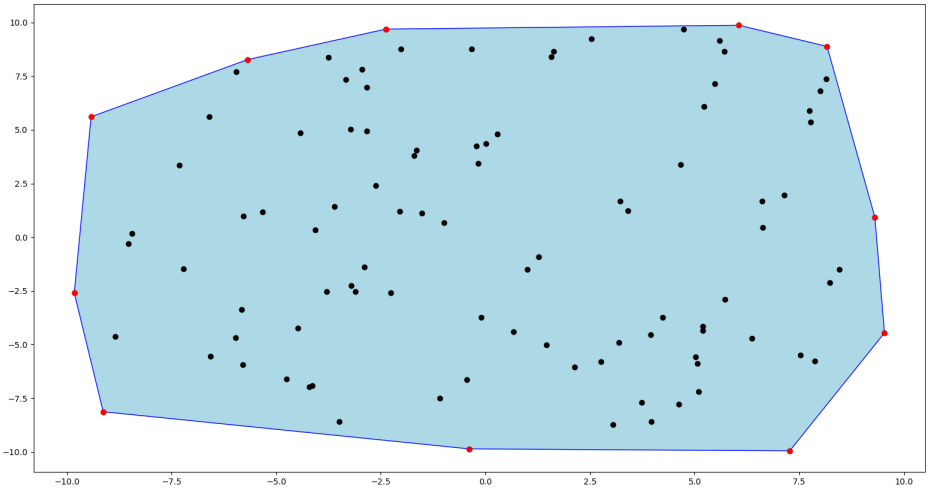
Slika 13: Izvođenje Graham skena nakon pronađene točke s najmanjom ordinatom

Implementacija *Graham Scan* algoritma

Implementacija *Graham Scan* algoritma se razlikuje od opisanog *Graham Scana* u nekoliko detalja koji ne utječu na ukupnu složenost algoritma. Za početak, prvih par linija samo provjerava sastoji li se dana lista objekata klase `Tocka` od najmanje dva elementa jer konveksnu ljusku ne možemo odrediti za manje od dvije točke. Zatim izbacujemo sve duplikate iz zadane liste i krećemo s izvođenjem *Graham Scana*. Umjesto da prvo sortiramo točke prema polarnom kutu, u implementaciji prvo tražimo točku s najmanjom ordinatom. Tek nakon toga sortiramo točke prema "polarnom" kutu. Umjesto da sve točke translatiramo i određujemo polarni kut između tih točaka i x-osi, stvaramo vektor koji je paralelan s x-osi i prolazi kroz točku s najmanjom ordinatom te točke sortiramo prema kutu koji one zatvaraju s tim vektorom. Rezultat će ostati isti samo nećemo prvo morati tražiti točku s najmanjom ordinatom u listi kako bi krenuli sa skeniranjem. Nakon toga, krećemo sa skeniranjem. Da bi provjerili dolazi li do lijevog ili desnog skretanja, dovoljno je provjeriti je li vektorski produkt između vektora koji definiraju prva i druga točka i vektora koji definiraju druga i treća točka jednak ili manji od nule. Ako je vektorski produkt manji ili jednak nuli, tada se treća točka nalazi s lijeve desne strane prvog vektora što je isto kao da dva vektora zatvaraju kut veći od $180°$. Na kraju, vraćamo sortiranu listu točaka

koje čine konveksnu ljusku.

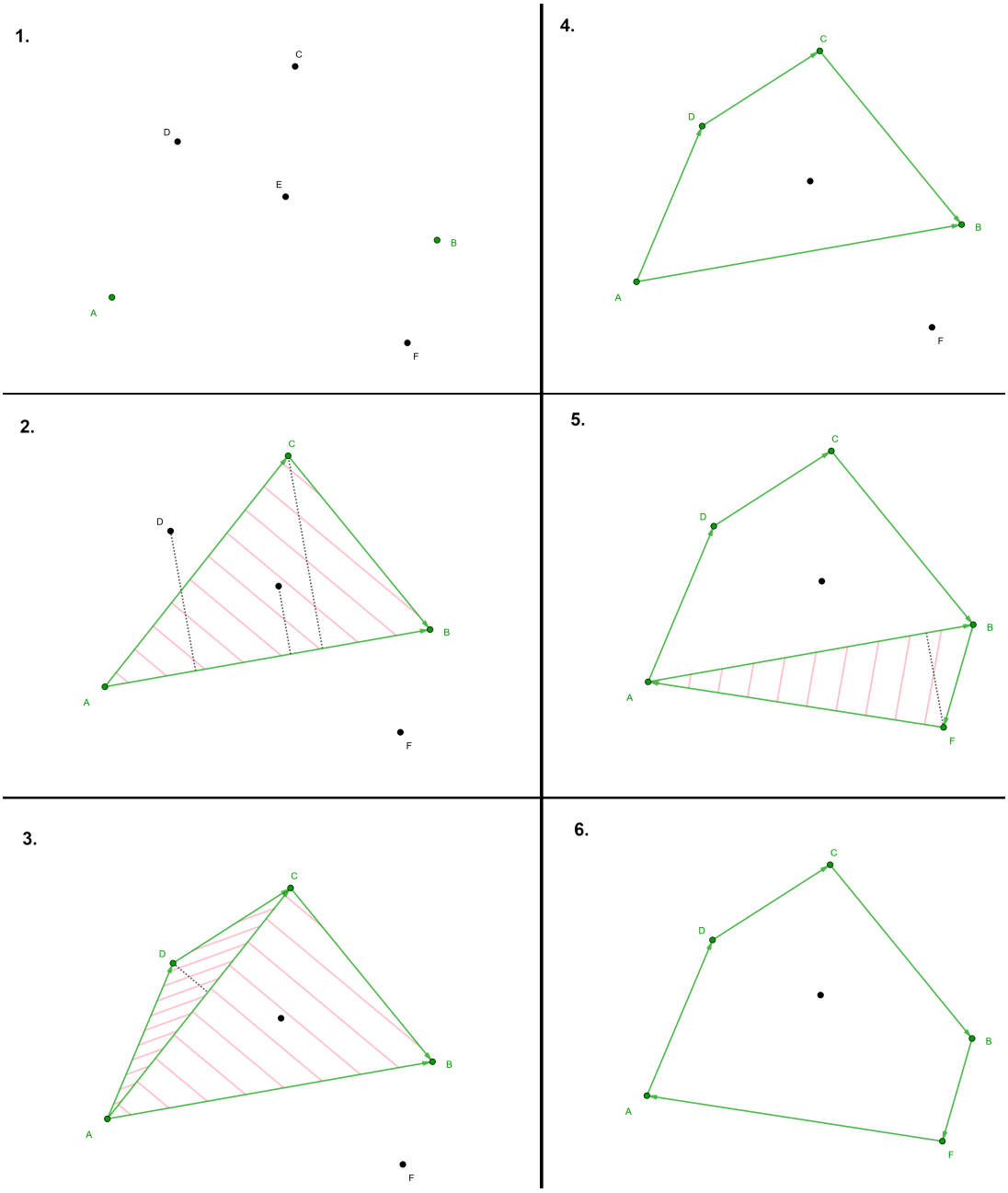
```
def konveksna_ljuska(tocke):
    br_tocaka = len(tocke)

    if br_tocaka < 2:
        raise PremaloTocakaError("Premalo točaka za određivanje konv. ljuske")
    if br_tocaka == 2:
        return tocke

    tocke = list(set(tocke))

    indeks_najnize_tocke = pronadi_indeks_najnize_tocke(tocke)
    najniza_tocka = tocke[indeks_najnize_tocke]

    tocke.pop(indeks_najnize_tocke)
    tocke.sort(key=lambda tocka: (najniza_tocka.polarni_kut(tocka),
                                  tocka.udaljenost_od(najniza_tocka)))
    tocke.append(najniza_tocka)

    indeks_n = -1
    tocka_1 = tocke[indeks_n]
    tocka_2 = tocke[indeks_n + 1]
    tocka_3 = tocke[indeks_n + 2]
    while tocka_3 != najniza_tocka:
        duzina_t1_t2 = Duzina(tocka_1, tocka_2)

        if tocka_3.lijevo_od(duzina_t1_t2):
            indeks_n += 1
        else:
            tocke.pop(indeks_n + 1)
            indeks_n -= 1

        tocka_1 = tocke[indeks_n]
        tocka_2 = tocke[indeks_n + 1]
        tocka_3 = tocke[indeks_n + 2]

    return tocke
```



Slika 14: Prikaz implementacije *Graham Scan* algoritma

5.3. *QuickHull* algoritam

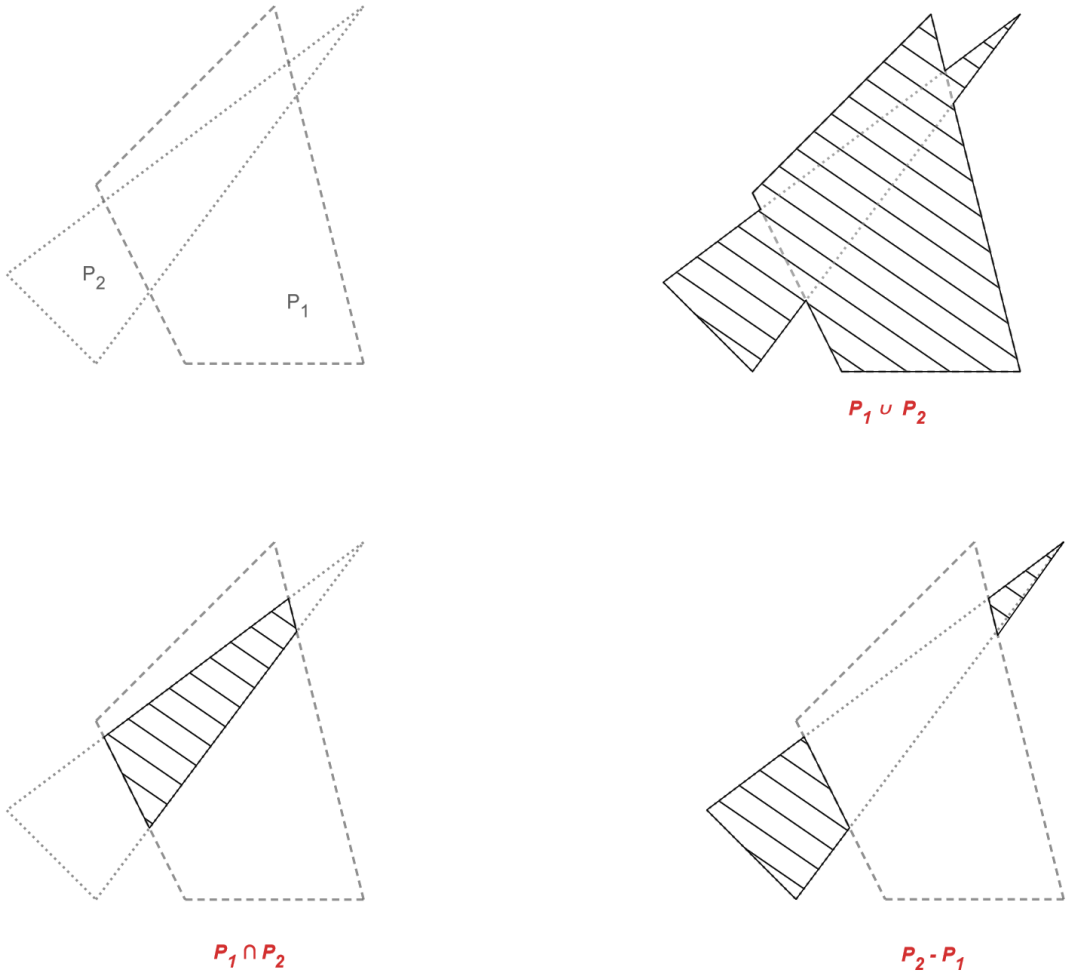
QuickHull algoritam je praktički najbrži algoritam za traženje konveksne ljuske. Iako mu je složenost za najgori slučaj $O(n^2)$, *QuickHull* se prosječno izvodi sa složenosti $O(n \log n)$ zbog čega je postao najkorišteniji algoritam za traženje konveksne ljuske. Algoritam se temelji na podijeli zadanog skupa točaka na manje podskupove i njihovom pretraživanju. Kao i u *Jarvis March* i *Graham Scan* algoritmima, prvo je potrebno pronaći točke koje sigurno pripadaju konveksnoj ljusci. Za dani skup točaka uvijek postoje četiri točke koje definitivno pripadaju konveksnoj ljusci (ako se skup točaka sastoji od barem 4 točke). To su točke s najmanjom apscisom, najmanjom ordinatom, najvećom apscisom i najvećom ordinatom. U prvom koraku algoritma tražimo dvije od tih četiri točke i tada kroz njih povlačimo usmjerenu dužinu $\vec{D}$. Najčešće, za te dvije točke se uzimaju točka s najmanjom i točka s najvećom apscisom ("najljevija" i "najdesnija" točka u zadanom skupu). Novonastala usmjerena dužina $\vec{D}$ dijeli zadani skup točaka na dva dijela. Nakon toga, pronalazimo točku T koja se nalazi najdalje od dužine $\vec{D}$ s njene lijeve strane. Točka T pripada konveksnoj ljusci, a sve točke koje se nalaze u trokutu $\Delta D_A T D_B$, gdje su D_A i D_B krajnje točke dužine D, sigurno se ne nalaze u konveksnoj ljusci zadanog skupa stoga ih više ne moramo provjeravati. Stranice trokuta $\Delta D_A T D_B$ su tri usmjerene dužine $\overrightarrow{D_A T}$, $\overrightarrow{T D_B}$ i $\overrightarrow{D_A D_B}$. Za stranice $\overrightarrow{D_A T}$ i $\overrightarrow{T D_B}$ tražimo najudaljeniju točku s njihove lijeve strane. Postupak traženja najudaljenije točke i izbacivanja točaka koje se nalaze u trokutu koju ta točka čini sa stranicom trokuta koji je nastao u prijašnjem koraku ponavljamo sve dok više nema točaka za pretraživanje. Cijeli taj postupak ponavljamo i za desnu stranu zadanog skupa točaka. Algoritam ima složenost $O(n \log n)$ jer je za početno pretraživanje najudaljenijih točaka potrebno iterirati kroz sve točke s lijeve i desne strane dužine $\vec{D}$. Kasnije se taj skup točaka za pretraživanje smanjuje (ili u najgorem slučaju ostaje isti) svaki put kada pronađemo najudaljeniju točku i uklonimo sve točke koje se sigurno ne nalaze u konveksnoj ljusci [12].



Slika 15: Prikaz *QuickHull* algoritma

6. Booleove operacije nad poligonima

Osnovne Booleove operacije nad skupovima su unija ($\cup$), presjek ($\cap$) i razlika ($-$). Svaka dva poligona možemo zamisliti kao dva skupa točaka zatvorena dužinama. Tako i nad poligonima možemo izvoditi Booleove operacije. Unija dvaju poligona predstavlja skup svih točaka koje pripadaju jednom ili drugom poligonu, odnosno skup svih točaka oba poligona. Presjek dvaju poligona je skup svih točaka koje se nalaze u prvom i u drugom skupu, odnosno skup zajedničkih točaka oba poligona. Na kraju, razlika dvaju poligona P_1 i P_2 je skup svih točaka koje pripadaju poligonu P_1, ali ne pripadaju poligonu P_2, odnosno to je skup točaka poligona P_1 bez točaka iz poligona P_2. Za razliku od unije i presjeka, razlika nije komutativna. Neki od konceptualno lakših algoritama za provođenje Booleovih operacija nad poligonima su Greiner-Hormann i Avraham-Knott algoritam [13].



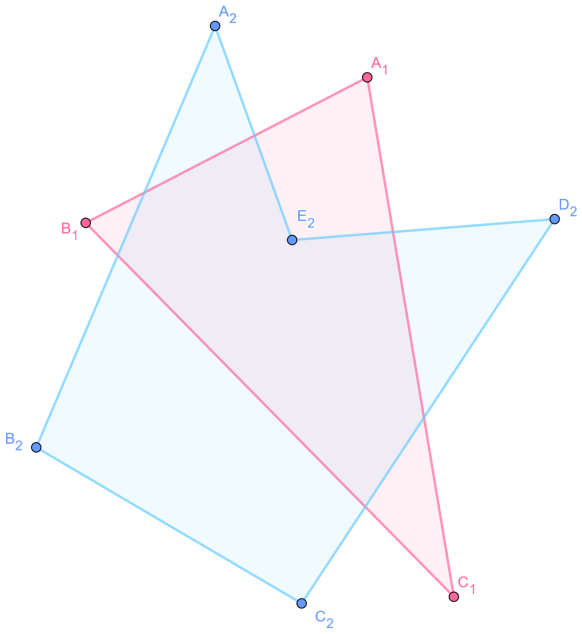
Slika 16: Prikaz osnovnih Booleovih operacija

6.1. Greiner-Hormann algoritam

Greiner-Hormann algoritam je jedan od najpoznatijih algoritama za izvođenje presjeka dvaju poligona. Algoritam radi za složene i nekonveksne poligone. Dodatno, algoritam se može vrlo jednostavno preinačiti tako da podržava uniju i razliku poligona. Za razliku od većine algoritama za Booleove operacije nad poligonima koji postoje, Greiner-Hormann algoritam je mnogo jednostaniji za implementirati.

