UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matic Mercina

Izračun triangulacije točk v ravnini

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Mentor: pred. dr. Boštjan Slivnik

Ljubljana, 2012

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00248/2012
Datum: 05.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MATIC MERCINA**

Naslov: **IZRAČUN TRIANGULACIJE TOČK V RAVNINI**
COMPUTING MINIMUM WEIGHT TRIANGULATION

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Napišite program, ki izračuna triangulacijo naključno izbranih točk v realni ravnini. Izračunana triangulacija naj ima kar se da majhno skupno dolžino vseh daljic, ki sestavljajo triangulacijo. Kot idejo za algoritem za izračun triangulacije vzemite metodo urejanja s porazdelitvami: zlivanje delnih rezultatov urejanja nadomestite z združevanjem dveh triangulacij. Program naj omogoča različne postopke delitve množice točk in iterativno izboljševanje izračunane triangulacije.

Mentor:
B. Slivnik
pred. dr. Boštjan Slivnik

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Matic Mercina, z vpisno številko **63090275**, sem avtor diplomskega dela z naslovom:

Izračun triangulacije točk v ravnini

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom pred. dr. Boštjana Slivnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 7. oktobra 2012 Podpis avtorja:

Vsebina

Povzetek

Diplomska naloga opisuje triangulacijo točk v realni ravnini, pri čemer se natančneje ukvarja s problemom triangulacije z najmanjšo dolžino povezav. Zaradi NP-polnosti problema se pojavi vprašanje, ali je nujno poznati točno rešitev za podano množico točk, ali je morda dovolj poznati razmeroma dober približek. Ob povečevanju velikosti problema se vedno bolj nagibamo k drugi možnosti, saj NP-polnega problema na velikih vhodih preprosto ne moremo izračunati v zadovoljivem času.

V nalogi je zato predstavljen algoritem tipa deli in vladaj, ki za vhodno množico točk najprej izračuna poljubno triangulacijo, nato pa jo z izmenjevanjem povezav približa triangulaciji z najmanjšo dolžino povezav. Implementirana rešitev omogoča različne načine delovanja v povezavi tako z načini deljenja na podprobleme, kot z načinom, na katerega triangulacijo izboljšuje. Rezultati prikazujejo, na kakšen način pridemo najhitreje do rešitve, kakšne so razlike v končnih rešitvah in kakšno časovno zahtevnost izkazuje algoritem.

Ključne besede

ravninski graf, triangulacija, MWT, konveksna ovojnica

Abstract

The thesis describes point set triangulation in a real plane, and deals with the minimum weight triangulation problem in more detail. Due to the NP-completeness of the problem, the following question arises: is it necessary to know the exact solution for the given point set, or is a close approximation enough? As we increase the size of the problem, we begin leaning towards the second option, due mainly to the fact that an NP-complete problem simply cannot be solved in sufficient time on large inputs.

For the above reasons, we present a divide and conquer algorithm, which first calculates an arbitrary triangulation for the given input, and then gradually improves the result by switching connections. The implemented solution allows for different modes of operation based on methods of subdivision and methods of improvement. The results show which operating mode is the fastest, what are the differences in the final solutions, and what is the time complexity of the algorithm.

Keywords

planar graph, triangulation, MWT, convex hull

1. Uvod

Triangulacija je eden od temeljnih problemov v računalniški geometriji, saj je ob delu z zapletenimi geometrijskimi objekti prvi korak pogosto razbitje le-teh na preproste geometrijske oblike. V dveh dimenzijah je najpreprostejša med njimi trikotnik.

Potreba po izračunu triangulacije za množico točk se naravno pojavi v mnogih aplikacijah, na primer v analizi končnih elementov (ang. finite element analysis), numerični analizi in računalniški grafiki, kjer se uporablja za izračun trikotne mreže (ang. triangle mesh) na podlagi množice točk [3].

Izračun triangulacije oz. postopek trianguliranja množice točk lahko povzamemo kot vstavljanje vseh možnih povezav med točkami na tak način, da je celotne območje razdeljeno na trikotnike in da se povezave med seboj ne sekajo. Ob tem se pojavijo posebne vrste triangulacij, ki optimizirajo določene geometrijske lastnosti (npr. velikost trikotnikov in koti v trikotnikih). Najbolj znana je Delauneyeva triangulacija, ki je zaradi svojih geometrijskih lastnosti (enakomerni in široki trikotniki) najpogosteje uporabljana triangulacija, med drugim tudi za izračun trikotne mreže v računalniški grafiki.

V diplomskem delu sem se osredotočil na triangulacijo z najmanjšo dolžino povezav (ang. minimum weight triangulation, ali MWT). Z uporabo prejšnjega povzetka lahko izračun triangulacije z najmanjšo dolžino povezav povzamemo kot trianguliranje množice točk na takšen način, da je skupna vsota vseh povezav v nastali triangulaciji čim krajša. Čeprav se problem na prvi pogled zdi trivialen, se ob natančnejši analizi izkaže, da je izjemno kompleksen, predvsem na velikih množicah točk.

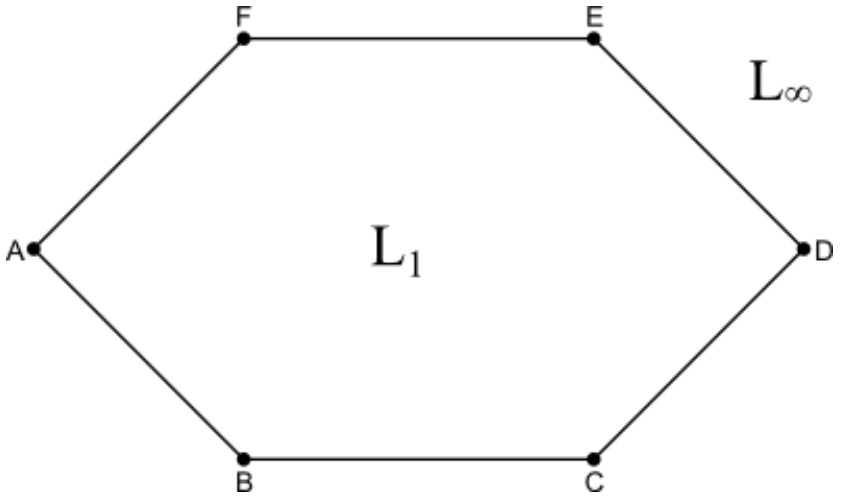
V nadaljevanju so natančneje predstavljeni osnovni koncepti triangulacije in problem triangulacije z najmanjšo dolžino povezav. Predlagan je algoritem za rešitev problema ter natančnejši opis njegove implementacije, navedene pa so tudi izmerjene zmogljivosti implementirane rešitve.

2. Definicija problema

Vzemimo realno ravnino $\mathbb{R}^2$. Na tej ravnini vzemimo končno množico točk $V = \mathbb{R}\times\mathbb{R}$. Triangulacija množice točk V je v tem primeru končna množica povezav $E = \mathrm{V}\times\mathrm{V}$, za katero velja:

1. nobena povezava iz množice *E* se ne začne in konča v isti točki,
2. noben par povezav e_1, e_2 iz množice *E* se ne seka ali prekriva na daljšem intervalu in
3. v množico E ne moremo vstaviti nobene druge povezave z izvorom in ponorom v množici V, ne da bi kršili pravila številka 2 (rečemo lahko tudi, da je množica E **maksimalna)**.

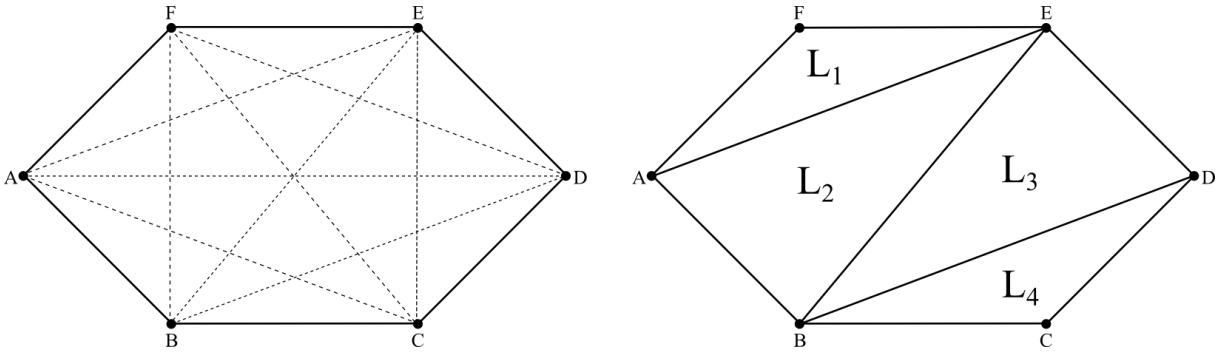
Triangulacijo lahko definiramo tudi kot poseben ravninski graf *T(V, E)*. Vsako najmanjše področje ravninskega grafa, ki je omejeno z točkami iz množice V, imenujemo **lice grafa**. Za vsako lice lahko navedemo število vseh povezav, ki ga omejujejo – tej lastnosti rečemo **dolžina** ali **velikost** lica. Vsak graf ima vsaj eno lice – **zunanje lice** grafa, ki mu rečemo tudi **neskončno lice**. Ostala lica so notranja lica grafa.



Slika 1: Graf z dvema licema. Z oznako L_1 je označeno notranje lice, z L_∞ zunanje ali neskončno lice.

Če je graf T triangulacija, potem velja, da je velikost vsakega lica (razen zunanjega), enaka 3. Drugače povedano, vsako notranje lice grafa triangulacije je trikotnik. Razlog za to lastnost je tretje pravilo veljavne triangulacije – množica povezav je maksimalna.