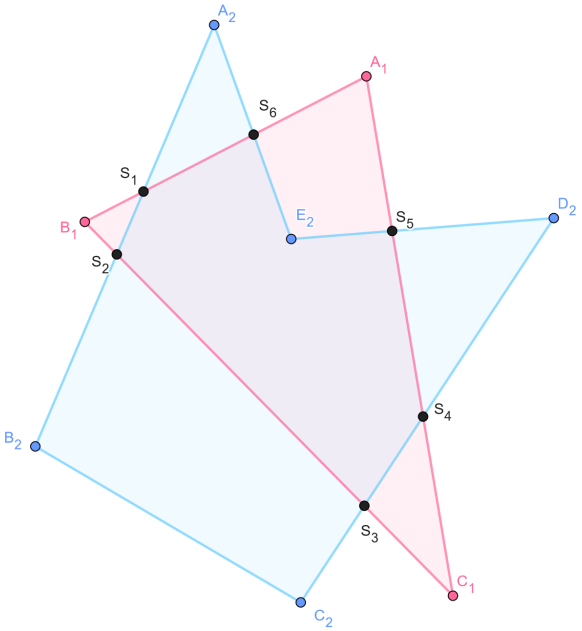
Algoritam se sastoji od tri koraka. Prvi je pronalaženje sjecišta između rubova poligona, drugi je označavanje sjecišta kao "ulaznih" ili "izlaznih", a treći je povezivanje određenih točaka oba poligona kako bi dobili njihov presjek. Za izvođenje algoritma potrebne su dvije strukture: posebna struktura za prikaz točke i dvostruko vezana lista tih točaka. Za točku želimo da sadrži nekoliko atributa. Koordinate, sljedeći i prijašnji vrh poligona, sljedeći poligon, je li točka sjecište, je li točka ulazna ili izlazna, koja je susjedna točka i alfa koeficijent. Koordinate, sljedeći i prijašnji vrh poligona su potrebne za prikaz točke i kretanje kroz listu. Podatak o sljedećem poligonu će služiti za slučaj da presjekom ili razlikom dobijemo više poligona, a sadržavati će idući nastali poligon nakon trenutnog. Svaka točka koja je sjecište mora tako biti i označena jer su sjecišta vrlo važan dio izvođenja algoritma. Sjecište može biti ulazno ili izlazno, odnosno ako se krećemo po jednom poligonu i nađemo na sjecište s drugim, bilježimo ulazimo li ili izlazimo iz poligona u toj točci. Atribut za susjednu točku ponovno se odnosi samo na sjecišta. Svaki put kada naiđemo na sjecište, ono mora pokazivati na svoju kopiju u skupu točaka drugog poligona. Na kraju, alfa koeficijent koji opisuje gdje se sjecište nalazi na rubu, a ima vrijednost od 0 do 1. Ukoliko je vrijednost točno 0 ili točno 1, znamo da je sjecište ujedno i vrh što je zapravo rubni slučaj koji ćemo kasnije riješiti.

U prvom koraku se prolazi kroz svaki rub oba poligona i traže se njihova sjecišta. Za svako pronađeno sjecište, računa se njegov alfa koeficijent. Svako sjecište se pohranjuje u dvostruko vezanu listu između točaka između kojih se nalazi i u poligonu. Sjecište se mora pohraniti u dvostruko vezanu listu točaka oba poligona. Kao što je prije navedeno, svako sjecište sadrži pokazivač na svoju kopiju iz liste točaka drugog poligona. Ako nakon prvog koraka ne postoji ni jedno sjecište, jedan od dva poligona se nalazi u drugom. U tom slučaju, uzimamo bilo koji vrh bilo kojeg poligona i određujemo nalazi li se on unutar ili izvan drugog poligona. Ako se taj vrh ne nalazi unutar drugog poligona, znamo da se cijeli drugi poligon nalazi unutar prvog i potrebno je vratiti cijeli drugi poligon kao presjek. U suprotnom, vraća se prvi poligon. Prvi korak algoritma za neka dva poligona prikazan je na slici [17]. $P1$ i $P2$ predstavljaju dvostruko vezane liste točaka svakog poligona.



P1 = {A1,B1,C1}

P2 = {A2,B2,C2,D2,E2}

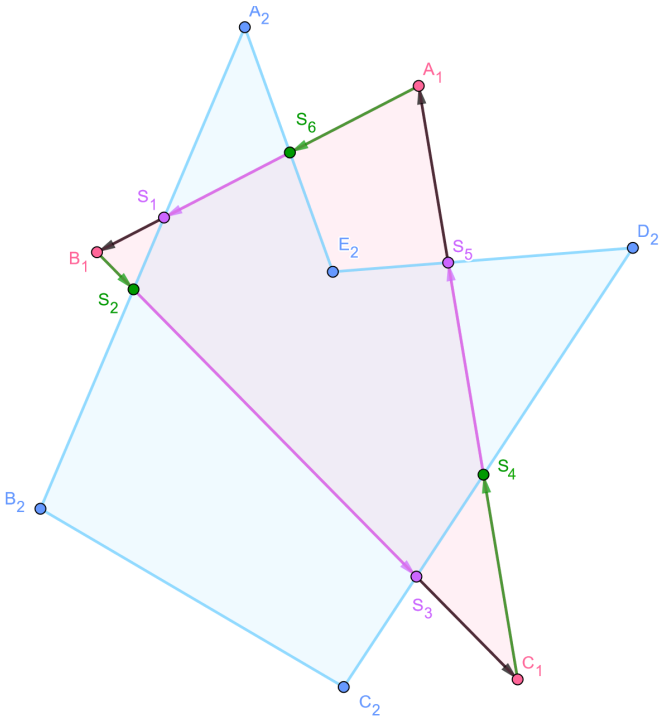


P1 = {A1, S6, S1, B1, S2, S3, C1, S4, S5}

P2 = {A2, S1, S2, B2, C2, S3, S4, D2, S5, E2, S6}

Slika 17: Prvi korak Greiner-Hormann algoritma

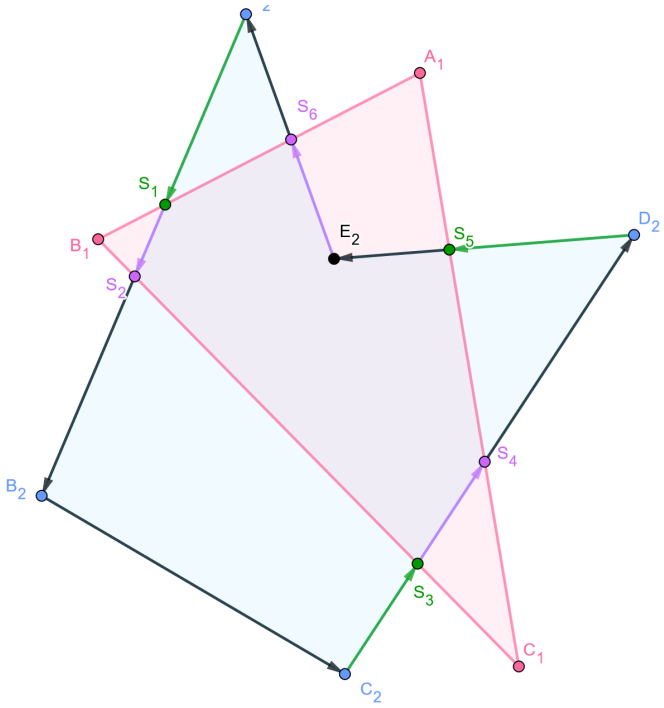
U drugom koraku se krećemo po točkama oba poligona. Svaki put kada naiđemo na sjecište, bilježimo je li ono ulazno ili izlazno. Prvo je potrebno uzeti početni vrh od kojeg krećemo i odrediti nalazi li se on unutar drugog poligona. Ako da, tada znamo da će prvo sjecište biti izlazno. U suprotnom, prvo sjecište je ulazno. Svako iduće sjecište na koje naiđemo će imati suprotan "smjer" od prijašnjeg. Na kraju drugog koraka, sjecišta će imati naizmjenične vrijednosti za ulaznost, odnosno izlaznost.



P1 = {A1, S6, S1, B1, S2, S3, C1, S4, S5}

P2 = {A2, S1, S2, B2, C2, S3, S4, D2, S5, E2, S6}

Slika 18: Drugi korak Greiner-Hormann algoritma za točke liste $P1$

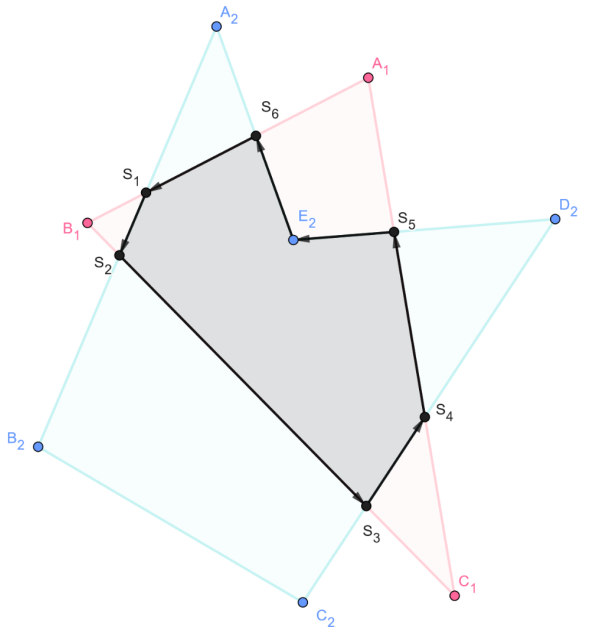


P1 = {A1, S6, S1, B1, S2, S3, C1, S4, S5}

P2 = {A2, S1, S2, B2, C2, S3, S4, D2, S5, E2, S6}

Slika 19: Drugi korak Greiner-Hormann algoritma za točke liste $P2$

Na kraju, potrebno je još jednom proći kroz jedan od poligona. Treći korak počinje od prvog neobrađenog sjecišta poligona. Ukoliko naiđemo na ulazno sjecište, krećemo se unaprijed po listi sve dok ne dođemo do prvog idućeg sjecišta u njoj. Pritom svaku točku na koju naiđemo pohranjujemo kao vrh poligona koji će predstavljati presjek. Ako je pak prvo sjecište izlazno, krećemo se unazad po listi dok ne naiđemo na prvo sljedeće sjecište. Kao i za ulaznu točku, svaku točku pohranjujemo kao vrh poligona u posebnu listu. Kada dođemo do idućeg sjecišta, pomičemo se u listu drugog poligona na susjedno sjecište trenutnog sjecišta. Zatim se ponovno krećemo kroz listu sve dok ne naiđemo na iduće sjecište. Ovaj postupak ponavljamo za svako neobrađeno sjecište dok više ne bude neobrađenih sjecišta. Za svako novo neobrađeno sjecište nastaje novi poligon koji će biti rezultat presjeka dva poligona.



Slika 20: Treći korak Greiner-Hormann algoritma

Rubni slučaj koji može nastati je ako se vrh jednog poligona nalazi na rubu drugog. Algoritam će primjetiti takav slučaj ako je alfa koeficijent jedne točke 0. Tada je potrebno tu točku neznatno pomaknuti od ruba drugog poligona. Kako bi izveli operacije unije i razlike, potrebno je samo promijeniti treći korak algoritma. Kako bi dobili uniju, potrebno je promijeniti redoslijed kojim se krećemo po listama. Kada naiđemo na ulaznu točku, krećemo se unazad po listi, a kada naiđemo na izlaznu točku, krećemo se unaprijed. Za razliku, ukoliko se nalazimo u listi prvog poligona, krećemo se unazad za uzlazne točke i unaprijed za izlazne, a ako se nalazimo u listi drugog poligona, za uzlazne točke se krećemo unaprijed, a za izlazne unazad. Složenost algoritma je $O(n^2)$ u najgorem slučaju, a $0(nm)$ u prosječnom gdje je n broj rubova u prvom poligonu, a m broj rubova u drugom [14].

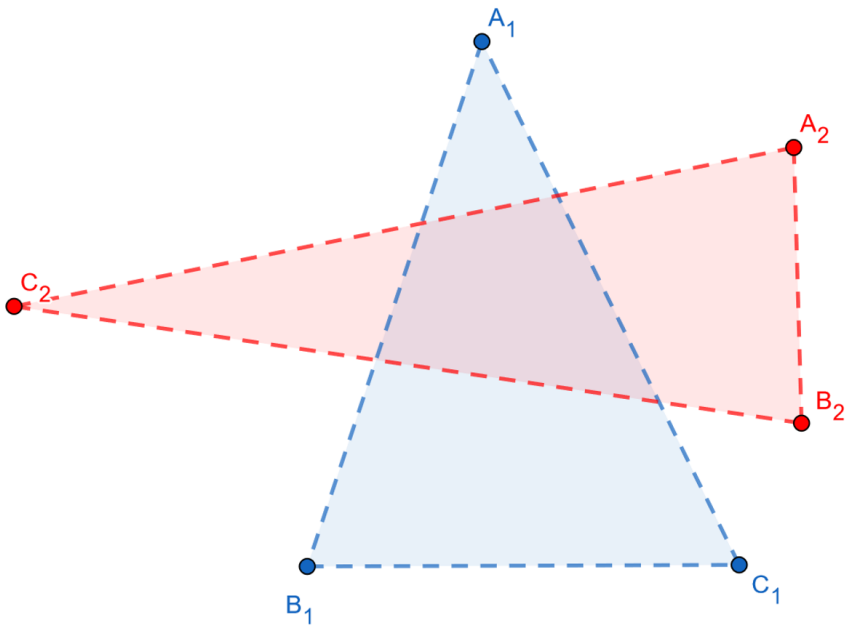
6.2. Avraham-Knott algoritam

Ovaj stariji algoritam je poseban jer podržava sve operacije nad svim vrstama poligona. Nije bitno jesu li poligoni jednostavni, složeni, konveksni ili konkavni, pa čak ni je li poligon "otok" ili "rupa". Poligoni čija je unutrašnjost beskonačna nazivaju se rupama, a poligoni s ograničenim unutrašnjostima nazivaju se otoci. Zbog jednostavnosti, ovaj algoritam ćemo opisati samo na primjeru s „otocima“ koji mogu biti jednostavni ili složeni.

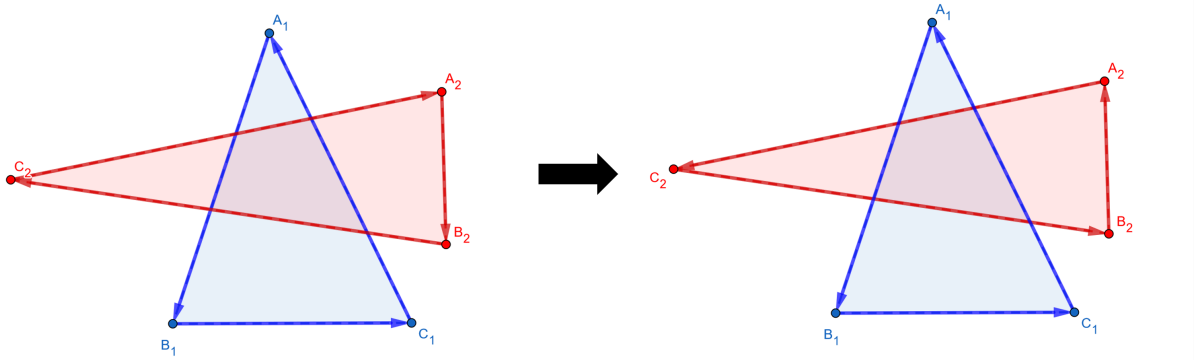
Algoritam se sastoji od dvije faze: klasifikacija dužina ulaznih poligona i izrade rezultatskog poligona. U prvoj fazi je potrebno proći kroz sve vrhove poligona, označiti ih kao unutar, izvan ili na rubu drugog poligona, te pronaći sva sjecišta između rubova ulaznih poligona. Točke moraju biti posložene tako da svaka točka sa svojim susjedom u strukturi čini neki rub ili dio nekog ruba. Svaki rub u strukturi ćemo nazvati fragmentom nakon što dodamo sjecišta u listu točaka poligona. Rubni fragmenti su rastavljeni rubovi koje ćemo kasnije spajati u rezultatski poligon. Nakon što smo poligone rastavili na fragmente, potrebno je za svaki fragment specificirati nalazi li se na rubu, unutar ili izvan poligona.

U drugoj fazi, ovisno o tome koju operaciju želimo izvesti, algoritam će spajati određene fragmente jednog i drugog poligona kako bi dobili uniju, razliku ili presjek tih poligona. Svaki put kada neki fragment iskoristimo, izbacujemo ga iz liste fragmenata kako bi mogli dalje dobivati rezultatske poligone. Sastavljanje poligona je zapravo iteriranje kroz listu fragmenata i traženje fragmenata koji imaju zajedničku krajnju točku dok se ne vratimo na početni i tako zatvorimo jedan rezultatski poligon. Ovo se ponavlja sve dok nisu izbačeni svi fragmenti iz liste fragmenata.

Prva faza sastoji se od pet koraka, a druga od samo jednog. Prvi korak prve faze je određivanje i mijenjanje orijentacije danih poligona ako je to potrebno. Orijentacija poligona je smjer u kojem se krećemo po njegovim vrhovima. Ukoliko tražimo presjek ili uniju, poligoni moraju imati istu orijentaciju, a ukoliko tražimo razliku, poligoni moraju imati različite orijentacije.

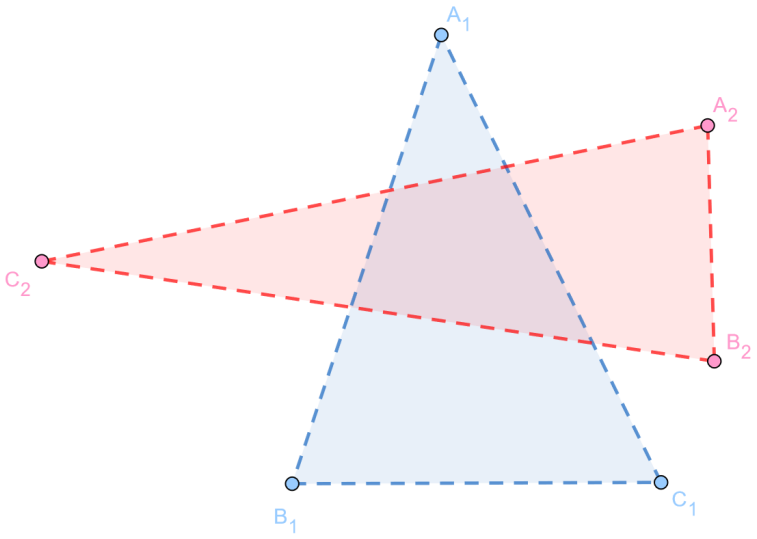


Slika 21: Početni poligoni koji se sijeku



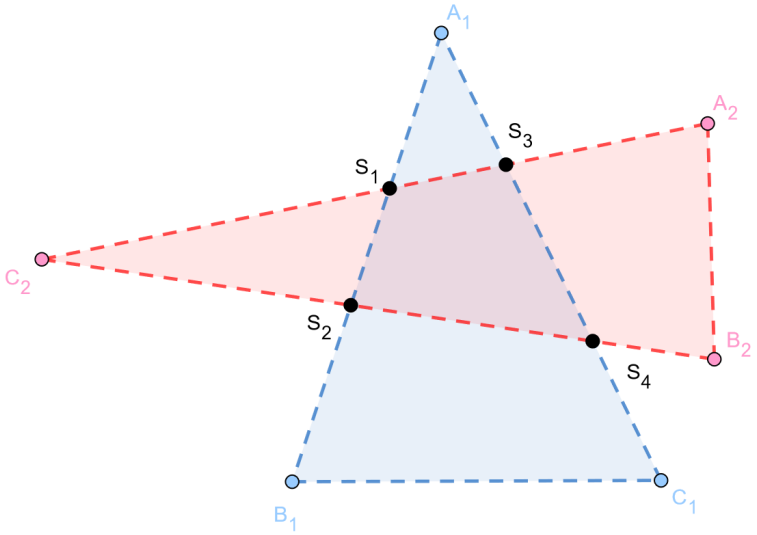
Slika 22: Izmijena orijentacije poligona kako bi mogli naći uniju ili presjek poligona

Zatim, u drugom koraku, potrebno je klasificirati vrhove oba poligona kao vrhove koji se nalaze na rubu, izvan ili unutar drugog poligona. Točke moraju biti sortirane tako da svaka dva uzastopna vrha čine neki rub poligona.



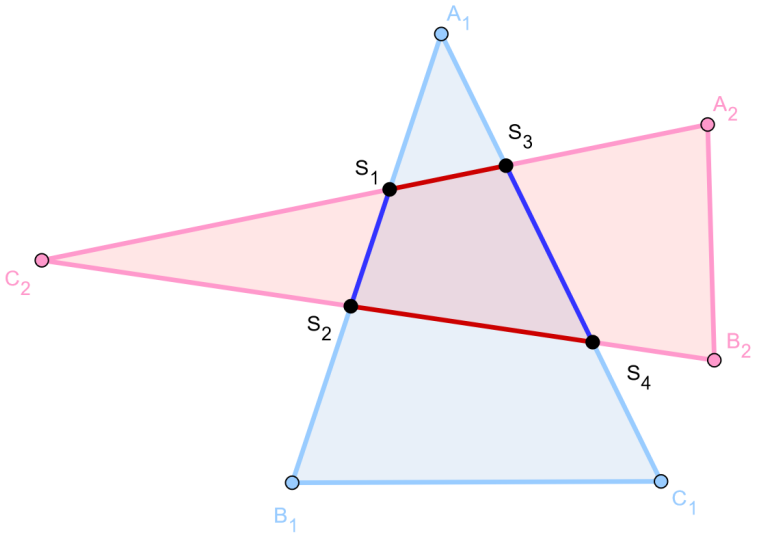
Slika 23: Klasifikacija točaka. U ovom slučaju sve točke su vanjske.

Nakon toga, tražimo sjecišta između rubova poligona. Svako sjecište se klasificira kao „na rubu“ osim ako se poligoni ne sijeku po cijelim rubovima tako da se ti rubovi preklapaju. U tom slučaju, obje krajnje točke tog ruba koji se preklapa se klasificiraju kao „na rubu“. Svaki put kada naiđemo na sjecište, ono se klasificira i pohranjuje u listu točaka poligona na odgovarajuće mjesto tako da svake dvije susjedne točke čine jedan fragment poligona. Kako bi se mogli podnijeti rubni slučajevi gdje se dva poligona sijeku u nekom vrhu, svaka točka se može pojavljivati najviše dva puta u listi točaka poligona.



Slika 24: Traženje sjecišta dvaju poligona. Svako sjecište definirano je kao točka na rubu

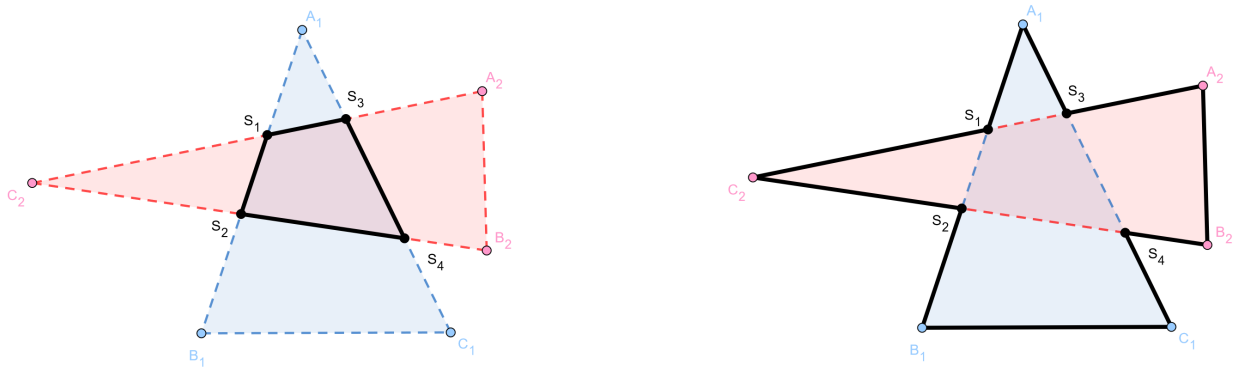
U četvrtom koraku, potrebno je klasificirati fragmente pomoću njihovih točaka. Ukoliko je ijedna točka fragmenta klasificirana kao unutarnja točka tada je cijeli fragment klasificiran kao unutarnji. Isto vrijedi i za vanjske fragmente. Ako su obje krajnje točke fragmenta na rubu poligona, tada se promatra gdje se nalaze točke na tom fragmentu. Ukoliko su sve točke takvog fragmenta unutar poligona, tada se on definira kao unutarnji fragment. Na isti način možemo odrediti hoćemo li taj fragment klasificirati kao izvan ili na rubu poligona. U implementaciji ovog algoritma ćemo vidjeti da je dosta provjeriti nalazi li se središte određenog fragmenta unutar, izvan ili na rubu poligona ako su obje krajnje točke na rubu nekog poligona.



Slika 25: Klasificiranje fragmenata. Tamnoplavo i tamnocrveno su označeni unutarnji fragmenti, a svijetloplavo i rozno su označeni vanjski fragmenti

Na kraju prve faze, potrebno je odabrati određene fragmente koje ćemo koristiti za sastavljanje rezultatskog poligona i pohraniti ih u listu. Za presjek, sačuvati ćemo sve unutarnje fragmente i sve fragmente na rubu. Za uniju, zadržati ćemo sve vanjske fragmente i sve fragmente na rubu. Za razliku, zadržavamo sve vanjske fragmente prvog poligona i sve unutarnje fragmente drugog poligona.

U drugoj fazi jednostavno prolazimo kroz sve zadržane fragmente i sastavljamo rezultatski poligon. Nasumice odabiremo jedan fragmente i tražimo prvi sljedeći čija je jedna krajnja točka jednaka jednoj od krajnjih točaka odabranog fragmenta. Taj postupak ponavljamo sve dok se ne vratimo do početnog fragmenta. Cijeli postupak se ponavlja dok više nema fragmenata u listi.



Slika 26: Odabir fragmenata za presjek (lijevo) i fragmenata za uniju (desno)

Složenost algoritma za najgori slučaj je $O((nm)^2)$ gdje je n broj vrhova prvog poligona, a m je broj vrhova drugog poligona. Algoritam najviše vremena gubi na traženju sjecišta jer je potrebno presjeći svaki rub prvog poligona sa svim rubovima drugog poligona [9].

Implementacija Avraham-Knott algoritma

Algoritam je implementiran samo za jednostavne poligone "otoke". U implementaciji, algoritam prima dva objekta klase `Poligon`, `self` i `drugi_poligon` , i broj koji predstavlja koju operaciju želimo izvršavati nad poligonima. Brojevi -1, 0 i 1 po redu predstavljaju operacije razlike, unije i presjeka. Ovisno o operaciji, definirati ćemo koje fragmente će algoritam zadržati i je li potrebno mijenjati orijentaciju poligona. Zatim, za izvršavanje algoritma, potrebne su dvije dodatne klase koje će u sebi sadržavati točku (`PomTocka`) ili dužinu (`PomDuzina`) i njihovu klasifikaciju. Klasifikacije su -1, 0 i 1, a redom označavaju da je točka ili dužina izvan, na rubu ili unutar poligona. Nakon provjere orijentacije oba poligona, klasificiramo vrhove prvog i drugog poligona tako da pozivamo funkciju za provjeru pripadnosti točke u poligonu. Nakon što smo vrhove oba poligona klasificirali i pohranili u zasebne liste, tražimo sjecišta između poligona. Za svaki poligon pozivamo funkciju `pom_tocke_sjecista` u koju se šalju obje liste klasificiranih točaka poligona. U funkciji se za svaki rub poligona traže sjecišta sa svim rubovima drugog poligona. Zatim, funkcijom `tocke_u_fragmente` spajamo točke u fragmente i odmah ih klasificiramo. U ovoj implementaciji, četvrti i peti korak su spojeni tako da odmah pri klasificiranju zadržavamo ili odbacujemo fragmente, ovisno o operaciji koja se izvršava. Tako nam ostane samo povezati preostale fragmente u poligone pomoću funkcije `povezi_fragmente`. Kako su u ovoj implementaciji poligoni predstavljeni kao skupovi poredanih točaka, nakon što smo dobili poredane fragmente, potrebno je njihove krajnje točke po redu pohraniti u listu koja će predstavljati vrhove rezultatskog poligona. Na kraju, sve dobivene poligone pohranjujemo u listu poligona koju metoda vraća kao rezultat.

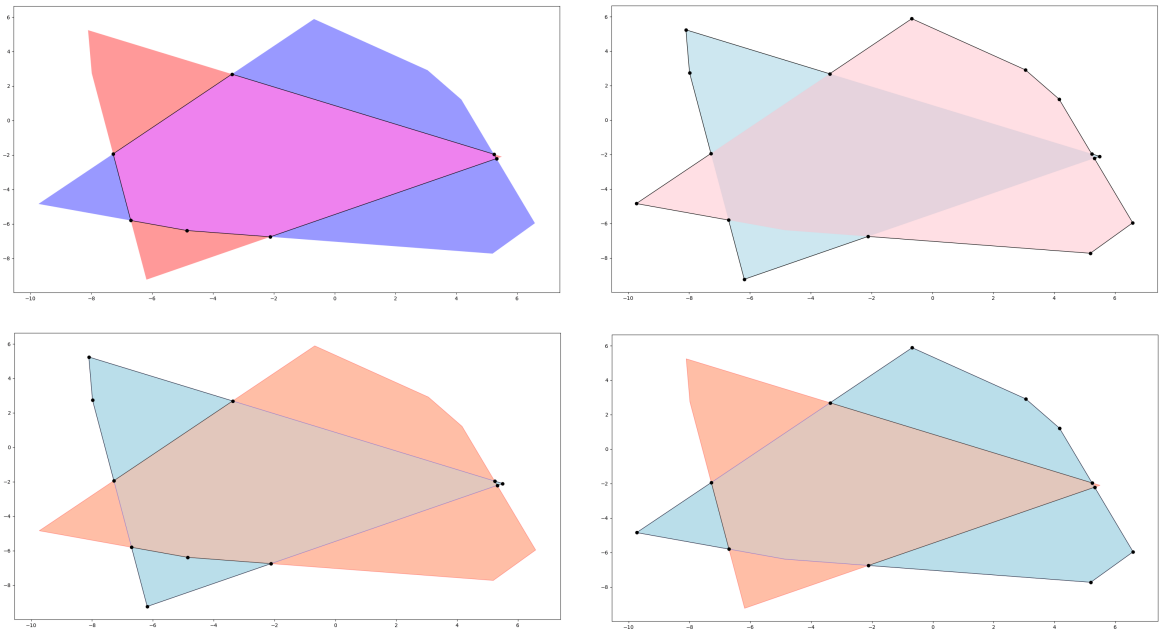
```
class PomTocka:
    def __init__(self, tocka, polozaj):
```

```
        self.tocka = tocka
        self.polozaj = polozaj


class PomDuzina:
    def __init__(self, duzina, polozaj):
        self.duzina = duzina
        self.polozaj = polozaj


def bool_operacije(self, drugi_poligon, operacija):
    if operacija == -1:
        fragmenti_za_zadrzati_self = -1
        fragmenti_za_zadrzati_other = 1
        if self.orijentacija() == drugi_poligon.orijentacija():
            self.promijeni_orijentaciju()

    elif operacija == 0:
        fragmenti_za_zadrzati_self = -1
        fragmenti_za_zadrzati_other = -1
        if self.orijentacija() != drugi_poligon.orijentacija():
            self.promijeni_orijentaciju()

    elif operacija == 1:
        fragmenti_za_zadrzati_self = 1
        fragmenti_za_zadrzati_other = 1
        if self.orijentacija() != drugi_poligon.orijentacija():
            self.promijeni_orijentaciju()

    pom_tocke_p_1 = [
        PomTocka(i, i.pripada_poligonu(drugi_poligon))
        for i in self.vrhovi
    ]
    pom_tocke_p_2 = [
        PomTocka(i, i.pripada_poligonu(self))
        for i in drugi_poligon.vrhovi
    ]

    razvrstane_tocke_p_1 = pom_tocke_sjecista(pom_tocke_p_1,
                                              pom_tocke_p_2)
    razvrstane_tocke_p_2 = pom_tocke_sjecista(pom_tocke_p_2,
                                              pom_tocke_p_1)
```

```
    pom_tocke_p_1 = razvrstane_tocke_p_1
    pom_tocke_p_2 = razvrstane_tocke_p_2


    razvrstani_fragmenti = (
        tocke_u_fragmente(
            pom_tocke_p_1,
            drugi_poligon,
            fragmenti_za_zadrzati_self,
            operacija)

        + tocke_u_fragmente(
            pom_tocke_p_2,
            self,
            fragmenti_za_zadrzati_other,
            operacija))

    razvrstani_fragmenti = [i.duzina for i in razvrstani_fragmenti]

    povezani_fragmenti = povezi_fragmente(razvrstani_fragmenti)

    rjesenje = []

    for i in povezani_fragmenti:
        popis_razlika = []
        for j in i:
            popis_razlika.append(j.A)
        rjesenje.append(Poligon(popis_razlika))

    return rjesenje


def pom_tocke_sjecista(pom_tocke_p_1, pom_tocke_p_2):
    br_p_1 = len(pom_tocke_p_1)
    br_p_2 = len(pom_tocke_p_2)

    razvrstane_tocke = []
    for i in range(0, br_p_1):
        tocka_1_p_1 = pom_tocke_p_1[i % br_p_1]
        tocka_2_p_1 = pom_tocke_p_1[(i + 1) % br_p_1]
        stranica_p_1 = Duzina(tocka_1_p_1.tocka, tocka_2_p_1.tocka)

        pom_lista_tocaka = [tocka_1_p_1, tocka_2_p_1]
```

```
        for j in range(0, br_p_2):
            tocka_1_p_2 = pom_tocke_p_2[j % br_p_2]
            tocka_2_p_2 = pom_tocke_p_2[(j + 1) % br_p_2]
            stranica_p_2 = Duzina(tocka_1_p_2.tocka, tocka_2_p_2.tocka)

            sjec = stranica_p_1.sjeciste(stranica_p_2)

            if not sjec.prazna():
                pom_lista_tocaka.append(PomTocka(sjec, 0))

        pom_lista_tocaka = \
            sorted(pom_lista_tocaka,
                   key=lambda z:
                   (z.tocka.x - pom_lista_tocaka[0].tocka.x)**2
                   + (z.tocka.y - pom_lista_tocaka[0].tocka.y)**2)
        pom_lista_tocaka.pop(-1)

        for pom_tocka in pom_lista_tocaka:
            razvrstane_tocke.append(pom_tocka)

    return razvrstane_tocke


def tocke_u_fragmente(pom_tocke_p_1, p_2, trazeni_polozaj, operacija):
    fragmenti = []
    br_p_1 = len(pom_tocke_p_1)
    for i in range(br_p_1):
        pom_tocka_1 = pom_tocke_p_1[i % br_p_1]
        pom_tocka_2 = pom_tocke_p_1[(i + 1) % br_p_1]
        if pom_tocka_1.tocka != pom_tocka_2.tocka:
            duzina = Duzina(pom_tocka_1.tocka, pom_tocka_2.tocka)
            if pom_tocka_1.polozaj == -1 or pom_tocka_2.polozaj == -1:
                polozaj = -1
            elif pom_tocka_1.polozaj == 1 or pom_tocka_2.polozaj == 1:
                polozaj = 1
            else:
                medutocka = Tocka(
                    (pom_tocka_1.tocka.x + pom_tocka_2.tocka.x) / 2,
                    (pom_tocka_1.tocka.y + pom_tocka_2.tocka.y) / 2)
                polozaj = medutocka.pripada_poligonu(p_2)

            nova_duzina = PomDuzina(duzina, polozaj)
```

```
            if (nova_duzina not in fragmenti
                    and polozaj == trazeni_polozaj
                    or (polozaj == 0
                        and operacija != -1
                        and nova_duzina not in fragmenti)):
                fragmenti.append(nova_duzina)
    return fragmenti


def povezi_fragmente(razvrstani_fragmenti):
    povezani_fragmenti = []
    while len(razvrstani_fragmenti) > 0:

        novi_poligon = []
        novi_poligon.append(razvrstani_fragmenti[0])

        razvrstani_fragmenti.pop(razvrstani_fragmenti.index(novi_poligon[0]))

        i = 0
        while i < len(razvrstani_fragmenti):
            if ((abs(novi_poligon[-1].B.x
                     - razvrstani_fragmenti[i].A.x) < EPSILON)
                    and (abs(novi_poligon[-1].B.y
                             - razvrstani_fragmenti[i].A.y) < EPSILON)):
                novi_poligon.append(razvrstani_fragmenti[i])
                razvrstani_fragmenti.pop(i)
                i = -1
            i += 1

        povezani_fragmenti.append(novi_poligon)
    return povezani_fragmenti
```