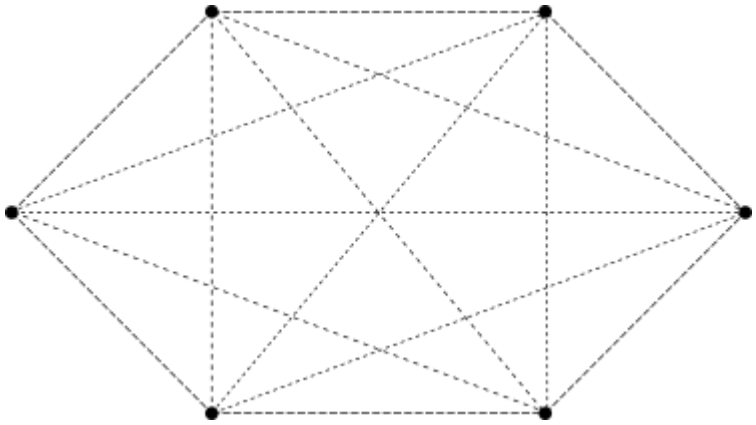
Trikotnik je največje možno lice grafa, ki ga ne moremo več razdeliti na manjše dele z vstavljanjem dodatnih povezav. Če v grafu obstaja lice z velikostjo večjo od 3 tedaj v graf lahko vstavimo povezavo, ki lice razdeli na dva manjša lica, ne da bi prekršili pravilo 2. To pa tudi pomeni, da za graf ne velja tretje pravilo veljavne triangulacije, iz česar sledi, da graf ne predstavlja veljavne triangulacije.



Slika 2: S črtkanimi črtami so prikazane vse povezave, ki notranje lice grafa razpolovijo na pol in se ne sekajo z obstoječo povezavo grafa (slika levo). Desno je prikazan graf veljavne triangulacije na istih točkah z označenimi notranjimi lici $L_1 - L_4$.

2.1 Različne triangulacije

V začetku ustvarjanja triangulacije imamo pred seboj množico vseh točk V. Preden dejansko začnemo z izračunom triangulacije, lahko za to množico najdemo vse **potencialne povezave**, ki lahko postanejo del triangulacije. Vsak par točk AB iz množice točk V omejuje potencialno povezavo triangulacije, v kolikor nobena druga točka ne leži na daljici AB.



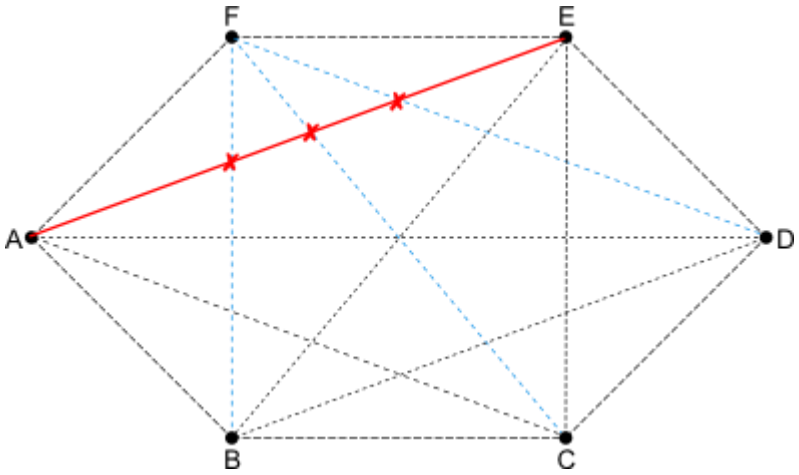
Slika 3: Vse potencialne povezave na množici 6 točk.

Če poenostavimo algoritem za izdelavo triangulacije, ga lahko predstavimo kot je prikazano v nadaljevanju.

```
Function Triangulate()                                  Komentarji
V ← {v1,v2,v3,…,vn}                                     množica vseh točk
P ← findAllPotentialConnections(V)                      množica vseh potencialnih povezav
E ← {}                                                  množica vseh povezav je na začetku prazna
foreach candidate in C:
    foreach edge in E:
        if cross(candidate, edge):                      če se kandidat križa z obstoječo povezavo,
            break and continue                          preizkusi naslednjega kandidata
      add {candidate} → E                               če se kandidat ne križa z nobeno povezavo, ga
                                                        dodaj v triangulacijo
```

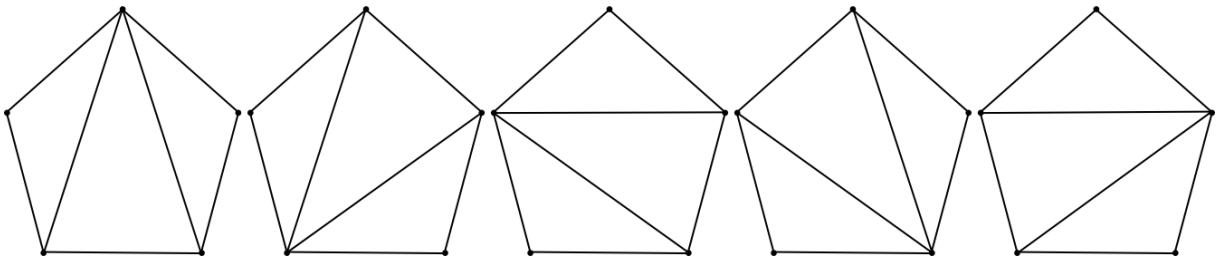
Algoritem 1: preprosta triangulacija.

Algoritem se sprehodi skozi seznam vseh potencialnih povezav in jih dodaja v triangulacijo v kolikor izpolnjujejo drugi pogoj veljavne triangulacije: povezava se ne sme križati z obstoječo povezavo iz triangulacije. Prav zaradi tega pogoja vsaka dodana povezava onemogoči celo vrsto drugih potencialnih povezav. V kolikor bi v istem koraku zunanje zanke algoritma namesto izbrane povezave izbrali eno izmed potencialnih povezav, ki se križajo z izbrano povezavo, bi dobili povsem drugačno triangulacijo.



Slika 4: Povezava AE onemogoči povezave BF, CF in DF.

Veljavna triangulacija množice točk torej ni enolična. Število vseh možnih triangulacij na isti množici je odvisno tako od števila točk kot njihovega relativnega položaja in je zato težko natančno določeno, velja pa, da narašča eksponentno s številom točk.



Slika 5: Vse veljavne triangulacije na isti množici petih točk.

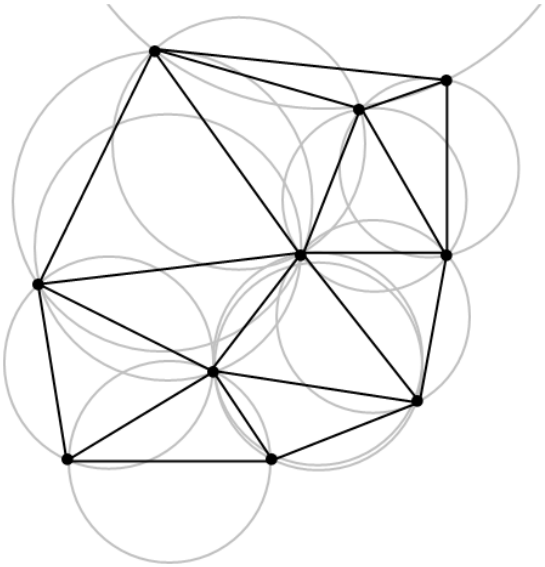
Iz gore različnih triangulacij, ki se jih torej lahko izračuna za isto množico točk, se za različne aplikacije pogosto potrebuje tiste, ki izpolnjujejo določene optimalne lastnosti. Take triangulacije so znane pod skupnim imenom **optimalne triangulacije** [3]. Med najbolj znanimi od teh sta Delaunayeva triangulacija in triangulacija z najmanjšo dolžino povezav (ang. minimum weight triangulation).

2.1.1 Delauneyeva triangulacija

Delauneyeva triangulacija je verjetno najbolj znana in najpogosteje uporabljena triangulacija. Da je triangulacija Delauneyeva, mora zanjo veljati, da očrtani krog vsakega trikotnika (lica) v triangulaciji znotraj svoje površine ne vsebuje nobene točke. Zaradi tega pogoja ima vsaka Delauneyeva triangulacija zanimive lastnosti [2]:

1. Koti v Delauneyevi triangulaciji so vedno široki. Dokazati se da, da je najmanjši kot v Delauneyevi triangulaciji vedno največji možen. V množici vseh triangulacij za podano množico točk je torej nemogoče najti triangulacijo z večjim najmanjšim kotom.
2. Največji očrtani krog notranjega lica (trikotnika) v Delauneyevi triangulaciji je najmanjši možen.
3. Največji minimalni obdajajoči krog (ang. smallest enclosing circle) trikotnega lica v Delauneyevi triangulaciji je najmanjši možen.

Prav zato, ker Delauneyeva triangulacija istočasno optimizira cel kup geometrijskih lastnosti, se šteje kot najbolj splošno uporabna triangulacija, in je kot taka najpogosteje uporabljena triangulacija množice točk.