Slika 27: Implementacija Avraham-Knott algoritma za presjek (gore lijevo), uniju (gore desno), razliku prvog i drugog poligona (dole lijevo) i razliku drugog i prvog poligona (dole desno)

7. Triangulacija skupa točaka

Triangulacija skupa točaka S je povezivanje točaka iz skupa S tako da se dužine koje povezuju te točke nikada ne sijeku i tako da je konveksna ljuska skupa S podijeljena na skup trokuta. Poseban vrsta triangulacije koja daje "ljepše" trokute je Delauneyjeva triangulacija. Delauneyjeva triangulacija je najpoznatija ograničena triangulacija. Ograničene triangulacije su triangulacije koje postavljaju dodatno ograničenje na dobivene trokute triangulacije. Tako Delauneyjeve triangulacije zahtijevaju da nijedna opisana kružnica bilo kojeg dobivenog trokute ne sadrži vrh nijednog drugog trokuta. Takve triangulacije većinom dijele konveksnu ljusku danog skupa točaka na gotovo jednakostranične trokute. Prvi prikazani algoritam biti će pohlepni algoritam za izradu obične triangulacije, a nakon toga biti će prikazan Bowyer-Watsonov algoritam za dobivanje Delauneyjevih triangulacija. Oba algoritma su jednostavna i relativno neefikasna za rješavanje problema uzimajući u obzir da ne vraćaju triangulacije u optimalnom vremenu koje je $O(n \log n)$ za najgori slučaj. Oba algoritma do rješenja dolaze sa složenosti većom ili jednakom $O(n^2)$ [6].

7.1. Pohlepna triangulacija

Pohlepni algoritmi su algoritmi koji napreduju bez da poništavaju ono što su napravili u prijašnjem koraku. U slučaju triangulacija, kada dobijemo neku dužinu triangulacije, ona će sigurno ostati u skupu dužina koje će predstavljati triangulaciju. Kasnije ćemo vidjeti da za razliku od pohlepnog algoritma, Bowyer-Watsonov algoritam često izbacuje dužine koje je pronašao u prethodnom koraku. Algoritam je konceptualno jednostavan, no vremenski složen.

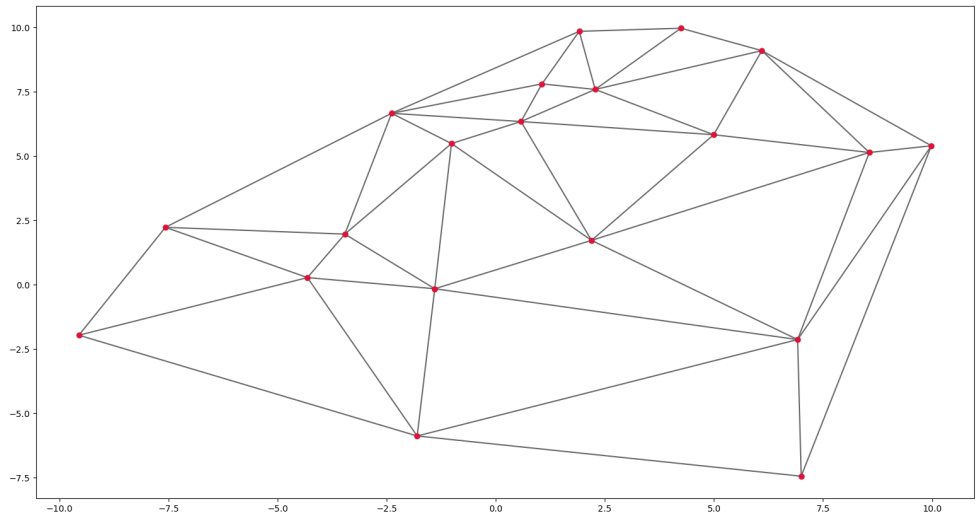
Algoritam počinje izradom svih mogućih dužina između točaka u zadanom skupu. Nakon toga, taj skup dužina potrebno je sortirati prema duljini. Zatim, uzimamo najmanju dužinu iz sortiranog skupa dužina i stavljamo je u triangulaciju ako se ona ne siječe s ostalim dužinama u triangulaciji. Najveća složenost koju algoritam može postići za najgori slučaj je $O(n^3)$ gdje je n broj točaka u početnom skupu točaka. Ta složenost se većinom postiže zbog potrebe za izradom svih mogućih dužina koje mogu nastati iz skupa točaka [6].

Implementacija pohlepne triangulacije

U implementaciji pohlepne triangulacije, prvi korak je izbacivanje duplikata iz liste objekata klase `Tocka`. Nakon toga pronalazimo sve moguće dužine između točaka u danom skupu i pohranjujemo ih kao potencijalne dužine triangulacije. Zatim sortiramo sve dužine prema njihovoj duljini. Na kraju, prolazimo kroz listu potencijalnih dužina i tražimo sjecište tih dužina sa dužinama koje su već u triangulaciji. Ako takvo sjecište ne postoji, tada se potencijalna dužina pohranjuje u listu dužina triangulacije. Ova implementacija ima složenost od $O(n^3)$.

```
def triangulacija(tocke):
    tocke = list(set(tocke))
    broj_tocaka = len(tocke)
```

```
potencijalne_duzine = []

for i in range(0, broj_tocaka):
    for j in range(i + 1, broj_tocaka):
        duzina = Duzina(tocke[i], tocke[j])
        potencijalne_duzine.append(duzina)

potencijalne_duzine.sort(key=lambda duzina: duzina.u_vektor().duljina())

broj_potencijalnih_duzina = len(potencijalne_duzine)
duzine_triangulacije = []

for i in range(0, broj_potencijalnih_duzina):
    potencijalna_duzina = potencijalne_duzine[i]
    postoji_sjeciste = False
    for j in range(0, len(duzine_triangulacije)):
        duzina_triangulacije = duzine_triangulacije[j]
        sjeciste = potencijalna_duzina.sjeciste(duzina_triangulacije)
        if (not sjeciste.prazna()
                and sjeciste != potencijalna_duzina.A
                and sjeciste != potencijalna_duzina.B):
            postoji_sjeciste = True

    if not postoji_sjeciste:
        duzine_triangulacije.append(potencijalna_duzina)

return duzine_triangulacije
```

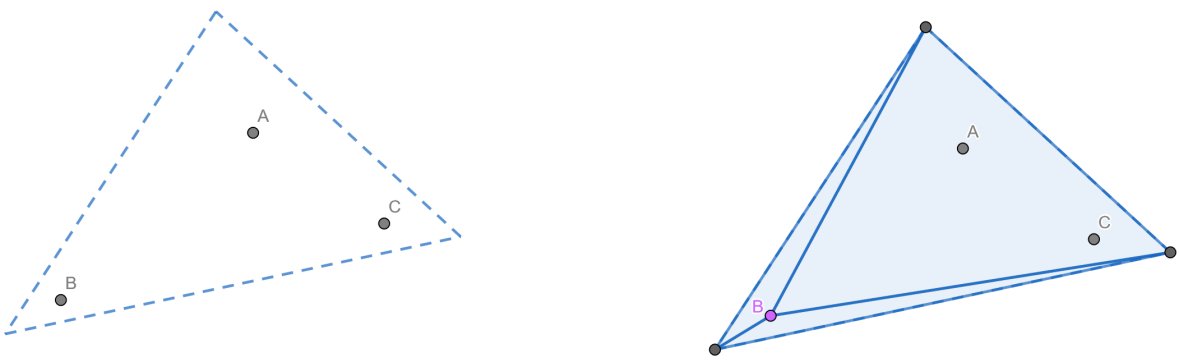


Slika 28: Implementacija pohlepne triangulacije

7.2. Bowyer-Watson algoritam

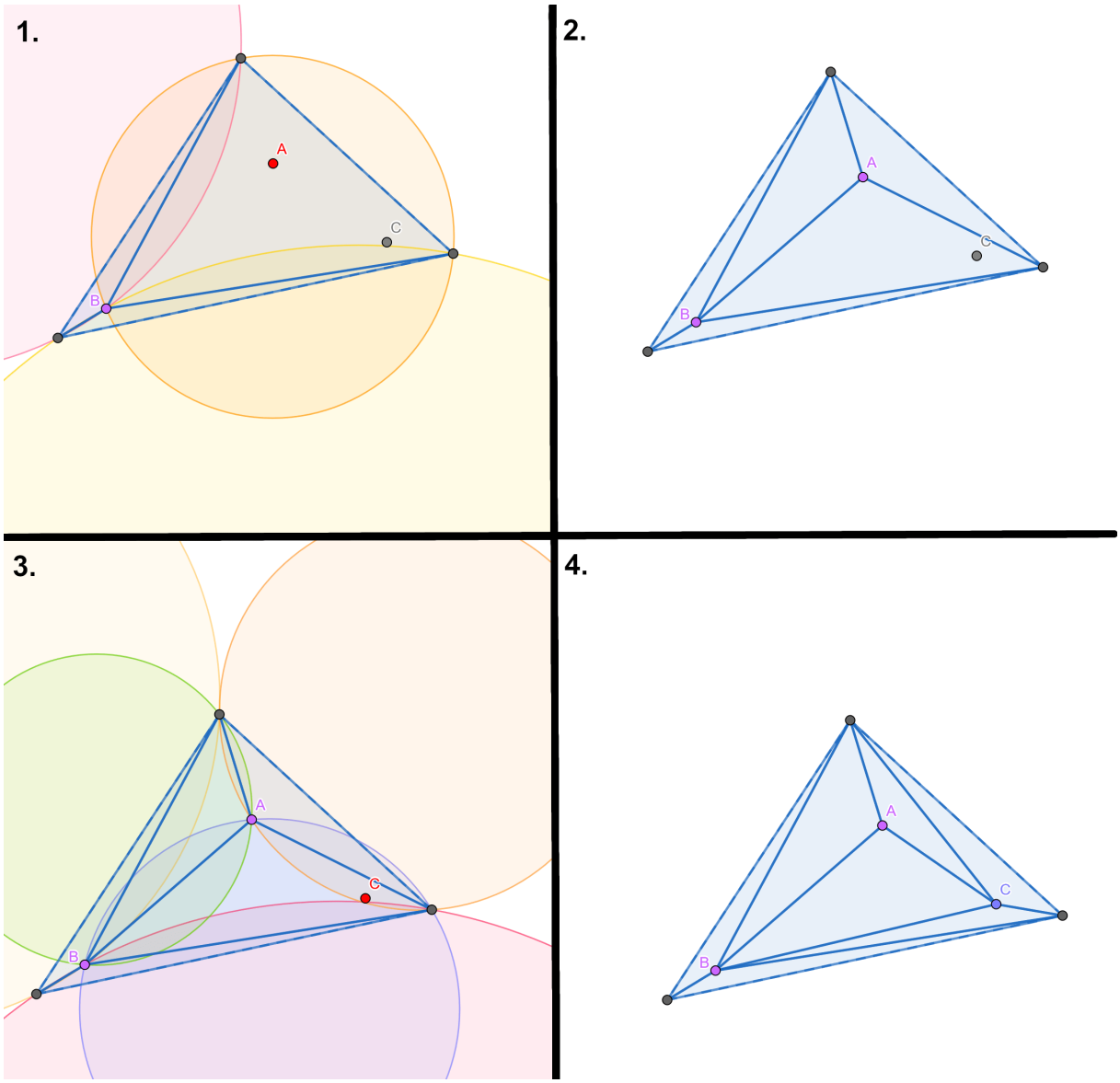
Bowyer-Watsonov algoritam je jednostavan algoritam za pronalaženje Delauneyjeve triangulacije za dani skup točaka. Bowyer-Watsonov algoritam se temelji na danom ograničenju za triangulaciju: svaka opisana kružnica nekog trokuta triangulacije ne smije sadržavati vrhove drugih trokuta. Za svaku točku u skupu točaka provjeravamo nalazi li se ona u nekoj opisanoj kružnici već napravljenih trokuta. Ovisno o tome, određene trokute triangulacije izbacujemo i dodajemo nove dok nismo prošli kroz sve točke zadanog skupa točaka.

Prvi korak algoritma je pronalaženje "supertrokuta". "Supertrokut" je trokut koji obuhvaća sve točke zadanog skupa. On služi samo za započinjanje izvršavanja algoritma i na kraju se briše on i svi trokuti koji sadrže njegove stranice. Nakon što smo odredili "supertrokut", uzimamo neku točku iz zadanog skupa točaka i povezujemo ju sa sva tri vrha "supertrokuta". Tako dobivamo tri nova trokuta unutar "supertrokuta" koja pohranjujemo kao moguće trokute triangulacije.



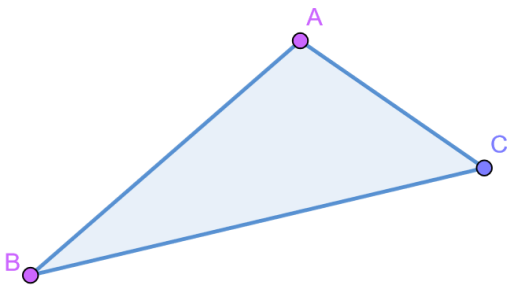
Slika 29: Izrada "supertrokuta" (lijevo) i početni trokuti triangulacije (desno)

Zatim, krećemo se dalje po zadanom skupu točaka i za svaku točku provjeravamo nalazi li se ona unutar opisane kružnice nekog od već napravljenih trokuta triangulacije. Ako da, izbacujemo trokut triangulacije u kojemu se ta točka nalazi. Nakon toga ostaje "šupljina" unutar koje se nalazi trenutna točka. Sve susjedne točke iz te "šupljine" povezujemo s trenutnom točkom tako da dobijemo nove trokute triangulacije. Ovaj postupak ponavljamo sve dok nismo prošli kroz sve točke zadanog skupa.



Slika 30: Provjera pripadnosti točke opisanim kružnicama postojećih trokuta triangulacije

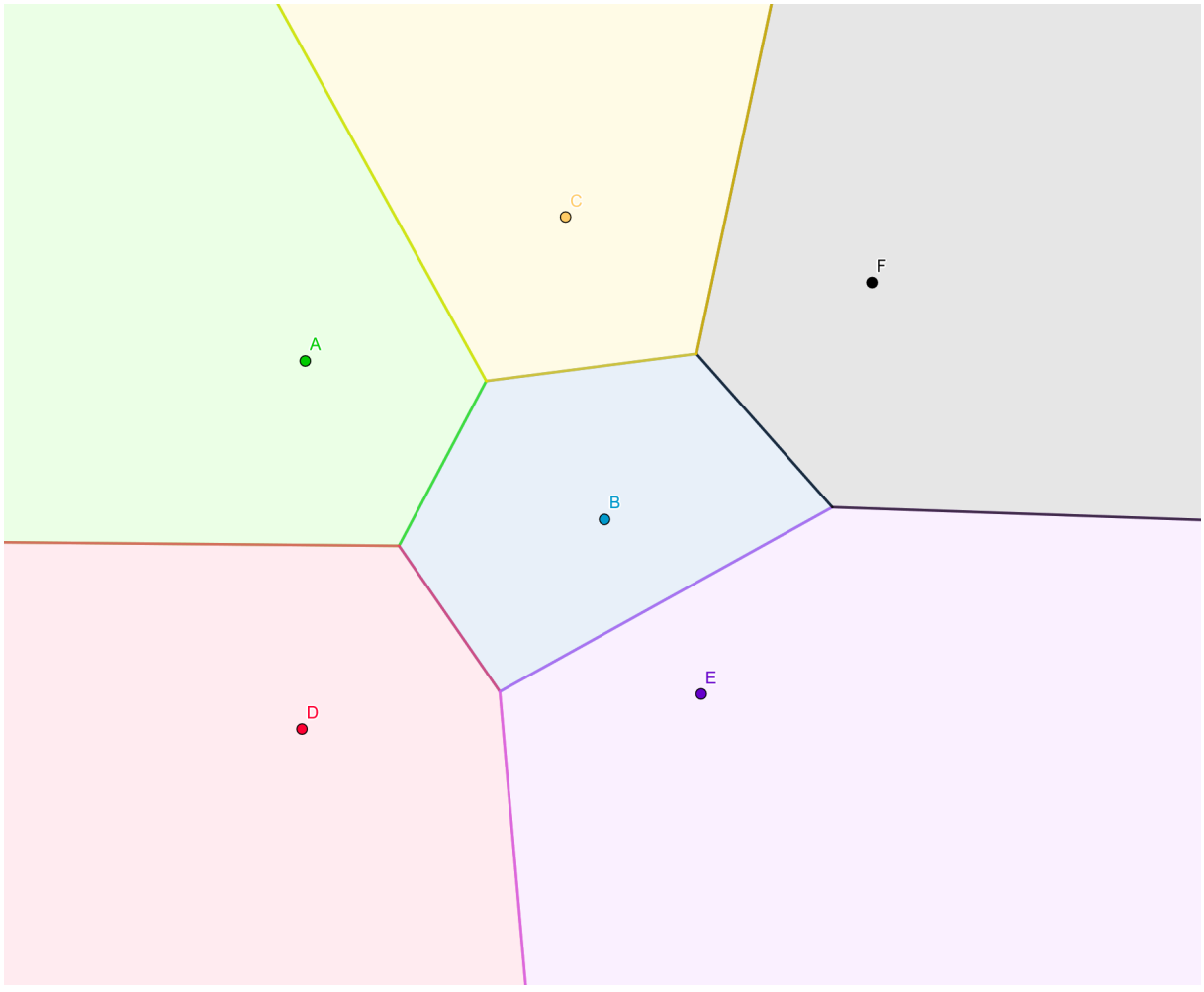
Na kraju, kada smo prošli kroz sve točke iz skupa točaka, potrebno je ukloniti "super-trokut" i sve ostale trokute koji su s njim povezani. Time dobivamo skup trokuta koji čine Delauneyjevu triangulaciju za zadani skup točaka. Složenost algoritma za najgori slučaj je $O(n^2)$ [15].



Slika 31: Rezultat Delauneyjeve triangulacije nakon izbacivanja "supertrokuta"

8. Voronojev dijagram

Voronojev dijagram predstavlja strukturu za rješavanje problema blizine točaka. Problemi blizine točaka se odnose na pronalaženje najbliže točke ili skupa najbližih točaka za neku danu točku iz zadanog skupa. Voronojev dijagram je rješenje temeljeno na *lokusima*. Lokusi su skupovi točaka koji zadovoljavaju neko svojstvo. U ovom slučaju, lokus predstavlja skup točaka koje su najbliže danoj točki. Tako će za zadani skup svaka točka iz tog skupa imati svoj lokus točaka koje su mu najbliže. Lokus ćemo nazvati Voronojevom ćelijom. Skup svih Voronojevih ćelija čini Voronojev dijagram. Svaki rub Vornojeve ćelije je simetrala dvije točke iz zadanog skupa točaka koja prostor dijeli na dvije poluravnine. Formalnije, Voronojevu ćeliju možemo definirati kao presjek poluravnina koje zadana točka čini sa simetralama svih ostalih točaka u zadanom skupu. Skup takvih presjeka poluravnina za točke iz zadanog skupa čini Voronojev dijagram. Slika [32] prikazuje Voronojeve ćelije (svaka u svojoj boji) unutar Voronojevog dijagrama za skup od 6 točaka [6].



Slika 32: Primjer Voronojevog dijagrama gdje je svaka Voronojeva ćelija obojana drugom bojom

Voronojev dijagram ima važna svojstva koja ga čine korisnim i daju nam uvid u moguće načine njegove izrade. Prvo svojstvo je da će svaka najbliža susjedna točka neke zadane

točke imati zajednički rub Voronojeve ćelije sa zadanom točkom. Zbog toga znamo da su najbliže susjedne točke neke točke iz skupa točaka one čije simetrale čine rubove Voronojeve ćelije odabrane točke. Osim toga, zanimljivo svojstvo je da će neke Vornojeve ćelije biti neograničene, odnosno imati će beskonačno veliku površinu. Kao što se vidi na slici [32], jedina ćelije koja je ograničena je ćelija točke B. Sve ostale su neograničene. Neograničene ćelije će uvijek biti ćelije onih točaka koje pripadaju konveksnoj ljusci zadanog skupa točaka. Na kraju, Voronojev dijagram je samo dual Delauneyjeve triangulacije. To znači da izvođenjem Delaunejeve triangulacije možemo dobiti Voronojev dijagram i obratno. Ovo svojstvo daje još jedan moguć način pronalaženja Voronojevog dijagrama. To znači da su trenutne dvije opcije za izvođenje Voronojevog dijagrama ili izravno traženje dijagrama ili njegovo izvođenje iz Delauneyjeve triangulacije za taj skup točaka. Prvi algoritam je naivni pristup traženja simetrala između točaka i njihovih sjecišta, a nakon toga će biti objašnjen algoritam na principu "podijeli pa vladaj". Na kraju, biti će prikazan jedan od postupaka dobivanja Voronojevog dijagrama iz Delauneyjeve triangulacije [6].

8.1. Naivni algoritam

Naivna metoda je daleko od efikasne, ali je intuitivna i lakša za implementirati od nekih optimalnijih. Naivni pristup se temelji na traženju simetrala koje će činiti rubove Voronojevih ćelija. Odabiremo neku točku iz početnog skupa i tražimo njenu simetralu sa svim ostalim točkama. Svaki put kada pronađemo simetralu, tražimo njeno sjecište sa svim ostalim pronađenim simetralama. Kako bi otkrili rub ćelije, odbacujemo sve dijelove simetrala koji se nalaze s one strane sjecišta na kojemu se ne nalazi trenutna točka. Ovo ponavljamo dok nismo prošli kroz sve točke u skupu. Možemo zaključiti da će nam za traženje simetrala trebati barem algoritam sa složenosti od $O(n^2)$, no dodatno, složenost može skočiti na $O(n^3)$ zbog potrebe za traženjem sjecišta između pronađenih simetrala [6].

Implementacija naivnog algoritma

Prije samog prikaza, potrebno je napomenuti glavno ograničenje ove implementacije. Algoritam ne radi konkretno sa simetralama prikazanim kao pravcima. Svaka simetrala je zapravo dužina sa vrlo dalekim krajnjim točkama. Zbog toga, izrada Voronojevog dijagrama je ograničena na kvadrat čiji vrhovi su točke $T_1(-100000000, -100000000)$, $T_2(100000000, -100000000)$, $T_3(100000000, 100000000)$ i $T_4(-100000000, 100000000)$. Za pohranjivanje Voronojevih ćelija potrebna je posebna klasa koja u sebi sadrži točku i poligon koji ju okružuje (rub lokusa najbližih točaka).

```
class VoronoiCelija:
    def __init__(self, tocka, poligon):
        self.tocka = tocka
        self.poligon = poligon
```

Prvo algoritam izbacuje duplikatne točke iz danog skupa. Zatim se provjerava jesu li dane bar dvije točke. Ako nisu, javlja se greška. Ako su dane točno dvije točke, izrađuje se njihova simetrala. Kako bi našli poligone koji okružuju zadane dvije točke, samo poveuzjemo simetralu sa graničnim točkama koje su joj s lijeve i desne strane. Granične točke su vrlo udaljene točke koje omeđuju cijeli Voronojev dijagram u ovoj implementaciji.

Ako je dan skup s više od dvije točke, pokreće se glavni algoritam. Prvo iteriramo kroz sve točke. Za svaku točku, sortiramo sve ostale prema udaljenosti od trenutne. Zatim, krećemo tražiti simetrale između trenutne simetrale i ostalih točaka. Ukoliko smo našli prvu simetralu, odmah ju pohranjujemo kao jedan od rubova Voronojeve ćelije. Kako smo sortirali točke prema udaljenosti, znamo da će prva simetrala biti rub Voronojeve ćelije jer je prva točka koju provjeravamo najbliža točka trenutnoj točki. Za sve ostale točke, tražimo simetrale, njihova sjecišta sa pronađenim rubovima ćelije i odbacujemo suvišne dijelove simetrala. Kada pronađemo sjecište između neke pronađene simetrale i trenutne simetrale, dijelimo obje simetrale na lijevi i desni dio. Sa svake simetrale ćemo odbaciti jedan od ta dva dijela, osim ako se simetrale ne sijeku u krajnjim točkama. Kako bi odredili koji dio postojećeg ruba ćemo zadržati, prvo računamo kut između trenutne točke i obje strane postojećeg ruba u smjeru od krajnje točke prema sjecištu. Zadržati ćemo onu stranu koja ima veći kut sa trenutnom točkom. Kako bi odredili točan dio trenutne simetrale, gledamo s koje strane postojeće simetrale se nalazi trenutna točka. Točna polovica trenutne simetrale biti će ona koja se nalazi s iste strane kao i trenutna točka. Kada smo pronašli sve simetrale koje čine rubove Voronojeve ćelije, uklanjamo suvišne dužine.

```
def ukloni_iste_tocke(skup_tocaka):
    pom_lista = []
    for tocka1 in range(0, len(skup_tocaka)):
        duplikat = False
        for tocka2 in range(tocka1 + 1, len(skup_tocaka)):
            if skup_tocaka[tocka1] == skup_tocaka[tocka2]:
                duplikat = True

        if not duplikat:
            pom_lista.append(skup_tocaka[tocka1])

    return pom_lista


def uklanjanje_rubova(rubovi, za_ukloniti):
    for rub in za_ukloniti:
        for rub2 in rubovi:
            if rub == rub2:
                rubovi.remove(rub2)
    return rubovi
```

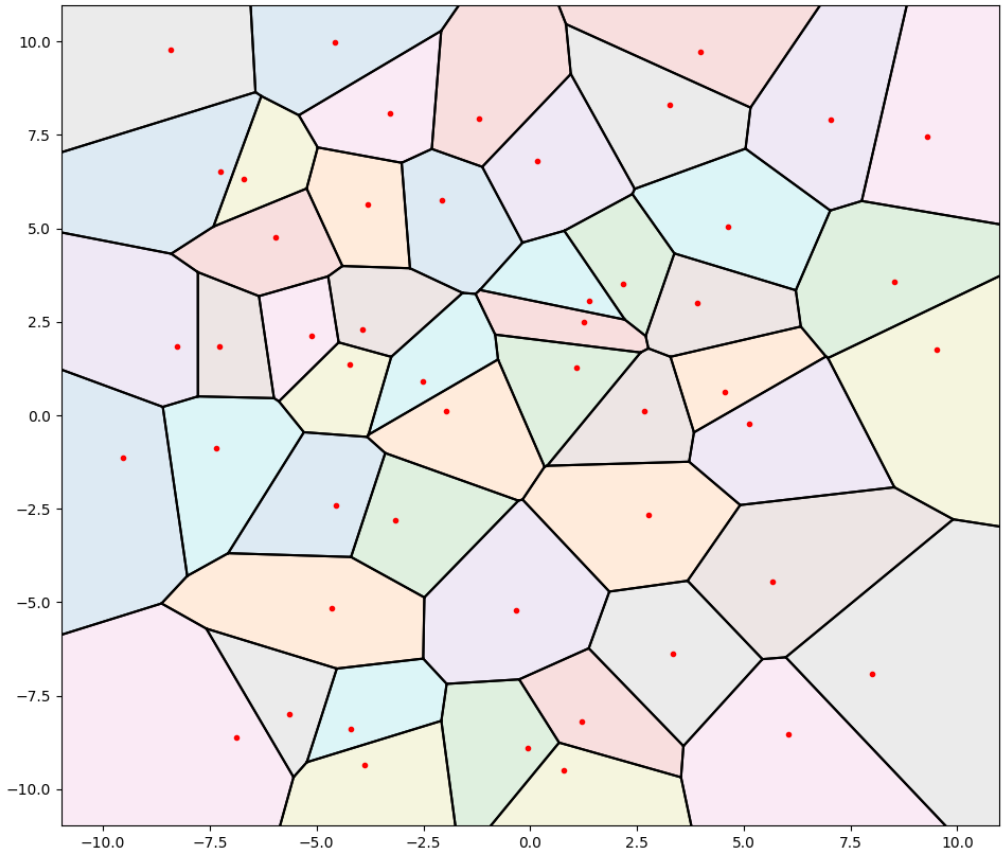
```
def voronoi_za_dvije_tocke(prva_tocka, druga_tocka):
    celije_dijagrama = []
    rub = Duzina(prva_tocka, druga_tocka).simetrala()

    daleka_tocka = 1000000000

    granicne_tocke = [Tocka(-daleka_tocka, -daleka_tocka),
                      Tocka(daleka_tocka, -daleka_tocka),
                      Tocka(daleka_tocka, daleka_tocka),
                      Tocka(-daleka_tocka, daleka_tocka),
                      ]

    tocka_je_na_lijevoj_strani = prva_tocka.lijevo_od(rub)

    vrhovi_prvog_poligona = []
    vrhovi_drugog_poligona = []

    for tocka in granicne_tocke:
        if tocka.lijevo_od(rub) == tocka_je_na_lijevoj_strani:
            vrhovi_prvog_poligona.append(tocka)
        else:
            vrhovi_drugog_poligona.append(tocka)

    vrhovi_prvog_poligona.append(rub.B)
    vrhovi_prvog_poligona.append(rub.A)
    vrhovi_drugog_poligona.append(rub.B)
    vrhovi_drugog_poligona.append(rub.A)

    vrhovi_prvog_poligona.sort(
        key=lambda vrh: (prva_tocka.polarni_kut(vrh),
                         Duzina(prva_tocka, vrh).u_vektor().duljina()))

    vrhovi_drugog_poligona.sort(
        key=lambda vrh: (druga_tocka.polarni_kut(vrh),
                         Duzina(druga_tocka, vrh).u_vektor().duljina()))

    prvi_poligon = Poligon(vrhovi_prvog_poligona)
    drugi_poligon = Poligon(vrhovi_drugog_poligona)
    celije_dijagrama.append(VoronoiCelija(prva_tocka, prvi_poligon))
    celije_dijagrama.append(VoronoiCelija(druga_tocka, drugi_poligon))

    return celije_dijagrama
```

```
def voronoi(tocke):
    tocke = list(set(tocke))

    if len(tocke) < 2:
        raise PremaloTocakaError("Premalo točaka za izradu dijagrama!")

    if len(tocke) == 2:
        return voronoi_za_dvije_tocke(tocke[0], tocke[1])

    celije_dijagrama = []

    tocke_pom = [i for i in tocke]

    for trenutna_tocka in tocke_pom:
        tocke.sort(key=lambda tocka: trenutna_tocka.udaljenost_od(tocka))

        tocke.pop(0)

        rubovi = []

        pronadeno_bar_jedno_sjeciste = False

        for tocka in tocke:
            trenutna_duzina = Duzina(trenutna_tocka, tocka)
            trenutna_simetrala = trenutna_duzina.simetrala()

            if len(rubovi) == 0:
                rubovi.append(trenutna_simetrala)

            else:
                za_ukloniti = []

                rubovi_pom = []

                postoji_sjeciste = False

                for postojeci_rub in rubovi:
                    sjeciste = trenutna_simetrala.sjeciste(postojeci_rub)

                    if not sjeciste.prazna():
                        postoji_sjeciste = True
                        pronadeno_bar_jedno_sjeciste = True
```

```
                if (postojeci_rub.A ==
                        sjeciste or postojeci_rub.B == sjeciste):
                    tocan_dio_postojeceg_ruba = postojeci_rub

                else:
                    za_ukloniti.append(postojeci_rub)

                    lijeva_polovica_postojeceg_ruba = Duzina(
                        postojeci_rub.A, sjeciste)
                    desna_polovica_postojeceg_ruba = Duzina(
                        postojeci_rub.B, sjeciste)

                    tocan_dio_postojeceg_ruba = \
                        lijeva_polovica_postojeceg_ruba

                    vektor_do_trenutne_točke = Duzina(
                        sjeciste, trenutna_tocka).u_vektor()

                    kut_između_lpolpostrub_i_vdtt = \
                        lijeva_polovica_postojeceg_ruba.u_vektor()\
                            .kut_izmedu_vektora(vektor_do_trenutne_točke)
                    kut_između_dpolpostrub_i_vdtt = \
                        desna_polovica_postojeceg_ruba.u_vektor()\
                            .kut_izmedu_vektora(vektor_do_trenutne_točke)

                    if (kut_između_lpolpostrub_i_vdtt <
                            kut_između_dpolpostrub_i_vdtt):
                        tocan_dio_postojeceg_ruba = \
                            desna_polovica_postojeceg_ruba

                    rubovi_pom.append(tocan_dio_postojeceg_ruba)

                if (trenutna_simetrala.A ==
                        sjeciste or trenutna_simetrala.B == sjeciste):
                    tocan_dio_trenutne_simetrale = trenutna_simetrala

                else:
                    lijeva_polovica_trenutne_simetrale = Duzina(
                        sjeciste, trenutna_simetrala.A)
                    desna_polovica_trenutna_simetrale = Duzina(
                        sjeciste, trenutna_simetrala.B)

                    trenutna_tocka_je_na_lijevoj_strani = \
```

```
                        trenutna_tocka.lijevo_od(
                            tocan_dio_postojeceg_ruba)

                    lpoltrensim_pripada_poligonu = \
                        trenutna_simetrala.A.lijevo_od(
                            tocan_dio_postojeceg_ruba)

                    if (lpoltrensim_pripada_poligonu ==
                            trenutna_tocka_je_na_lijevoj_strani):
                        tocan_dio_trenutne_simetrale = \
                            lijeva_polovica_trenutne_simetrale

                    else:
                        tocan_dio_trenutne_simetrale = \
                            desna_polovica_trenutna_simetrale
                    trenutna_simetrala = tocan_dio_trenutne_simetrale

                elif not pronadeno_bar_jedno_sjeciste:
                    rubovi_pom.append(trenutna_simetrala)

            rubovi = uklanjanje_rubova(rubovi, za_ukloniti)

            for rub in rubovi_pom:
                rubovi.append(rub)

            if postoji_sjeciste:
                rubovi.append(trenutna_simetrala)

    for rub_1 in rubovi:
        tocka_je_lijevo = trenutna_tocka.lijevo_od(rub_1)
        for rub_2 in rubovi:
            if not (rub_2.A == rub_1.A or rub_2.A == rub_1.B):
                tockaA_je_lijevo = rub_2.A.lijevo_od(rub_1)
                if tockaA_je_lijevo != tocka_je_lijevo:
                    za_ukloniti.append(rub_2)
            if not (rub_2.B == rub_1.A or rub_2.B == rub_1.B):
                tockaB_je_lijevo = rub_2.B.lijevo_od(rub_1)
                if tockaB_je_lijevo != tocka_je_lijevo:
                    za_ukloniti.append(rub_2)

    rubovi = uklanjanje_rubova(rubovi, za_ukloniti)

    vrhovi = [t.A for t in rubovi]
```

```
            vrhovi += [t.B for t in rubovi if t.B not in vrhovi]

            vrhovi = ukloni_iste_tocke(vrhovi)

            vrhovi.sort(
                key=lambda tocka:
                (trenutna_tocka.polarni_kut(tocka),
                 Duzina(trenutna_tocka, tocka).u_vektor().duljina()))

            voroni_celija = VoronoiCelija(trenutna_tocka, Poligon(vrhovi))

            celije_dijagrama.append(voroni_celija)

            tocke.insert(0, trenutna_tocka)

    return celije_dijagrama
```