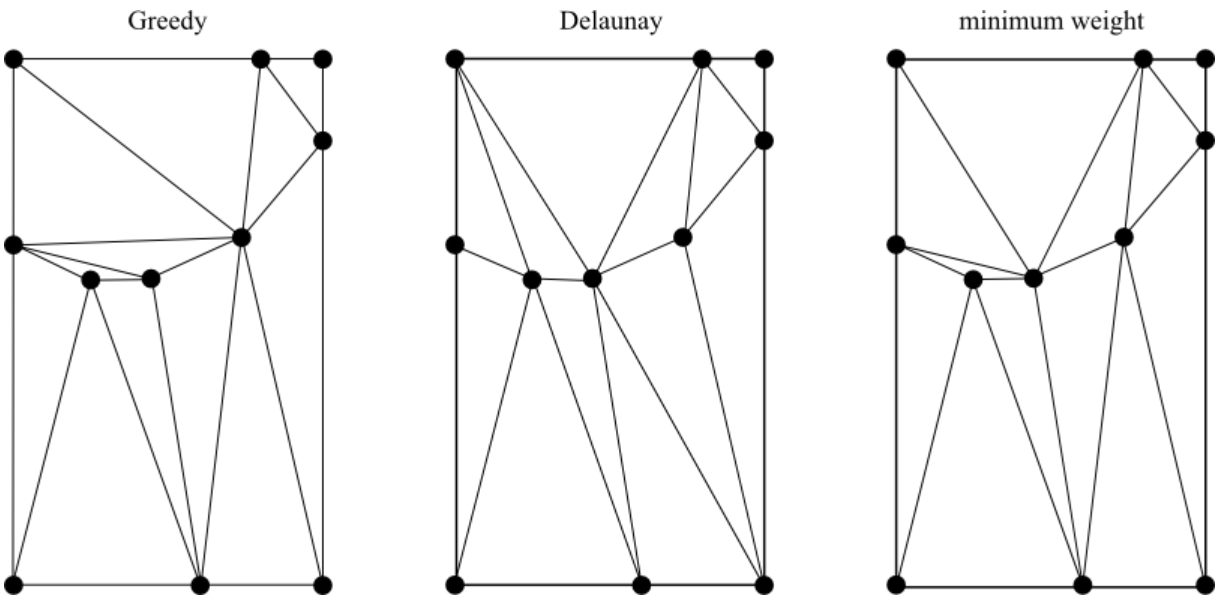
Slika 6: Delauneyeva triangulacija z vidnimi očrtanimi krogi [4].

2.1.2 Triangulacija z najmanjšo dolžino povezav

V svojem diplomskem delu sem se podrobneje ukvarjal s triangulacijo z najmanjšo dolžino povezav. Da je triangulacija T triangulacija z najmanjšo dolžino povezav, mora zanjo poleg že omenjenih treh lastnosti veljavne triangulacije veljati tudi naslednje: vsaka druga triangulacija T_i na isti množici točk ima večjo ali kvečjemu enako skupno dolžino vseh povezav kot triangulacija T.

Kot je torej razvidno že iz imena triangulacije gre za triangulacijo, ki ima to optimalno lastnost, da je skupna dolžina vseh povezav v triangulaciji najmanjša možna.

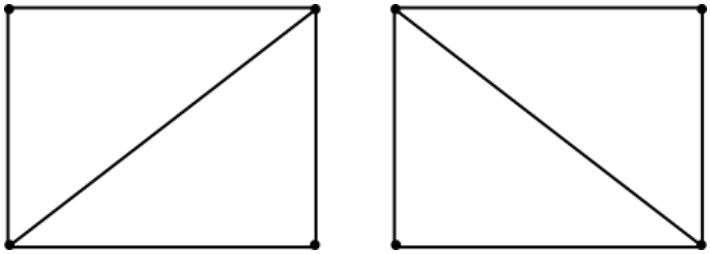
Problem triangulacije z najmanjšo dolžino je postal razvpit potem, ko sta ga leta 1979 Garey in Johnson vključila v knjigo Computers and Intractability: A Guide to the Theory of NP-Completeness kot enega izmed odprtih problemov NP-polnosti (problem, za katerega ne obstaja dokaza, da je NP-poln, hkrati pa ne obstaja algoritem, ki bi ga izračunal v polinomskem času) [1]. Avtorji so se problema lotevali na različne načine, med drugim z dinamičnim programiranjem, iskanjem rešitev na omejenih primerih, ipd. Popularna metoda za iskanje rešitve je postalo iskanje podobnih triangulacij. Prvi tak algoritem je povezave preprosto vstavljal po vrstnem redu glede na njihovo dolžino. Triangulacija ustvarjena s tem algoritmom je znana pod imenom »greedy triangulation«. Nekateri avtorji so predlagali Delauneyevo triangulacijo kot morebiten približek, vendar pa je bilo kasneje dokazano, da se dolžina povezav Delauneyeve triangulacije od dolžine povezav triangulacije TNDP lahko razlikuje za linearni faktor [3].



Slika 7: Triangulacija z najmanjšo skupno dolžino povezav(desno) in njeni približki z uporabo požrešnega algoritma in Delaunayeve triangulacije [3].

Triangulacija z najmanjšo dolžino povezav je ostal eden najdlje odprtih problemov NP-polnosti vse do leta 2008, ko sta avtorja Mulzer in Rote dokazala NP-polnost z objavo dela »Minimum weight triangulation is NP-hard« [3].

Za triangulacijo z najmanjšo dolžino povezav poleg osnovnih lastnosti triangulacij velja tudi, da ni enolična. Trditev lahko dokažemo na podlagi primera: predstavljajmo si, da imamo množico štirih točk postavljenih na ogliščih pravokotnika. Med potencialnimi povezavami triangulacije imamo vse 4 stranice pravokotnika in obe diagonali. Ker se med sabo križata samo diagonali, imamo na voljo 2 rešitvi, eno za vsako diagonalo. Ker pa sta diagonali enako dolgi, obe triangulaciji izpolnjujeta pogoj za triangulacijo z najmanjšo dolžino povezav.



Slika 8: Obe triangulaciji štirih točk na ogliščih pravokotnika imata najmanjšo možno dolžino povezav.

Cilj algoritma, ki išče triangulacijo z najmanjšo dolžino povezav, je torej poiskati eno izmed veljavnih rešitev za podano množico točk. Kot smo že omenili, je problem težji kot se zdi na prvi pogled. Ker zaradi NP-polnosti problema na zelo velikih množicah točk preprosto ne moremo rešiti, se v diplomskem delu osredotočimo na iskanje približka dejanski rešitvi.

Cilj diplomskega dela je torej napisati in predstaviti algoritem, ki za množico naključno izbranih točk poišče približek triangulaciji z najmanjšo dolžino povezav.

3. Algoritem

Problema triangulacije z najmanjšo dolžino povezav sem se lotil z algoritmom, ki poizkuša najti čim boljši približek iskanemu rezultatu. Deluje tako, da za množico točk najprej izračuna poljubno triangulacijo, nato pa jo približa želenemu rezultatu s postopnim izboljševanjem. Konceptualno lahko algoritem razdelimo na dve fazi:

1. triangulacija točk s pristopom »deli in vladaj« in
2. izboljšava triangulacije z izmenjevanjem povezav.

Natančnejši opis faz in delovanja algoritma sledi v nadaljevanju.

3.1 Triangulacija točk s pristopom deli in vladaj

Deli in vladaj je pogosto uporabljen pristop v programiranju. Algoritem, ki deluje po principu deli in vladaj, problem deli na manjše podprobleme, vse dokler niso ti dovolj trivialni, da so lahko rešeni neposredno. Delne rešitve so nato združene in skupaj tvorijo rešitev originalnega problema.

Ko govorimo o triangulaciji je problem trivialen, če imamo pred seboj prazno množico, ali množico z eno samo točko. V nobeno izmed teh ne moremo vstaviti povezave (nobena nima vsaj dveh točk, ki bi lahko tvorili povezavo), torej lahko predpostavimo, da sta množici že triangulirani. Algoritem tako vsako množico in vsako podmnožico točk deli dalje na manjše dele vse dokler nima pred seboj prazne množice ali ene same točke. Triangulirane podmnožice mora nato še združiti.

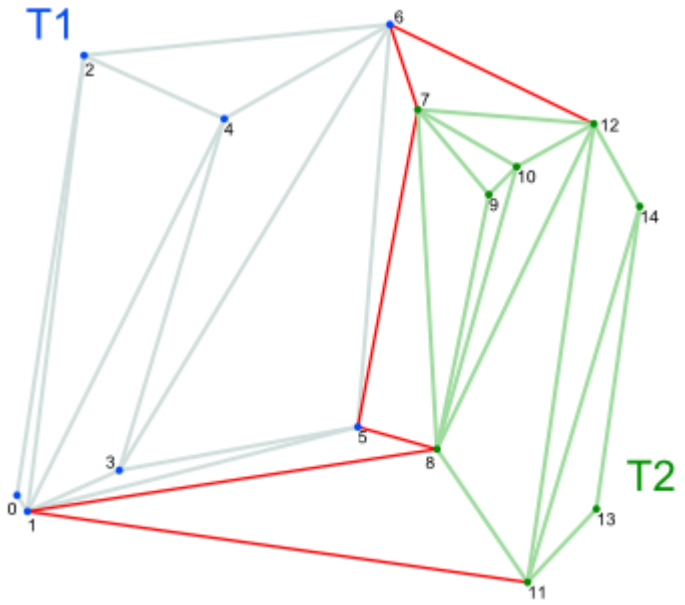
Osnovni algoritem lahko predstavimo z sledečo psevdokodo:

```
function Triangulate(V):                                   Komentarji
if |V| ≤ 1 return {V, {}};          Množica z eno točko ali manj je triangulirana. Rezultat
                                    triangulacije je triangulacija z množico točk(prazna ali
                                    ena) in prazno množico povezav.
else
     V₁, V₂ ← split(V)                                  Množico razdeli na manjše množice.
     T₁ ← Triangulate(V₁)                                    Trianguliraj prvo podmnožico.
     T₂ ← Triangulate(V₂)                                    Trianguliraj drugo podmnožico
     T ← Join(T₁, T₂)                                Delne triangulacije združi v rezultat.
     return T                                                      Vrni končni rezultat.
```

Algoritem 2: algoritem tipa deli in vladaj za izračun osnovne triangulacije.