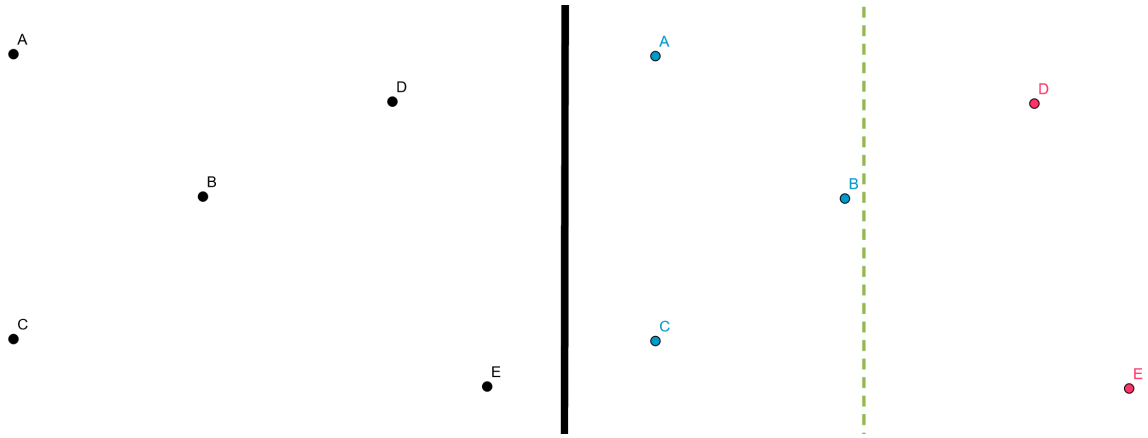


Slika 33: Prikaz implementacije naivnog algoritma za traženje Voronojevog dijagrama

8.2. "Podijeli pa vladaj" algoritam

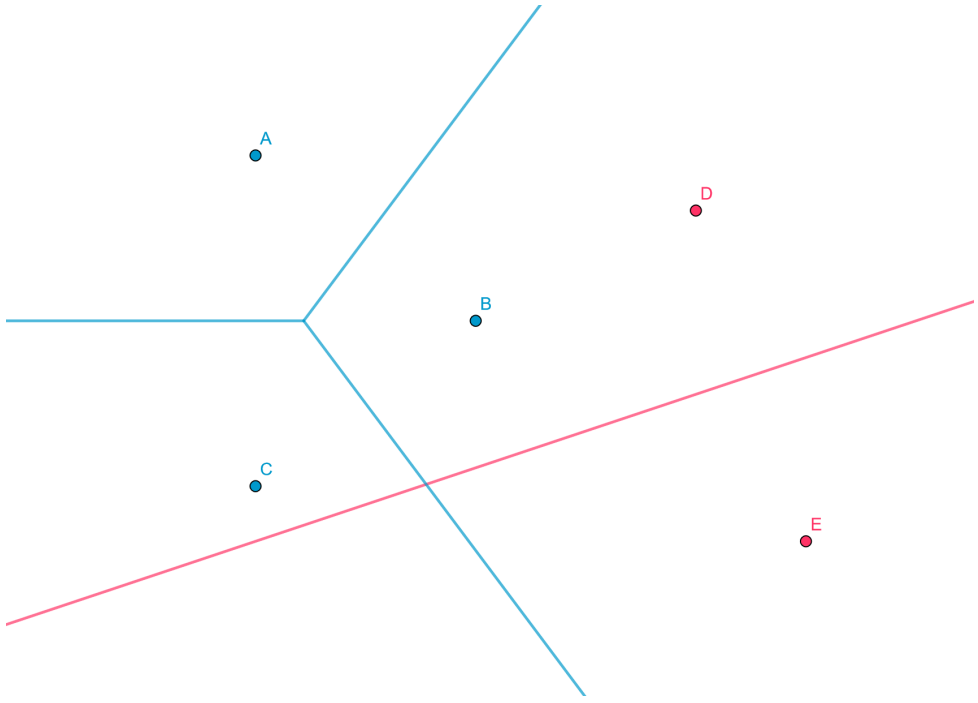
Temelj algoritma "podijeli pa vladaj" je izrada manjih Voronojevih dijagrama za manji broj točaka i njihovo spajanje u jedinstveni Voronojev dijagram. Spajanje se temelji na izradi monotonog lanca zajedničkih rubova ćelija prvog i drugog dijagrama. Lanac je skup povezanih dužina, a monotonost se odnosi na svojstvo tog lanca da će svaki pravac okomit na y-os sječi taj lanac u samo jednoj točci. Taj monotoni lanac možemo iskoristiti za povezivanje dvaju Voronojeva dijagrama u jedan [6].

Prvi korak je rekurzivno rastavljanje danog skupa točaka na manje skupove. Rastavljanje se provodi tako da se točke podijele prema prosječnoj vrijednosti x-osi. Sve točke čija je apscisa manja od prosjeka dijele se u lijevi skup L, a sve točke čija apscisa je veća od prosjeka dijele se u desni skup D. Skup se rastavlja sve dok ne dođemo do skupova od dvije ili tri točke.



Slika 34: Rastav početnog skupa na lijevi (plavi) i desni skup (crveno)

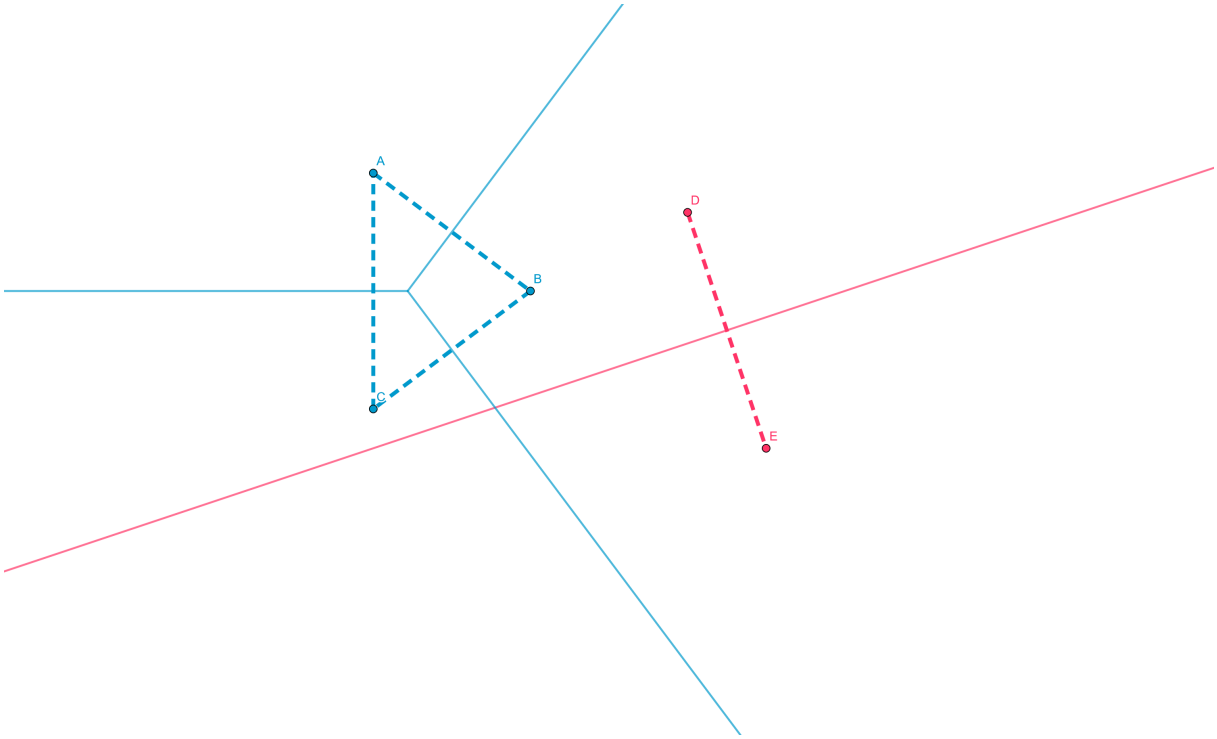
Zatim, stvaramo Voronojev dijagram za lijev i desni skup. Voronojev dijagram za skupove od dvije točke je trivijalan. Jedino što je potrebno je napraviti simetralu za te dvije točke. Voronojev dijagram za tri točke moguće je napraviti tako da se naprave simetrale za svake dvije točke iz tog skupa. Nakon toga traže se sjecišta tih simetrala i izbacuju se dijelovi simetrala koji ne pripadaju Voronojevom dijagramu.



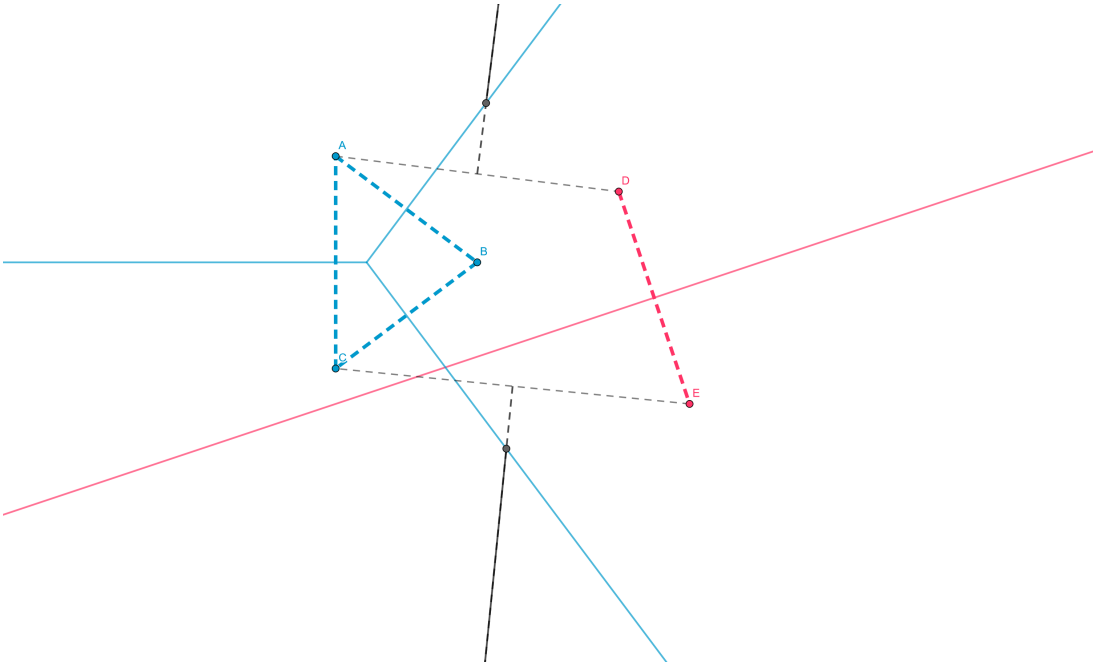
Slika 35: Izrada Voronojevog dijagrama za dvije i tri točke

Nakon toga, potrebno je spojiti dijagrame za lijevi i desni skup. Za to, potrebno je napraviti monotoni lanac između ta dva dijagrama. Prvo je potrebno pronaći konveksne ljuske lijevog i desnog skupa točaka. Za skupove od dvije i tri točke to je trivijalno, no teže je kada će se rekurzivno spajati veći skupovi točaka. Nakon toga, potrebno je naći potporne dužine tih dviju konveksnih ljuski. Potporne dužine su dužine koje povezuju lijevu i desnu konveksnu ljusku u konveksnu ljusku cjelokupnog skupa točaka koji trenutno promatramo. Zatim je potrebno pro-

naći simetrale tih dviju potpornih dužina. Te dvije simetrale biti će polupravci od beskonačnosti do njihovog sjecišta s prvim najbližim rubom Voronojevog dijagrama. Na kraju, sastavljati ćemo monotoni lanac od gornje simetrale sve dok ne dođemo do donje.