Združevanje delnih triangulacij je samo po sebi razmeroma preprosto. Algoritem ima ob začetku združevanja pred seboj dve veljavni triangulaciji $T_1 = \{V_1, E_1\}$ in $T_2 = \{V_2, E_2\}$. Vsaka triangulacija ima svojo množico točk (V) in množico povezav (E). Množico novo-

nastalih povezav ob združevanju imenujemo **šiv** triangulacij. Triangulaciji se združi tako, da se vstavi vsako povezavo AB z izvorom (A) v množici V_1 in ponorom (B) v množici V_2, v kolikor se ta ne seka s katerokoli povezavo iz množic E_1, E_2 ali nastajajočega šiva triangulacij.



Slika 9: Triangulacija po združitvi T1 in T2. Šiv triangulacij je označen z rdečo.

Izkaže pa se, da je prav združevanje triangulacij ozko grlo algoritma. Z naraščanjem velikosti delnih triangulacij narašča tako število vseh potencialnih povezav šiva kot število obstoječih povezav, ki jih mora algoritem v vsakem koraku preveriti za sekanje. Najhuje je v primerih, ko se preverja veljavno povezavo. Ker se sekanja nikoli ne ugotovi, gre algoritem skozi popolnoma vse povezave, preden nadaljuje z delovanjem. Logaritemske časovne zahtevnosti s tako omejitvijo ni mogoče doseči, saj število povezav v odvisnosti od števila točk zelo hitro narašča. Tu se pojavita dve pomembni vprašanji:

1. je vsaka povezava z izvorom v eni in ponorom v drugi delni triangulaciji res potencialna povezava šiva, in
2. ali je res potrebno preverjati vse obstoječe povezave za morebitno sekanje?

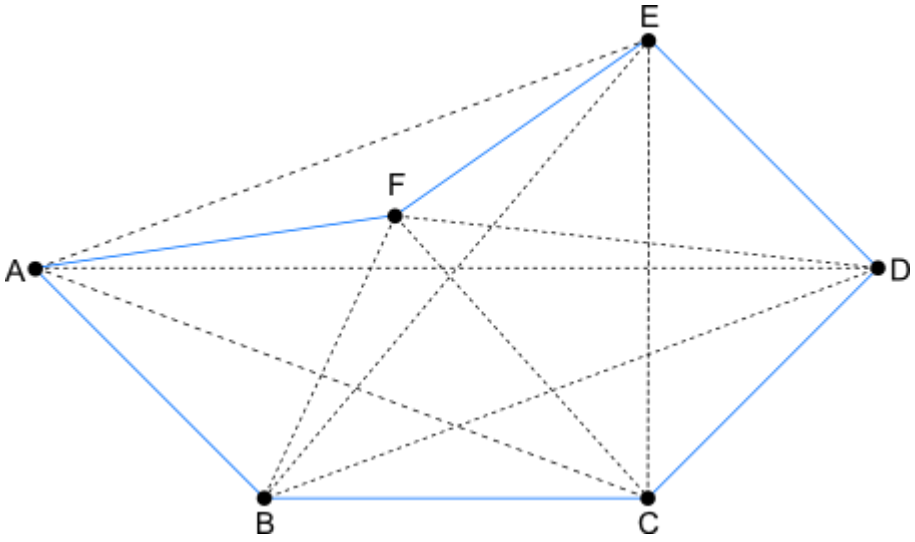
3.1.1 Obod triangulacije

Preden odgovorimo na vprašanji si poglejmo pomembno lastnost triangulacije množice točk v ravnini: obod triangulacije. Obod triangulacije je množica skrajnih povezav triangulacije, ki triangulacijo omejujejo navzven. Obod triangulacije ima par pomembnih lastnosti:

1. vedno omejuje konveksen poligon in
2. je konstanten za vse triangulacije na isti množici točk.

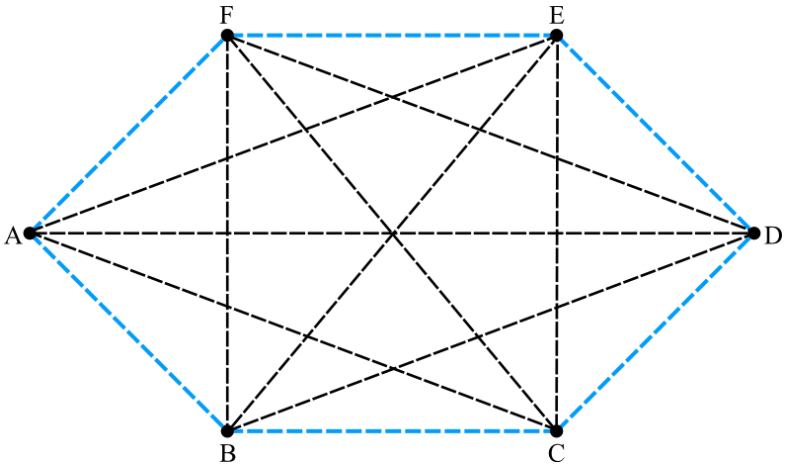
Najprej preverimo prvo trditev. Če bi obod lahko omejeval konkavni poligon, bi sledilo, da ima nekje vdrtino. V kolikor v poligonu med točko A in B obstaja vdrtina, se le-to da

zagotovo zapreti z vstavitvijo povezave AB, ki se ne seka z obstoječo povezavo triangulacije. Zaradi druge lastnosti veljavne triangulacije – vstavi se vsako povezavo, ki se ne križa s katero drugo, mora taka povezava že obstajati, torej je poligon, ki ga omejuje obod triangulacije, zagotovo konveksen.



Slika 10: Obod (modra) ne more omejevati konkavnega poligona, saj vedno obstaja potencialna povezava (AE), ki ga spremeni v konveksnega.

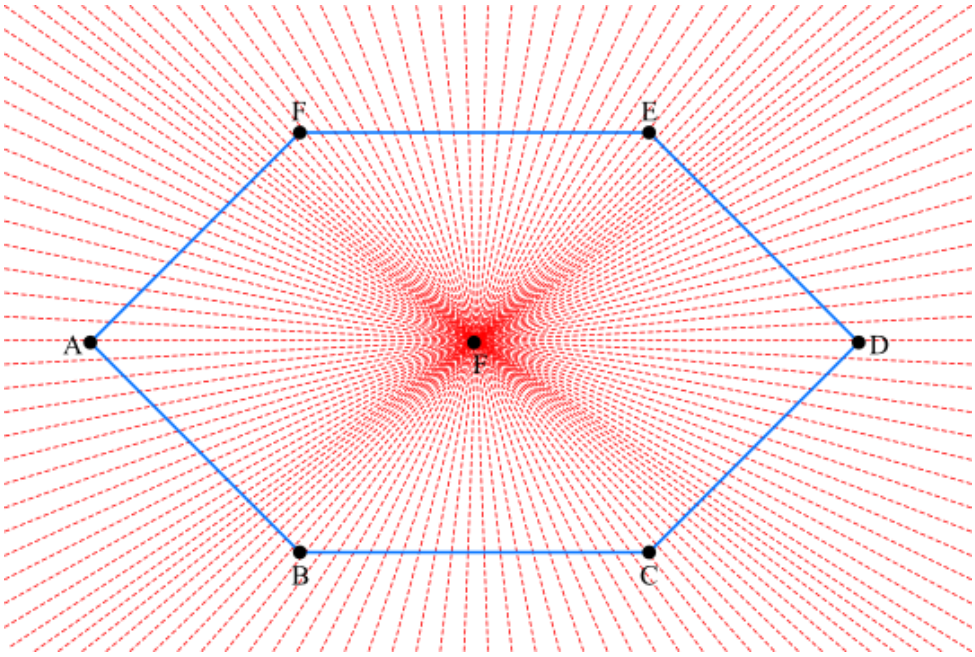
Razlog, da je obod konstanten za vse triangulacije na isti množici točk, leži v pomembni lastnosti obodnih povezav: obodne povezave se ne sekajo z nobeno potencialno povezavo triangulacije. Z drugimi besedami: v množici potencialnih povezav triangulacije ne obstaja taka povezava *e*, ki bi jo lahko vstavili pred obodno povezavo triangulacije in s tem preprečili vstavitev le-te. Prav zato so obodne povezave vedno prisotne ne glede na ostale povezave v triangulaciji.



Slika 11: Nobena potencialna povezava ne seka oboda (modro).

Z vpeljavo oboda triangulacije lahko tako točke kot povezave triangulacije razdelimo v dve skupini: obodne in notranje. Obodne povezave so povezave, ki sestavljajo obod triangulacije. Notranje povezave so vse ostale, ki ležijo znotraj oboda. Podobno so obodne točke tiste, ki omejujejo obodne povezave in notranje vse ostale.

Za vsako notranjo točko velja sledeče: vsaka povezava AB s točko A znotraj oboda triangulacije in točko B izven triangulacije, bo zagotovo sekala eno izmed obodnih povezav. S tem smo tudi odgovorilo na prvo vprašanje – ne, algoritmu ni potrebno preverjati vseh potencialnih povezav, temveč zgolj tiste, ki imajo izvor v obodu prve triangulacije in ponor v obodu druge.



Slika 12: Obod triangulacije blokira vsako povezavo (označeno z rdečo) iz notranje točke G ven iz triangulacije.