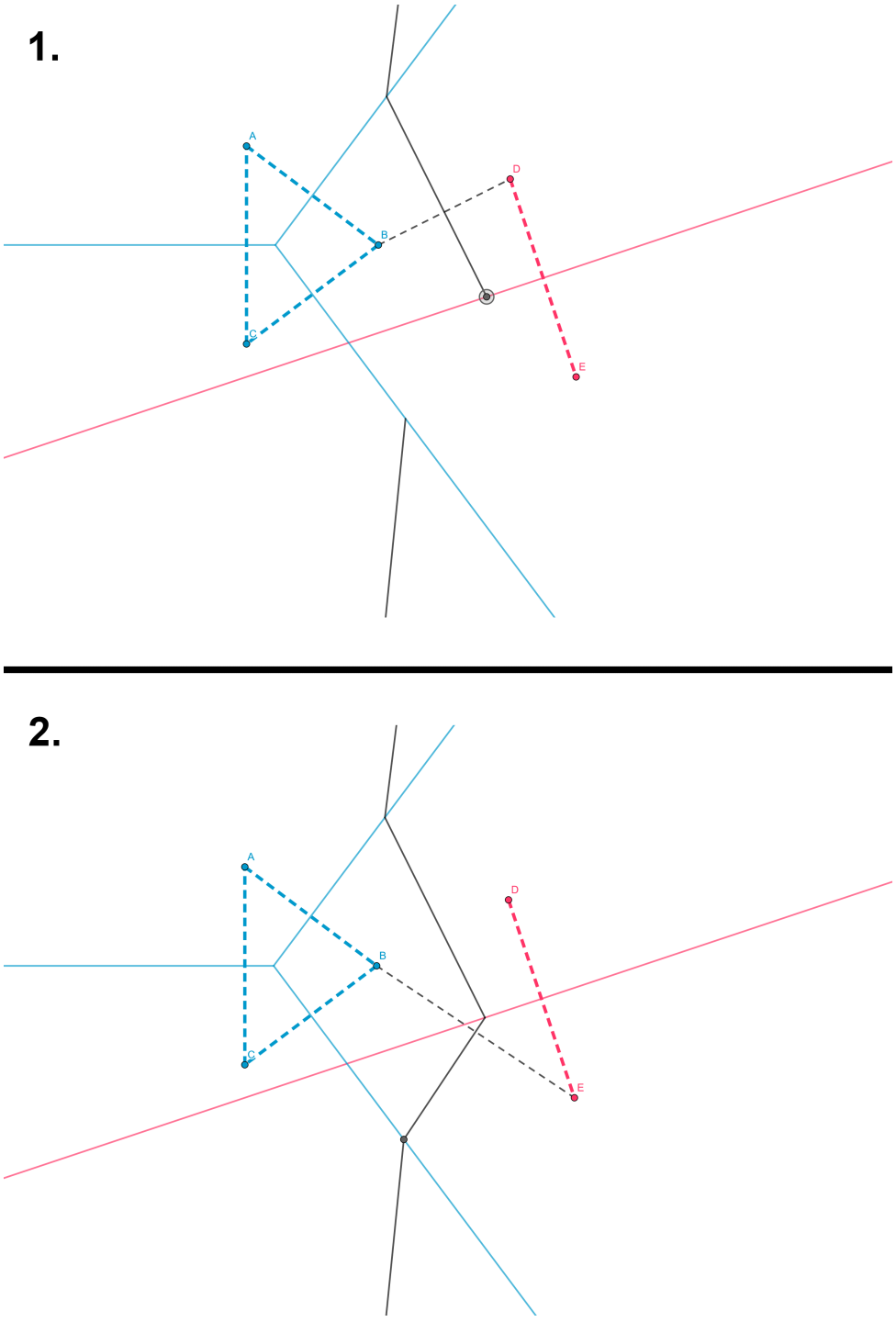
Slika 36: Prikaz konveksnih ljuski lijevog i desnog skupa točaka



Slika 37: Izrada simetrala na pomoćnim dužinama za izradu monotonog lanca

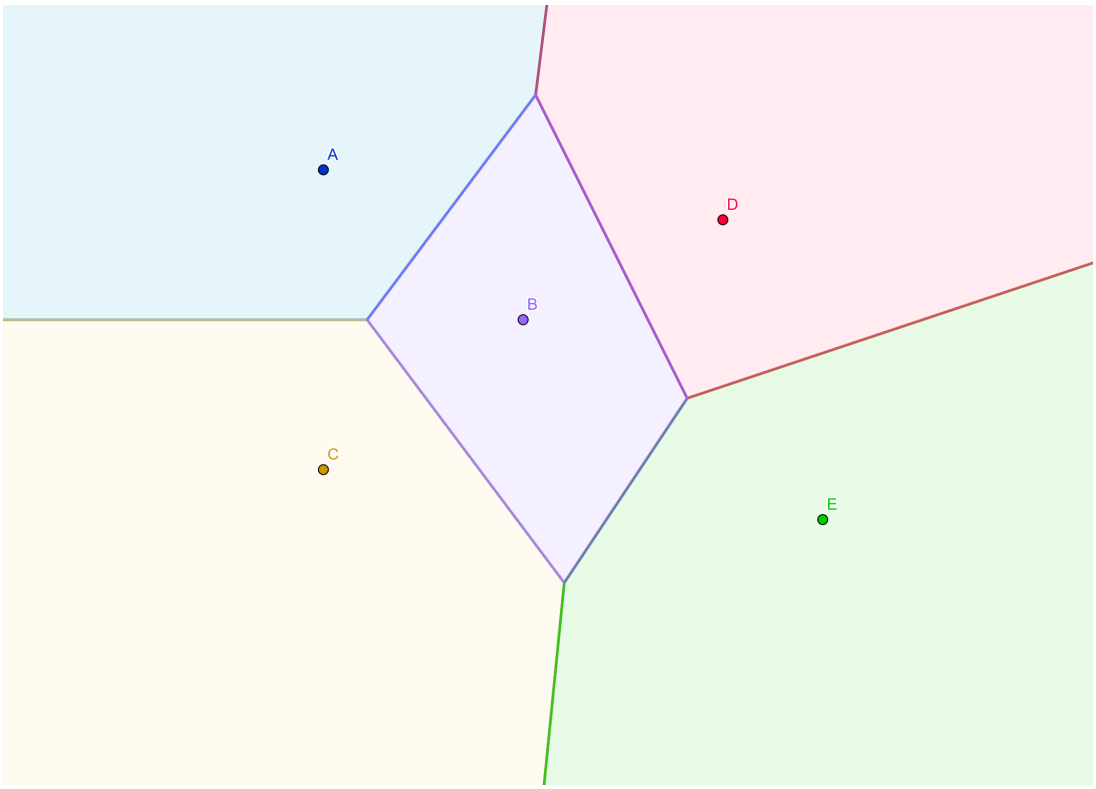
Za svaku novu dužinu koju ćemo dodati u monotoni lanac potrebno je prvo napraviti potpornu dužinu između dviju točki dvaju Voronojevih dijagrama. Prva točka će biti točka po-

ligona koji smo pogodili početnom simetralom, a druga će biti prva točka drugog Voronojevog dijagrama. Zatim, od sjecišta početne simetrale povlačimo simetralu na novostvorenu dužinu sve dok ne pogodimo novi rub nekog od dva Voronojeva dijagrama. Čim pogodimo rub, mijenjamo jednu od točaka potporne dužine kako bi povukli novu potpornu dužinu. Ovisno o tome koji Voronojev dijagram smo pogodili, mijenjamo točku potporne dužine u točku koju okružuje Voronojeva ćelija koju smo pogodili. Ovaj postupak se ponavlja sve dok ne dođemo do druge simetrale koju smo stvorili na početku [6].



Slika 38: Sastavljanje monotonog lanca

Na kraju, potrebno je ukloniti suvišne dijelove rubova Voronojevih ćelija. Za lijevi dijagram, suvišni dijelovi su svi koji se nalaze s desne strane monotonog lanca, a za desni dijagram, suvišni dijelovi su oni koji se nalaze s lijeve strane monotonog lanca. Cijeli postupak ponavljamo dok nismo spojili sve stvorene Voronojeve dijagrame u jedan. Tako dobivamo potpuni Voronojev dijagram za sve točke iz zadanog skupa sa složenosti od $0(n \log n)$.

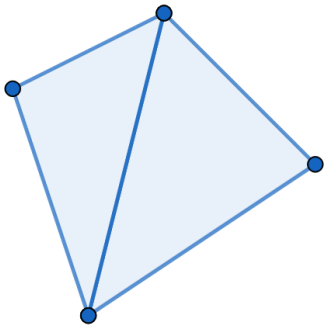


Slika 39: Gotov Voronojev dijagram nakon izbacivanja suvišnih dijelova rubova

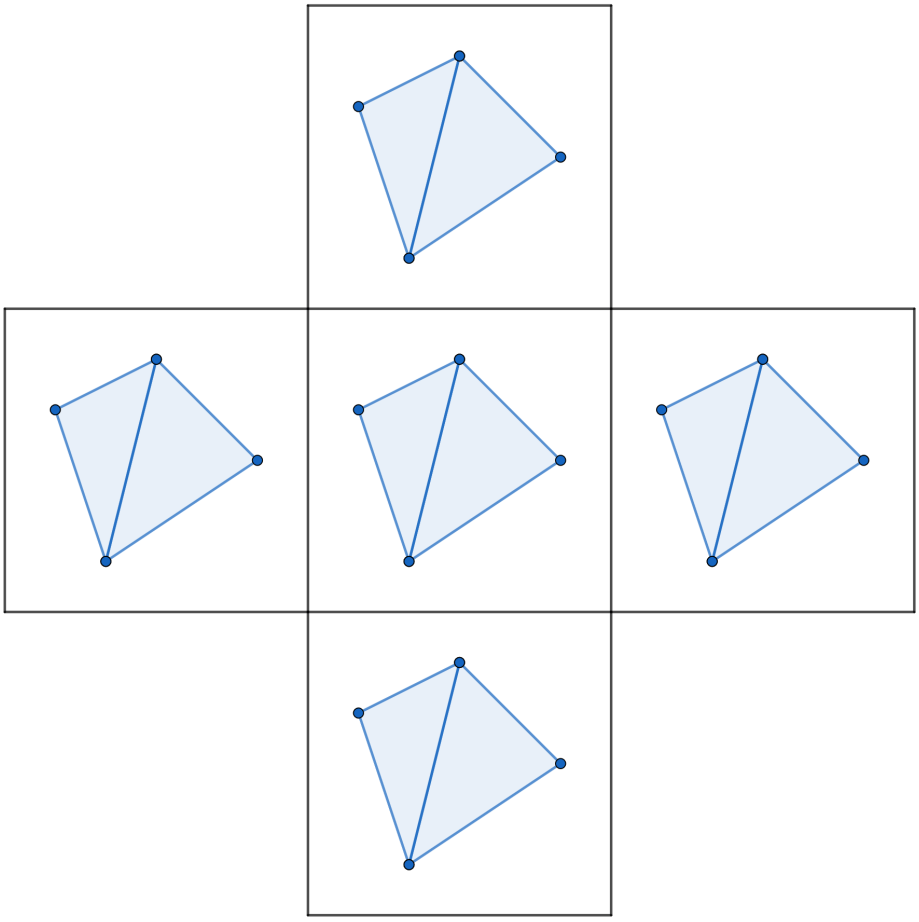
8.3. Izvođenje Voronojevog dijagrama iz Delauneyjeve triangulacije

Kao što je ranije rečeno, Voronojev dijagram je dual Delauneyjeve triangulacije. To znači da je moguće osmisliti algoritam koji će za danu Delauneyjevu triangulaciju moći izvesti Voronojev dijagram i obratno. Jedan od načina dobivanja Voronojevog dijagrama iz Delauneyjeve traingulacije je korištenje opisanih kružnica svakog trokuta triangulacije. Svako središte tih opisanih kružnica predstavlja jedan vrh neke Voronojeve ćelije Voronojevog dijagrama. Povezivanjem tih vrhova u određenom redoslijedu dobivamo Voronojev dijagram.

Za početak potrebno je pronaći početnu triangulaciju skupa točaka. Zatim, potrebno je omeđiti triangulaciju kvadratom. Omeđivanjem triangulacije određujemo prostor za koji ćemo izraditi Voronojev dijagram. Sljedeće, potrebno je preslikati triangulaciju iznad, ispod, lijevo i desno od početne triangulacije. To radimo kako bi kasnije dobili sve vrhove Voronojevog dijagrama.

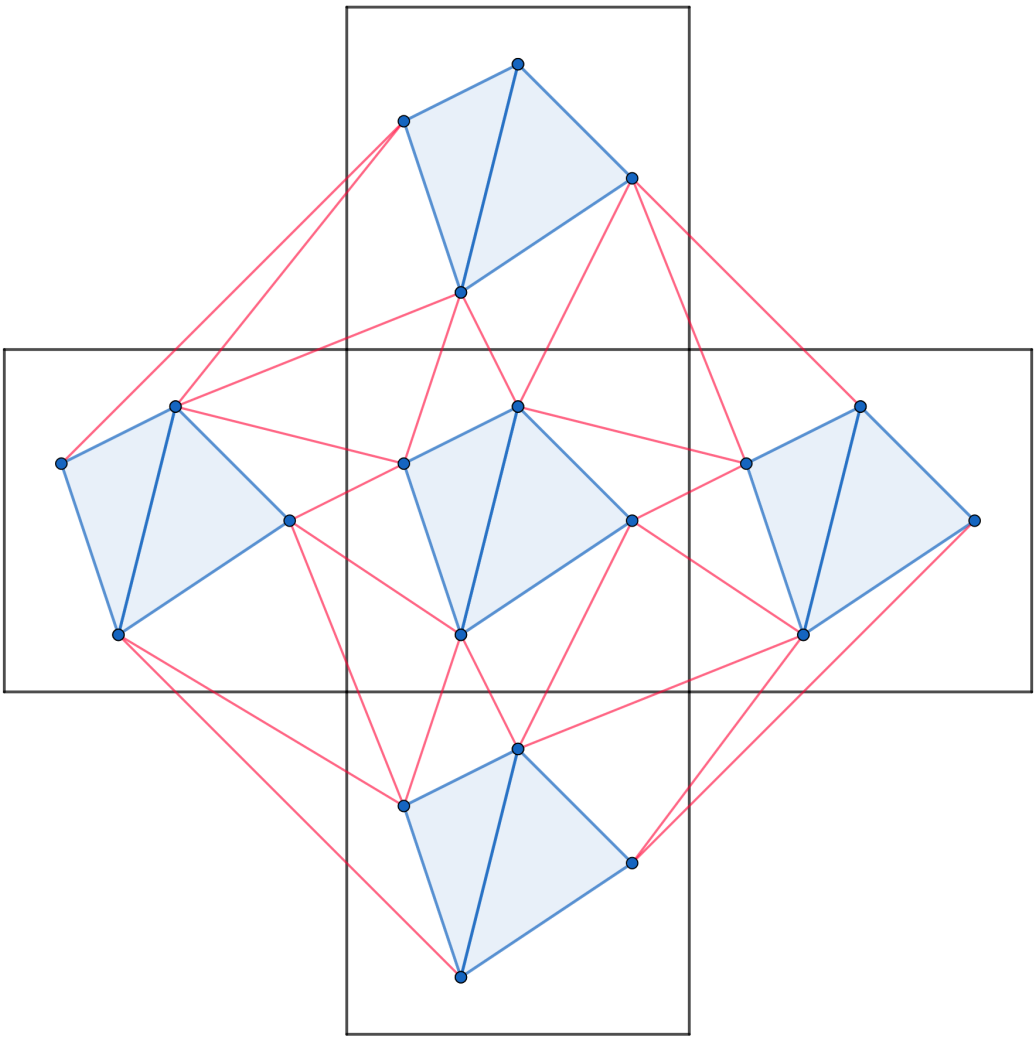


Slika 40: Početna Delauneyjeva triangulacija



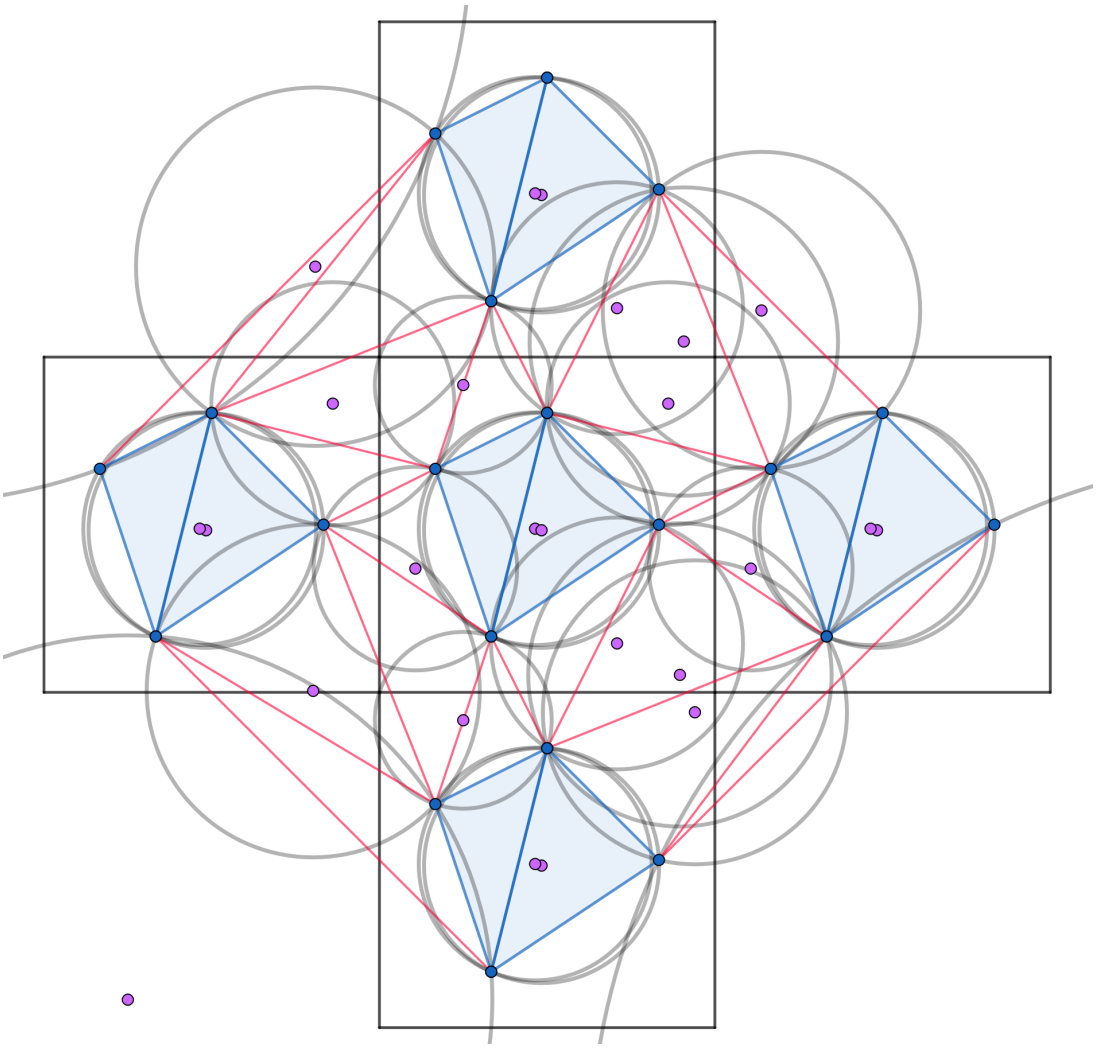
Slika 41: Početna triangulacija sa preslikanim triangulacijama iznad, ispod, lijevo i desno

Nakon toga, potrebno je proširiti triangulaciju između početne i preslikanih triangulacija.