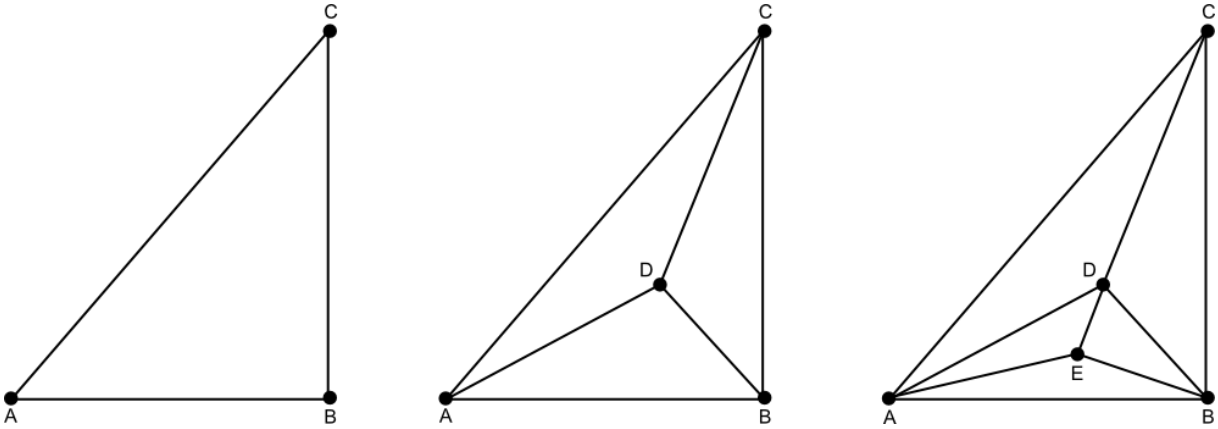
Podobno velja tudi za vsako notranjo povezavo: da potencialna povezava nove triangulacije seka notranjo povezavo ene izmed delnih triangulacij mora najprej prečkati obod triangulacije. Z drugimi besedami, če se notranja povezava delne triangulacije seka s potencialno povezavo šiva triangulacij, se slednja zagotovo seka tudi z eno izmed obodnih povezav. S tem lahko odgovorimo tudi na drugo vprašanje – ne, za morebitno sekanje ni potrebno preverjati vseh povezav nastajajoče triangulacije. Dovolj je, da preverimo obodne povezave obeh delnih triangulacij in povezave šiva.

Da razložimo, za koliko lahko z uporabo pravkar izpeljanih lastnosti triangulacij izboljšamo hitrost algoritma, pa si moramo najprej pogledati še eno lastnost, ki jo lahko določimo za triangulacijo ob poznavanju oboda – število vseh povezav. Predpostavimo, da imamo pred seboj triangulacijo s 3 točkami. Te so seveda vedno vse na obodu triangulacije. Taka

triangulacija ima skupno 3 povezave. Če dodamo v isto triangulacijo še eno točko znotraj oboda triangulacije, pridobimo še 3 dodatne povezave: eno za vsako oglišče oboda. Isto velja za vsak prazen obod velikosti *n*: prva točka znotraj oboda se lahko poveže z **vsemi** obodnimi točkami. Ker je notranjost oboda zdaj triangulirana in torej razdeljena na trikotnike, bo vsaka nova točka znotraj oboda triangulacije stala v enem izmed lic triangulacije in prinesla 3 dodatne povezave, 1 za vsako oglišče. Če pravilo posplošimo, pridemo do sledeče formule za izračun skupnega števila povezav v triangulaciji v odvisnosti od števila točk in števila točk na obodu:

$$|E| = 2 * |O| + (|V| - |O| - 1) * 3,$$

kjer je E množica vseh povezav, V množica vseh točk, in O množica vseh obodnih točk triangulacije. Velja, da je število točk na obodu triangulacije z *n* točkami, kjer je $n \geq 3$, najmanj 3 (ko 3 točke omejujejo vse ostale) in največ *n* (ko so vse točke v obodnem položaju).



Slika 13: Triangulacija točk ABC. Z dodano točko D povečamo število povezav za 3 (v splošnem je povečava pri dodajanju točk v prazen obod enaka številu obodnih oglišč). Točka E doda 3 povezave in trikotnik, v katerega je vstavljena, razdeli na 3 manjše trikotnike. Posledično velja isto za vsako točko po njej.

Predpostavimo, da združujemo triangulaciji T_1, ki ima skupno *n* točk, in T_2, ki ima skupno *m* točk. Z uporabo pravkar izpeljane formule za izračun števila povezav v triangulaciji, lahko o pohitritvi algoritma z vpeljavo optimizacij 1 in 2 povemo naslednje:

1. V najslabšem primeru (ko imata obe delni triangulaciji vse točke v obodu) lahko ob vsakem preverjanju sekanja preskočimo (*n* – 3) + (*m* – 3) notranjih povezav.
2. V najboljšem primeru(ko imata obe delni triangulaciji 3 obodne točke) lahko za preverjanje sekanja preskočimo ((*n* – 3 – 1) * 3 + 3) + (*m* – 3 – 1) * 3 + 3) notranjih povezav in pri preizkušanju novih povezav preskočimo (*n* – 3) * *m* + 3 * (*m* – 3) povezav.

3. Izboljšave so multiplikativne. Z vsako potencialno povezavo, ki jo preskočimo, preskočimo tudi celoten proces preverjanja sekanja za le-to.

Končni algoritem za združevanje zato sestoji tudi iz algoritma za izračun oboda in izgleda kot sledi:

function Join(T1, T2)	**Komentarji**
`S ← {}`	`Množica povezav šiva je sprva prazna.`
`E1, V1 ← Hull(T1)`	`Iz točk in povezav delnih triangulacij se`
`E2, V2 ← Hull(T2)`	`izloči obodne.`
`for each v1 in V1`	
`  for each v2 in V2`	`Preizkusi vsako novo povezavo v1v2.`
`  for each e in E1, E2, S`	
`  if Cross(e, {v1, v2})`	`Če se povezava križa z obstoječo povezavo,`
`  break and continue`	`se dalje preizkuša naslednjo povezavo.`
`  add {v1, v2} → S`	`V kolikor se nova povezava ne križa z`
	`nobeno obstoječo, se jo doda v šiv.`

Algoritem 3: Združevanje delnih triangulacij.

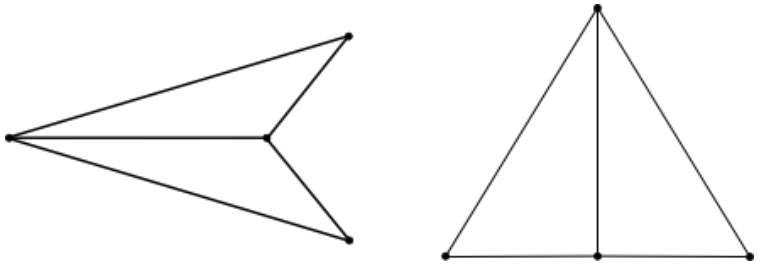
3.2 Izboljševanje triangulacije z izmenjevanjem povezav

Algoritem iz prve faze povezuje točke po vrsti. Zaradi tega je verjetnost, da nastala triangulacija izkazuje kakršne-koli optimalne lastnosti, izjemno majhna, verjetnost, da ima triangulacija najmanjšo možno dolžino povezav, pa še toliko manjša. Tu nastopi druga faza algoritma, ki rezultat približa iskani triangulaciji. Algoritem za drugo fazo lahko povzamemo tako: v nastali triangulaciji poišče vse poligone s štirimi oglišči (štirikotnike), ki znotraj svoje površine ne vsebujejo nobene druge točke. Nato ugotovi, kateri izmed teh imajo na voljo diagonalo, ki je krajša od trenutno izbrane, in diagonali zamenja.

Ob tem se pojavita vsaj dve omejitvi: štirikotnik, ki mu želimo menjati diagonalo, mora biti **konveksen** in ne sme vsebovati kota, ki je enak 180°. Z drugimi besedami, štirikotnik mora biti **strogo konveksen**. Za konveksni objekt v ravnini velja, da se ne da najti para točk iz površine objekta, za katerega velja, da daljica, ki jo omejujeta, ni v celoti znotraj površine objekta. Konveksni poligon ima zato vse notranje kote manjše ali enake 180°. Pri strogo konveksnemu poligonu je omejitev strožja in mora biti vsak notranji kot strogo manjši od 180°.

Razlog za omejitev je preprost: vsak štirikotnik, ki ni strogo konveksen, ima znotraj svoje površine samo eno možno diagonalo. V primeru, da je štirikotnik konkavni, se druga »diagonala« nahaja izven štirikotnika in mora torej, v kolikor gre za veljavno triangulacijo, povezava na tem mestu že obstajati, ali pa obstaja druga povezava, ki se s to seka. V primeru,

da je štirikotnik konveksen, vendar pa vsebuje 180 stopinjski kot, pa se druga diagonala nahaja na isti premici, kot dve povezavi oboda in bi se v primeru, da jo vstavimo, z njimi prekrivala na celotnem intervalu, na katerem se nahaja.

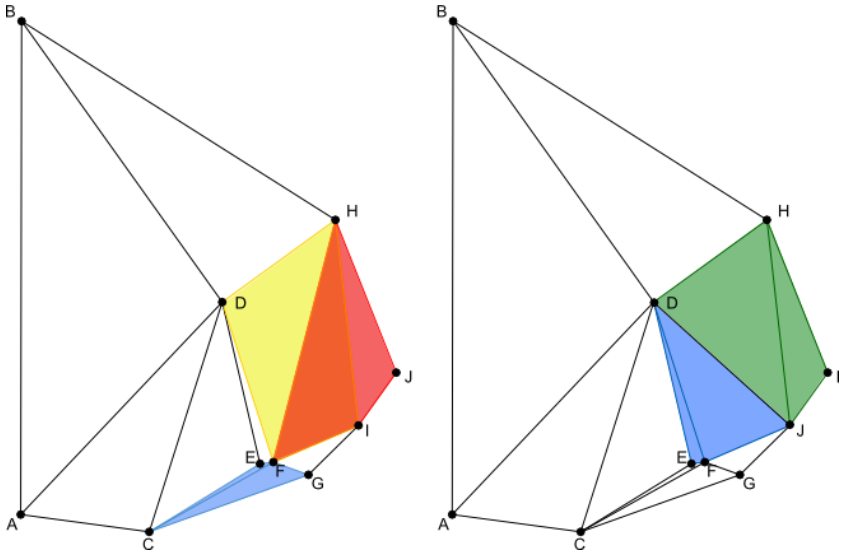


Slika 14: konkavni in šibko konveksni štirikotnik.

Osnovni algoritem pa sam po sebi ni dovolj. Vsakič ko zamenjamo diagonalo štirikotnika, se namreč zgodita dve stvari:

1. vseh štirikotnikov, ki so pravkar zamenjano diagonalo imeli v svojem obodu, ne moremo več uporabiti, ker ne obstajajo,
2. nova diagonala lahko postane obodna povezava za nove štirikotnike z daljšo izbrano diagonalo.

Za ponazoritev vzemimo primer na sliki 15, ki prikazuje analizo vseh štirikotnikov, ki jih lahko uporabimo za izboljšavo triangulacije. Algoritem najde tri štirikotnike: CEFG, DFIH in FHJI (slika levo). Kritična sta drugi in zadnji štirikotnik, ki se prekrivata. Ko algoritem zamenja diagonalo FH v drugem štirikotniku, štirikotnik FHJI ne obstaja več. Hkrati se pojavita dva nova štirikotnika DEFJ in DHIJ (slika desno), ki ju lahko uporabimo za nadaljnjo izboljšavo.



Slika 15: Analiza štirikotnikov v triangulaciji.

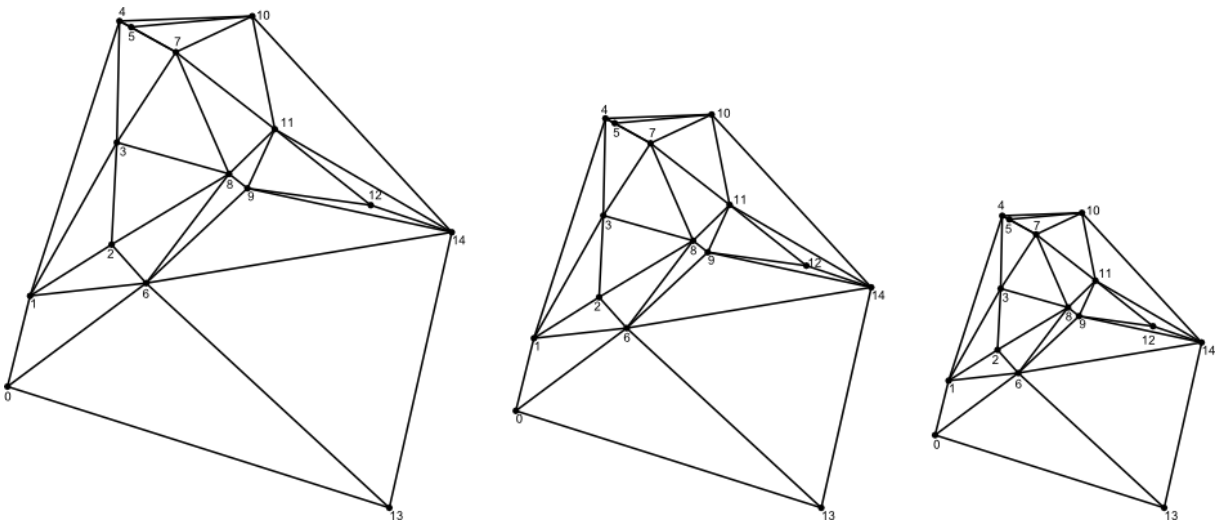
Zaradi teh razlogov je potrebno po vsaki zamenjavi povezave ponovno preveriti veljavnost štirikotnikov pridobljenih v analizi in s ponovno analizo poiskati morebitne nove. Končni algoritem izgleda takole:

function Improve(T)	**Komentarji**
q $\{d_1, d_2, v_1, v_2\}$	Štirikotnik za zamenjavo sestoji iz diagonalnih točk d1 in d2 in preostalih
Q ← Analyze(T)	Z analizo triangulacije se pridobi vse štirikotnike, ki jim lahko zamenjamo diagonalo
for each q ∈ Q:	
Remove $\{q.d_1, q.d_2\}$ ← T.E	Iz množice povezav triangulacije odstrani staro diagonalo
Add $\{q.v_1, q.v_2\}$ → T.E	Novo diagonalo se vstavi v triangulacijo.
Q ← Re-Analyze(T, $q.d_1$, $q.d_2$)	Triangulacijo se ponovno analizira za spremembe do katerih privede zamenjava stare diagonale

Algoritem 4: Postopno izboljševanje triangulacije.

4. Implementacija

Končni algoritem izvaja triangulacijo poljubne množice točk na realni ravnini, ki se nahajajo na intervalu [(0,0), (1,1)]. Kljub navidezni omejitvi lahko algoritem triangulira poljubne množice točk, saj lahko vsako množico točk skrčimo na interval [(0,0), (1,1)] in obratno (slika 16). Algoritem pozna različne načine delitve problema in različne načine optimizacije osnovne triangulacije, ki so natančneje razloženi v kasnejših podpoglavjih.



Slika 16: Isto triangulacijo lahko ustvarimo na različnih intervalih.

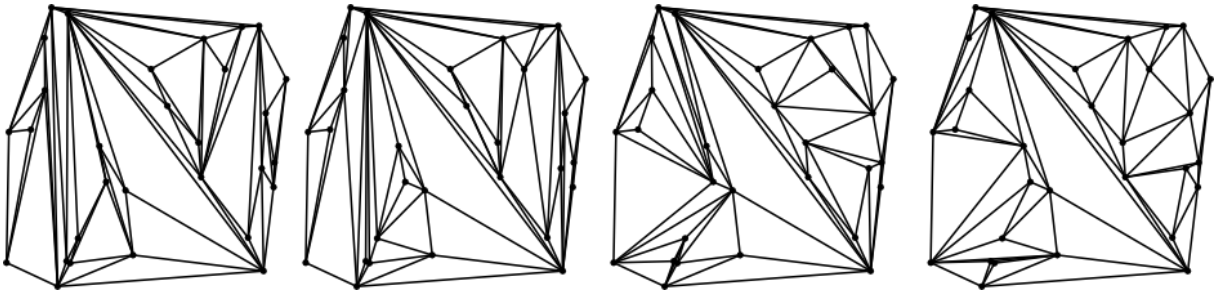
Algoritem je implementiran z uporabo programskega ogrodja .NET. Vsa koda je napisana v programskem jeziku C#. Program za vhod sprejme množico točk, podanih kot par koordinat (x, y) na osnovi podatkovnega tipa Double, in podatke o načinu delitve in izboljševanja, izpisu in shranjevanju testnih primerov.

4.1 Deljenje na podprobleme

Kot že povedano v prejšnjem poglavju, algoritem deluje v dveh fazah. V prvi ustvari poljubno triangulacijo, ki jo nato v drugi fazi približa želenemu rezultatu – triangulaciji z najmanjšo dolžino povezav. Rekli smo tudi, da algoritem deluje po principu »deli in vladaj«, torej množico vhodnih točk deli na manjše dele, dokler ni podmnožica prazna ali vsebuje eno točko – taka množica je obvladljiva sama po sebi, saj je že triangulirana. Nismo pa povedali, na kakšen način množico razdelimo na manjše podmnožice.

Pristopov k deljenju je več, izbran pristop pa ima velik vpliv na končen rezultat. Algoritem ponudi dva različna načina deljenja: deljenje na polovico po osi x ali osi y koordinatnega sistema in deljenje na četrtine. Poleg tega ponuja dva različna pristopa k samemu deljenju: deljenje z enakomerno razporeditvijo in deljenje s prepolovitvijo območja. Pri deljenju po prvem pristopu bosta obe polovici vedno dobili enako število točk, oziroma se bosta

razlikovali za največ eno, v kolikor število točk ni deljivo z 2. Pri deljenju po drugem pristopu se razpolavlja os, po kateri delimo točke. Ker točke po osi niso nujno enakomerno razporejene, s tem pristopom dobimo drugačne rezultate.



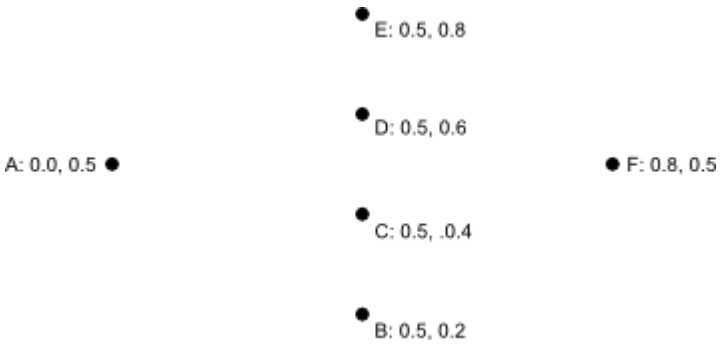
Slika 17: Ista množica 30 točk, triangulirana z različnimi načini deljenja. Od leve proti desni: enakomerno deljenje na dva dela, deljenje območja na dva dela, enakomerno deljenje na četrtine, deljenje območja na četrtine.

Skupno nam algoritem torej ponuja štiri možnosti deljenja na podprobleme. Prvi in najpreprostejši je deljenje na polovico s pristopom enakomerne porazdelitve.