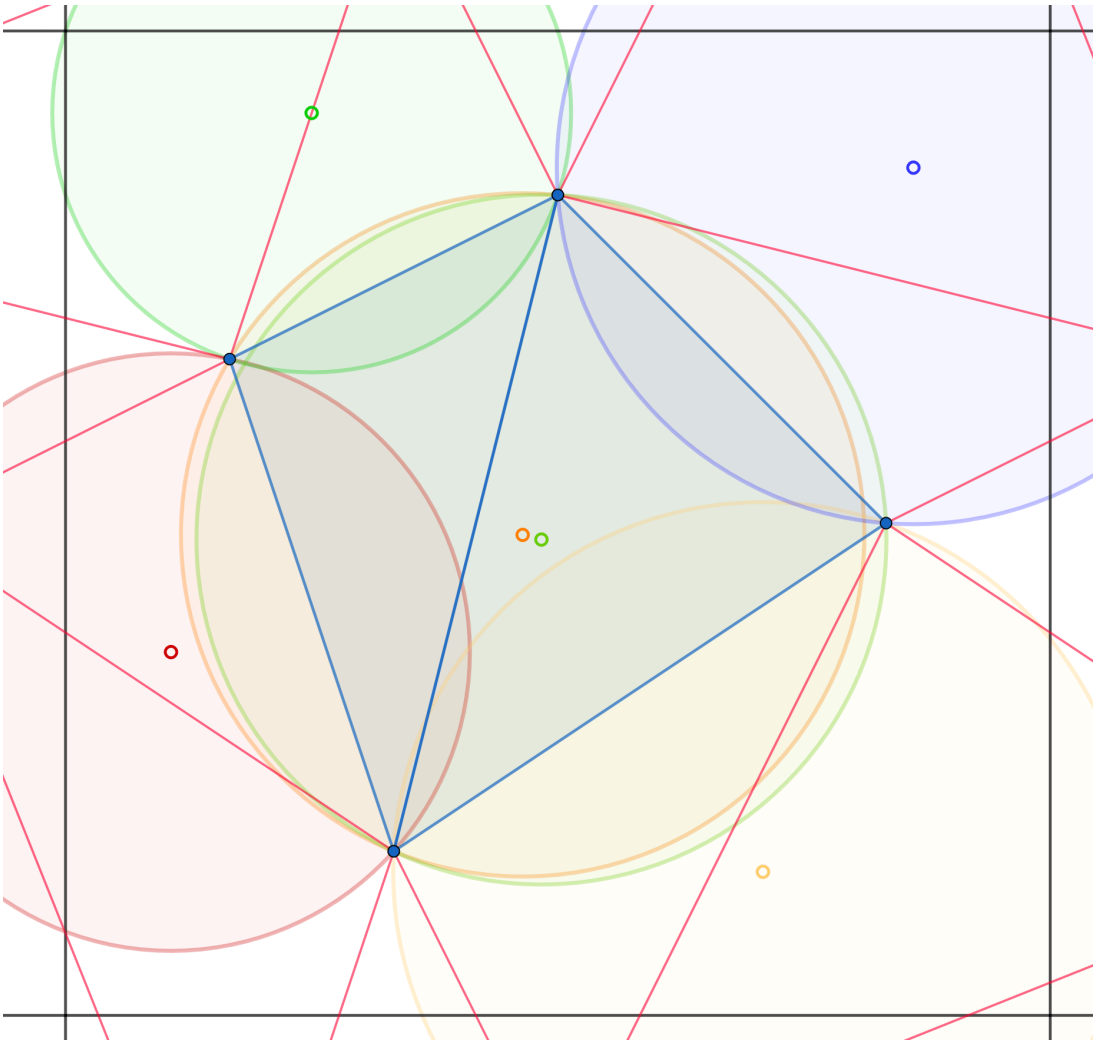
Slika 42: Proširena triangulacija

Nakon izrađene proširene triangulacije možemo krenuti tražiti potencijalne vrhove Voronojevog dijagrama. Za svaki trokut proširene triangulacije tražimo njegovu opisanu kružnicu. Svako središte tih kružnica predstavlja potencijalan vrh Voronojevog dijagrama. To je zato što se svako središte kružnice nalazi na jednakoj udaljenosti od triju vrhova trokuta, a jedno od zanimljivih svojstava Voronojevog dijagrama je da je svaki vrh Voronojevog dijagrama sjecište točno triju rubova dijagrama.



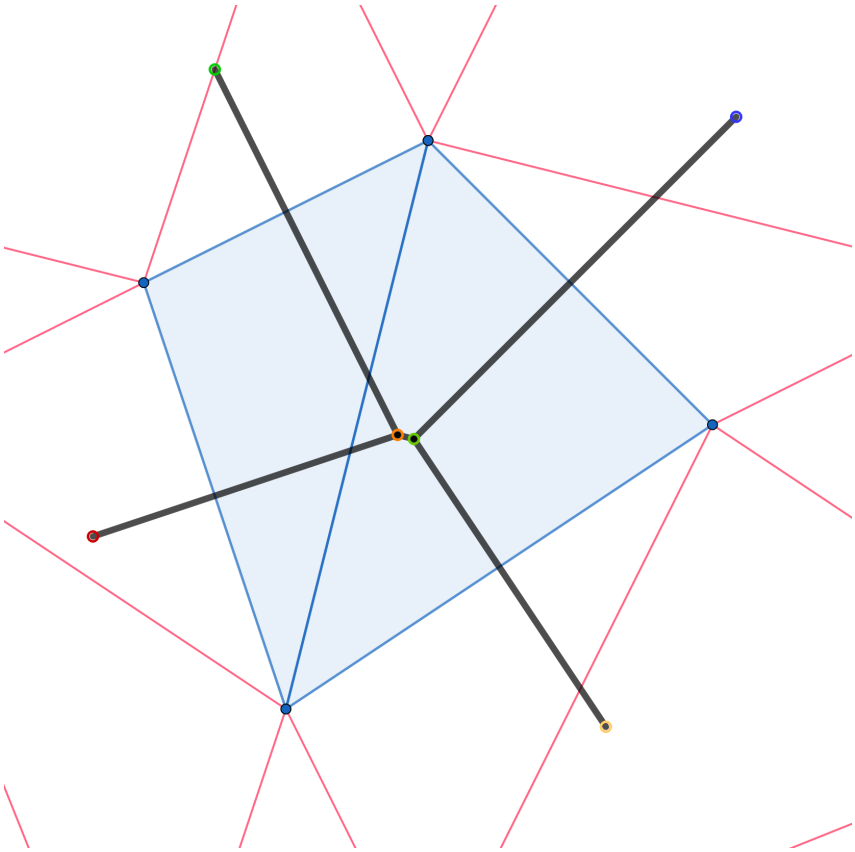
Slika 43: Kružnice sa središtima za sve dobivene trokute proširene triangulacije

Kako nas zanima samo Voronojev dijagram omeđene triangulacije, odbacujemo sva središta i sve kružnice koje se nalaze izvan početnog omeđenog prostora.

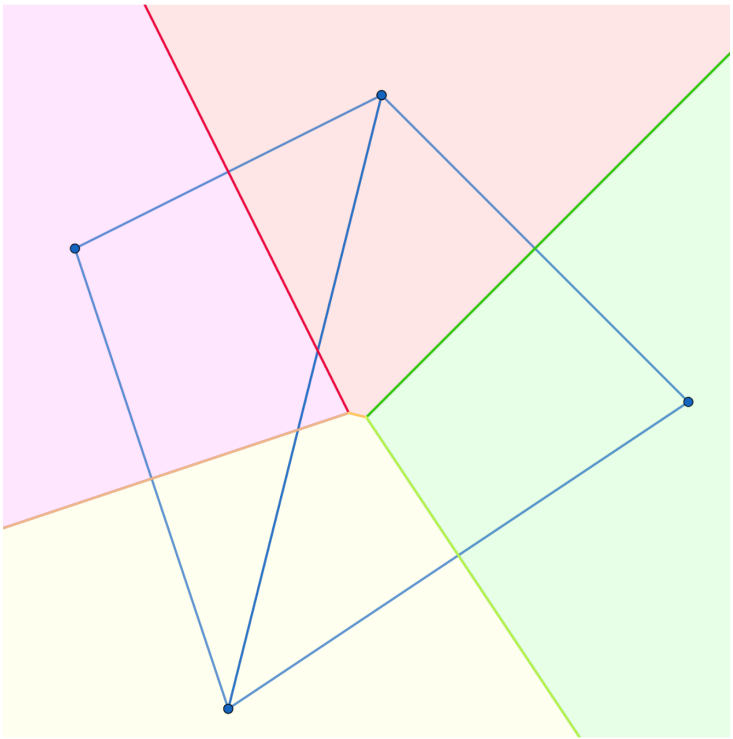


Slika 44: Odabrane kružnice čija su središta unutar omeđenog prostora

Za neku točku triangulacije, krećemo se po trokutima čiji je ona vrh u smjeru suprotnom od kazaljke na satu i povezujemo središta njihovih opisanih kružnica kako bi dobili Vornojevu ćeliju za tu točku. Ovo ponavljamo za sve točke početnog skupa točaka za koji tražimo Voronojev dijagram [16].



Slika 45: Izrađeni rubovi Voronojevog dijagrama



Slika 46: Izveden Voronojev dijagram

9. Zaključak

Računalna geometrija nudi razne alate za rješavanje geometrijskih problema. Kao što se moglo vidjeti iz rada, svaki problem ima nekoliko postojećih algoritama koji se mogu implementirati sa različitim razinama vremenske, ali i implementacijske složenosti. Neki od pristupa koji se često pojavljuju u algoritmima računalne geometrije su *Sweep Line* metoda, "podijeli pa vladaj" metoda, naivni pristup problemu i sortiranja i pretraživanja za olakšavanje rješavanja danog problema.

Osim toga, može se primijetiti kako efikasan algoritam za jedan problem, pomaže pri izgradnji efikasnih algoritama za drugi problem. Tako se recimo u provođenju Boolovih operacija nad poligonima može olakšati traženje sjecišta koristeći algoritme za efikasno traženje presjeka skupa dužina. Većina problema opisanih u radu su na neki način povezani. Gotovo svaki problem je tražio da se u nekom trenutku pronađe sjecište između dužina, ponekad je bilo potrebno provjeriti nalazi li se točka unutar nekog poligona, a specifično u slučaju Voronojevih dijagrama i Delauneyjevih triangulacija, rješavanje jednog problema može prethoditi rješavanju drugog.

S druge strane, može se primijetiti da se algoritmi često uvelike razlikuju u načinu izvođenja jer su njihove temeljne ideje pristupa problemu vrlo različite. Naravno, pravilno definiranje problema i postavljanje jasnih ograničenja na njih pomaže u tom procesu. Smišljanje efikasnih algoritama zahtjeva veliku količinu znanja vezanog uz razna geometrijska područja te dobro baratanje njima. Obrađeni problemi predstavljaju samo uvod u računalnu geometriju, a cilj rada je bio vizualizacijama i implementacijama približiti dane probleme i njihova rješenja. Sve implementacije rađene su u Pythonu čija jednostavna sintaksa olakšava čitljivost i "sitne poslove" kao što su sortiranja, izrada klasa i dokumentiranje koda.

Popis literature

[1] A. Selim i H. S. Muzafer, *Computational Geometry Applications*, 2018. [Na internetu]. Dostupno: `https://bit.ly/3dIxgvh` [pristupano 07.08.2022.].

[2] J. Renze i E. W. Weisstein, *Discrete Mathematics*, bez dat. [Na internetu]. Dostupno: `https://bit.ly/3QVY0GO` [pristupano 07.08.2022.].

[3] *Computational geometry*, bez dat. [Na internetu]. Dostupno: `https://bit.ly/3A86dkv` [pristupano 07.08.2022.].

[4] M. D. Mount, *CMSC754 - Computational Geometry*, 2012. [Na internetu]. Dostupno: `https://bit.ly/3K2LYtb` [pristupano 07.08.2022.].

[5] M. de Berg, M. van Kreveld i O. Overmars Mark Schwarzkopf, *Computational Geometry Algorithms and Applications*, 2000. [Na internetu]. Dostupno: `https://bit.ly/3c0hQ5f` [pristupano 07.08.2022.].

[6] P. P. Franco i S. Michael Ian, *Computational Geometry - An Introduction*. 175 Fifth Avenue, NY, USA: Springer-Verlag, 1985.

[7] V. John, *Mathematics for Computer Graphics - Fifth Edition*. 236 Gray's Inn Road, London WC1X 8HB, United Kingdom: Springer-Verlag, 2017.

[8] H. Kai i A. Alexander, *The point in polygon problem for arbitrary polygons*, 2001. [Na internetu]. Dostupno: `https://bit.ly/3AxGRxX` [pristupano 21.08.2022.].

[9] M. Avraham i K. Gary D., *An algorithm for computing the union, intersection or difference of two polygons*, 1989. [Na internetu]. Dostupno: `https://bit.ly/3K57hdo` [pristupano 11.08.2022.].

[10] J. Erickson, *Convex Hulls*, 2002. [Na internetu]. Dostupno: `https://bit.ly/3pvGz4w` [pristupano 22.08.2022.].

[11] *Gift Wrap Algorithm (Jarvis March Algorithm) to find Convex Hull*, bez dat. [Na internetu]. Dostupno: `https://bit.ly/3px6k48` [pristupano 23.08.2022.].

[12] F. Davis i J. Hongisto, *Two-Dimensional Convex Hull Algorithm Using the Iterative Orthant Scan*, 2017. [Na internetu]. Dostupno: `https://bit.ly/3T2ZKjy` [pristupano 24.08.2022.].

[13] P. N. Hossein, *Introduction to Probability, Statistics, and Random Processes - Set Operations*, 2014. [Na internetu]. Dostupno: `https://www.probabilitycourse.com` [pristupano 26.08.2022.].

[14] G. Gunther i H. Kai, *Efficient clipping of arbitrary polygons*, 17.4.1998. [Na internetu].
Dostupno: `https://bit.ly/3TSTb3g` [pristupano 26.08.2022.].

[15] S. S. W. i H. G. T., *An implementation of Watson's algorithm for computing 2-dimensional Delaunay triangulations*, 1984. [Na internetu].
Dostupno: `https://bit.ly/3QjRrNP` [pristupano 28.08.2022.].

[16] *Graphics! Voronoi, Delaunay, Natural Neighbor - Code, Sound & Surround E03*, 9.6.2021. [Youtube].
Dostupno: `https://bit.ly/3L259nl` [pristupano 29.08.2022.].

Popis slika