4.1.1 Deljenje na dva dela z enakomerno porazdelitvijo

Deljenje z enakomerno porazdelitvijo je najpreprostejše, če so točke že razporejene glede na os, po kateri bomo opravljali deljenje. Recimo osi, po kateri delimo točke, **sortirna os**, koordinati te osi pa **sortirna koordinata**. Razvrščanje lahko opravimo s poljubnim splošno namenskim algoritmom za razvrščanje, kot sta na primer merge-sort ali quicksort. Sam sem se odločil za uporabo vgrajene funkcije Sort(), ki se je izkazala za hitrejšo od lastnih poskusov. Ko imamo pred seboj pravilno razvrščene točke, deljenje n točk preprosto pomeni vzeti prvih $\frac{n}{2}$ točk v prvo delno triangulacijo in ostale v drugo.

Pri tem načinu deljenja je potrebno paziti le na eno stvar: če sta dve točki po sortirni koordinati enaki, ju moramo razporediti po drugi koordinati. Razlog za to pravilo je najlažje razložiti na podlagi primera.



Slika 18: Množica točk ABCDEF.

Poglejmo si množico točk na sliki 18. Recimo da točke delimo po osi x. Algoritem za razvrščanje bo zagotovo A postavil na prvo mesto (najmanjša koordinata x) in F na zadnje (največja koordinata x). Med prvo in zadnjo točko bodo sledile točke BCDE, vendar pa bojo brez upoštevanja njihovih y koordinat razporejene **naključno**. Med možnimi razvrstitvami je torej gotovo tudi naslednja:

A	B	D	E	C	F

Tabela 1: Razporeditev točk po x koordinati.

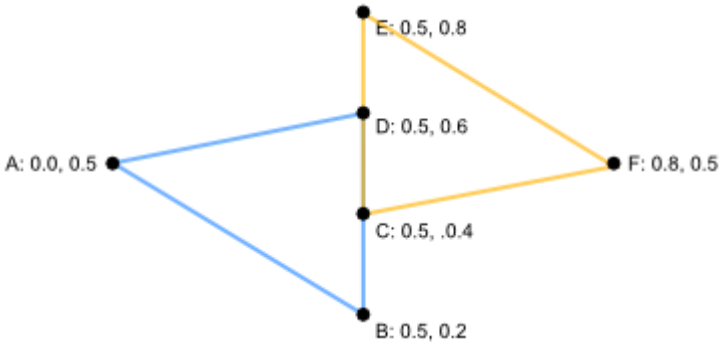
Ker se delne triangulacije izračunajo neodvisno ena od druge, bosta delni triangulaciji točk ABD in ECF izgledali kot kaže slika 19. Triangulaciji bosta ustvarili povezavi BD in CE, ki ležita na isti premici in se prekrivata na celotnem intervalu CD.

V primeru, da upoštevamo y koordinato pri razporejanju točk z isto koordinato, pa bo razporeditev točk iz slike 18 vedno enaka, in sicer:

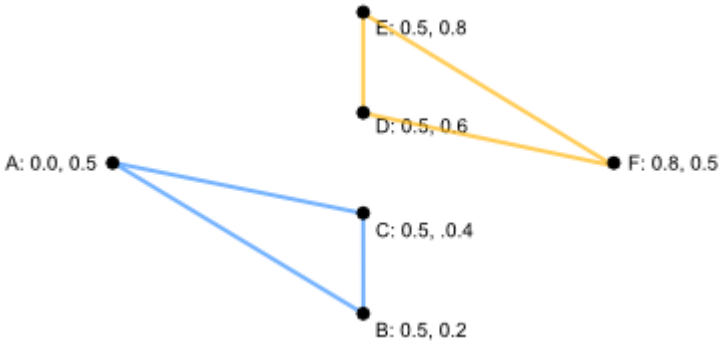
A	B	C	D	E	F

Tabela 2: Razporeditev točk po x koordinati ob upoštevanju koordinate y.

Delni triangulaciji v tem primeru ne moreta vsebovati povezave, ki bi se prekrivala s povezavo druge polovice. Rezultat za to množico točk je prikazan na sliki 20.



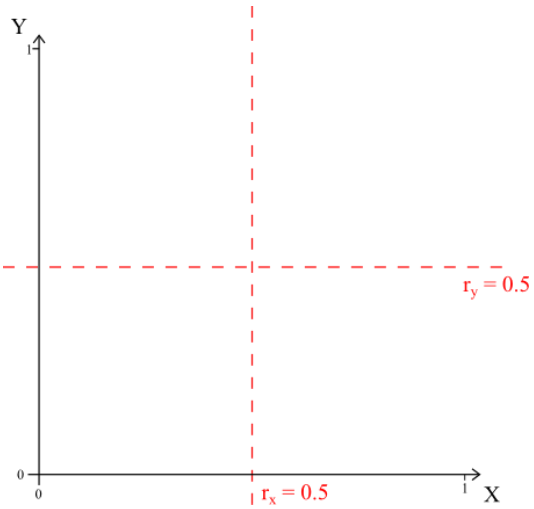
Slika 19: Nepravilni delni triangulaciji za množico točk ABCDEF.



Slika 20: Pravilni delni triangulaciji za množico točk ABCDEF.

4.1.2 Deljenje na dva dela s prepolovitvijo območja

Deljenje na dva dela s prepolovitvijo območja se od prejšnjega načina deljenja precej razlikuje po sami implementaciji. Točke moramo najprej razporediti glede sortirno os prav tako kot pri prejšnjem pristopu. Od tu dalje pa se pristop razlikuje. V vsakem koraku algoritem razdeli interval, na katerem se nahaja, na polovico. Deljenje intervala si lahko predstavljamo s premico, ki interval seka na polovici. Vse točke na levi strani premice spadajo v levo delno triangulacijo, vse na desni pa v desno delno triangulacijo. Premica sovpada s funkcijo $x = r$, ko območje delimo na osi x, oziroma $y = r$, ko območje delimo na osi y. Število r imenujmo **razpolovna koordinata**.



Slika 21: Razpolovna koordinata r_x razpolavlja os X, r_y pa os Y.

Algoritem se bo v prvem koraku vedno nahajal na intervalu [0, 1] in ga razdelil v razpolovni koordinati $r = 0.5$. Vse točke levo od te koordinate postanejo del prve delne triangulacije, vse ostale postanejo del desne. V naslednjih korakih bo interval [0, 0.5] razdelil v koordinati 0.25, interval [0.5, 1] pa v koordinati 0.75. S postopkom bo nadaljeval, dokler interval, v katerem se nahaja, ne omejuje trivialnega problema – prazne množice ali ene same točke.

Deljenje s prepolovitvijo območja ima zanimivo lastnost, ki se pri enakomernemu deljenju ne more pojaviti: deljenje lahko producira prazno množico. Pri enakomernemu deljenju je to nemogoče, saj ima najmanjša množica, ki jo še delimo, dve točki, in bo od teh vsaka šla v eno delno triangulacijo. V primeru deljenja območja pa se lahko zgodi, da so vse točke na eni strani in dobimo eno prazno množico.

Za dejansko implementacijo deljenja množice uporabimo pristop **dvojiškega iskanja**. Algoritem za dvojiško iskanje je namenjen iskanju elementov v urejeni tabeli. Deluje tako, da interval, na katerem se nahaja, razdeli na dva dela. Če je zadnji element prvega dela ali prvi element zadnjega enak iskanemu ključu, vrne algoritem njegov indeks kot rezultat in zaključi z delovanjem. V nasprotnem primeru išče dalje v prvem delu, v kolikor je zadnji element prvega dela večji od iskanega ključa, oziroma v drugem delu, če je prvi element zadnjega dela manjši od iskanega ključa.

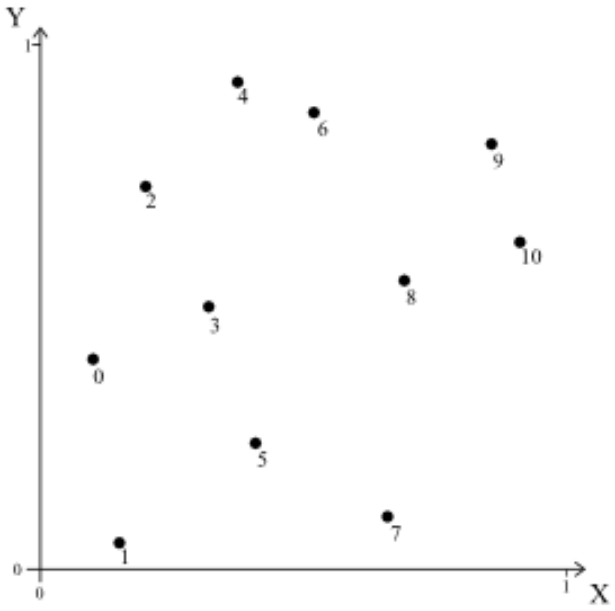
Da lahko dvojiško iskanje uporabimo v našem problemu, se moramo najprej vprašati, kaj pravzaprav iščemo. Za razliko od klasičnega dvojiškega iskanja ne iščemo dejanskega elementa (točke) v tabeli, temveč iščemo **indeks zadnje točke, ki ima sortitno koordinato še vedno manjšo ali enako razpolovni koordinati**. Algoritem zato spremenimo na sledeč način: interval, na katerem se nahaja, razdeli na dva dela. Če je prvo število desne polovice manjše ali enako razpolovni koordinati, nadaljuje v desni polovici, v nasprotnem primeru nadaljuje v levi. Algoritem se ustavi, ko je velikost intervala, na katerem se nahaja, enaka 1, in vrne indeks na katerem se nahaja. Funkcija za razpolavljanje v prvo polovico vzame vse točke do in vključno z indeksom, vse ostale pa vzame v drugo polovico.

Algoritem določi polovico intervala po formuli:

$$h = s + \frac{(e-s)}{2},$$

kjer je h indeks polovice, s indeks začetka in e indeks konca intervala. Levi pod-interval se nahaja na indeksih s do h, desni pod-interval pa na indeksih h+1 do e.

Izpeljani algoritem še preizkusimo. Vzemimo množico enajstih točk, prikazanih na sliki 22. Točke se nahajajo na intervalu [0, 1]. Če jih razvrstimo na osi x, dobimo razporeditev, prikazano na tabeli 3.

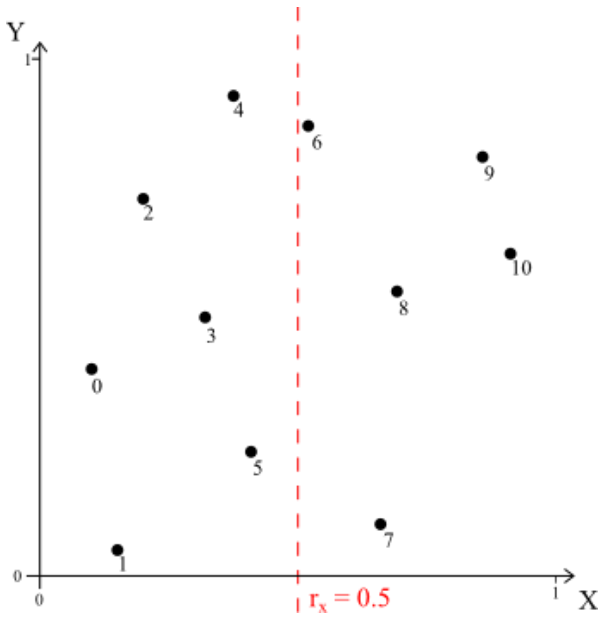


Slika 22: Naključna množica točk.

x	0.1	0.15	0.2	0.32	0.375	0.409	0.52	0.66	0.692	0.858	0.912
y	0.4	0.05	0.729	0.5	0.928	0.24	0.87	0.1	0.55	0.81	0.623
index	0	1	2	3	4	5	6	7	8	9	10

Tabela 3: Urejena množica 11 točk.

Recimo, da želimo interval razdeliti na polovico. Ker je interval na začetku [0, 1], za razpolovno koordinato vzamemo 0.5. Deljenje lahko predstavimo s sliko 21, poteka pa tako, ko prikazuje tabela 4.



Slika 23: Deljenje točk z razpolovno koordinato r_x=0.5.

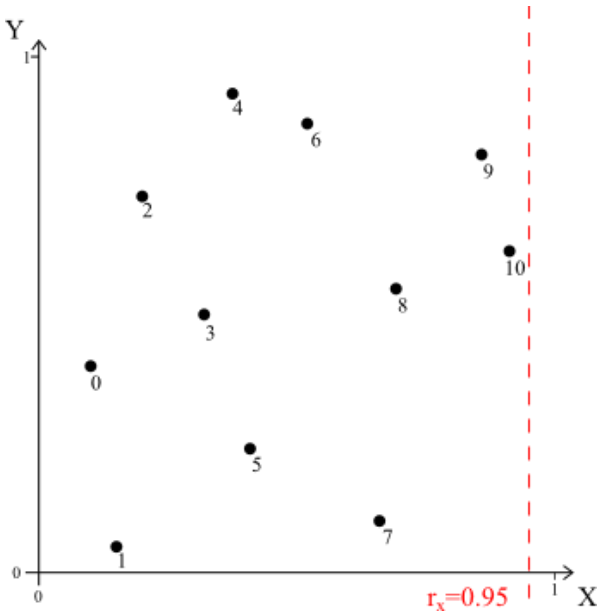
Korak	Začetek intervala	Konec intervala	Polovica intervala	število za primerjavo	Pojdi
1	0	10	5	0.52	Levo
2	0	5	2	0.32	Desno
3	3	5	4	0.409	Desno
4	5	5		KONČANO	

Tabela 4: Postopek deljenja točk z razpolovno koordinato r_x=0.5.

Algoritem vrne 5 kot indeks zadnjega števila, ki je manjše ali enako 0.5. Preizkusimo še, kako se obnaša, če so vsa števila **manjša** od razpolovne koordinate. Za razpolovno koordinato vzemimo število 0.95.

Korak	Začetek intervala	Konec intervala	Polovica intervala	število za primerjavo	Pojdi
1	0	10	5	0.52	Desno
2	6	10	8	0.858	Desno
3	9	10	9	0.912	Desno
4	10	10		KONČANO	

Tabela 5: Postopek deljenja točk z razpolovno koordinato r_x = 0.95.



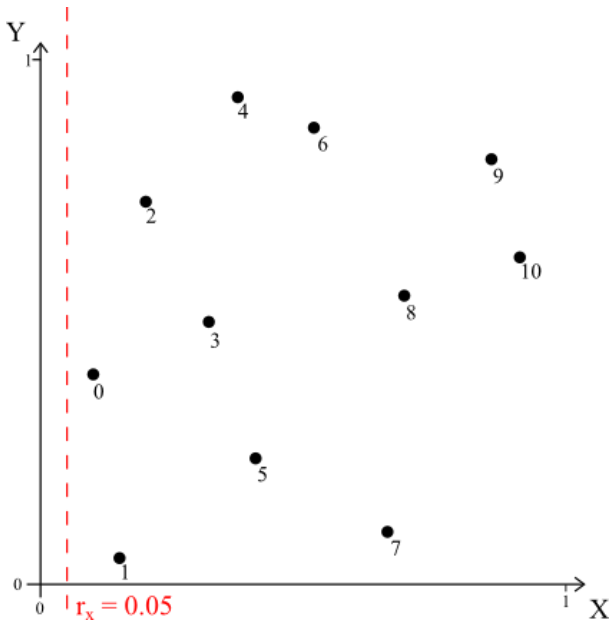
Slika 24: Deljenje točk z razpolovno koordinato r_x = 0.95.

Algoritem vrne 10, zadnji indeks tabele, ker so vsa števila v tabeli manjša od razpolovnega indeksa. Preizkusimo ga še za primer, ko so vsa števila **večja** od razpolovne koordinate. Vzemimo razpolovno koordinato 0.05.

Korak	Začetek intervala	Konec intervala	Polovica intervala	število za primerjavo	Pojdi
1	0	10	5	0.52	Levo
2	0	5	2	0.32	Levo
3	0	2	1	0.2	Levo
4	0	1	0	0.15	Levo
5	0	0	KONČANO		

Tabela 6: Postopek deljenja točk z razpolovno koordinato r_x=0.05.

Algoritem vrne indeks 0, kot zadnji indeks točke, ki ima x koordinato manjšo ali enako razpolovni koordinati 0.05. Rezultat je seveda napačen, ker nobena točka iz seznama ne ustreza pogoju – zato moramo uvesti dodatni pogoj: v primeru, da je rezultat enak 0 in je koordinata, po kateri razpolavljamo območje, za prvi element v tabeli **večja** od razpolovne koordinate, mora metoda vrniti indeks -1.



Slika 25: Deljenje točk z razpolovno koordinato r_x=0.05.

Pri deljenju s prepolovitvijo območja ni pomembno, kako razvrščamo elemente, ki imajo koordinato, po kateri razpolavljamo območje, enako. Se pa kljub temu prav v takem primeru lahko pojavi napaka. Če v množici točk obstaja par točk z isto sortirno koordinato, ju

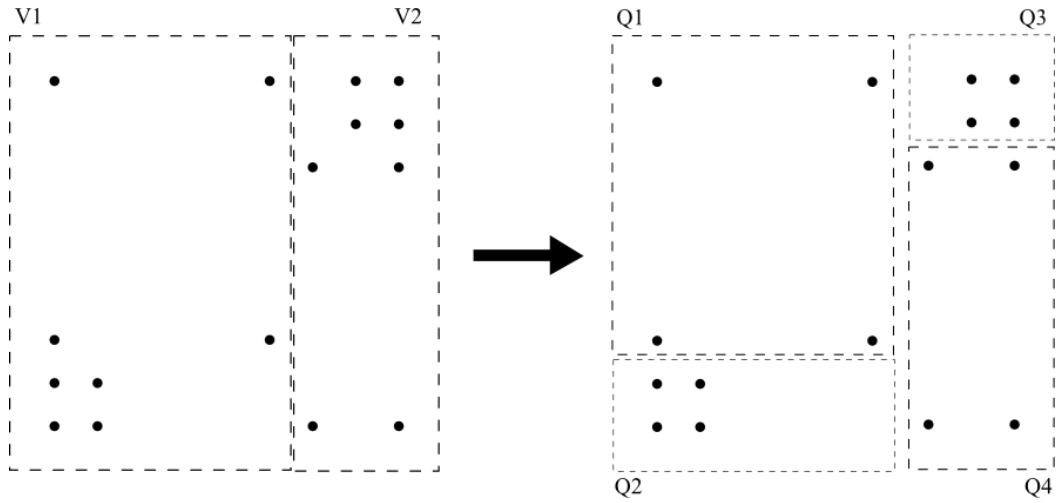
algoritem ne more razdeliti, območje pa bo kljub temu razpolavljal v nedogled, kar bi se odražalo v neskončni rekurziji. Zato moramo v algoritem za razpolavljanje uvesti dodaten pogoj: če je sortirna koordinata vseh točk v intervalu enaka, jih dalje deli po drugi koordinati. Ker so točke vedno razvrščene po sortirni koordinati, je operacija dovolj preprosta: če imata prva in zadnja točka enako sortirno koordinato, jo imajo tudi vse ostale točke.

Kljub mnogim razlikam med enim in drugim pristopom k deljenju pa velja naslednje: v velikih, enakomerno razporejenih množicah točk bo deljenje s prepolovitvijo območja produciralo **podobne** oziroma lahko celo **enake** rezultate kot deljenje z enakomerno porazdelitvijo.

4.1.3 Deljenje na četrtine z obema pristopoma delitve.

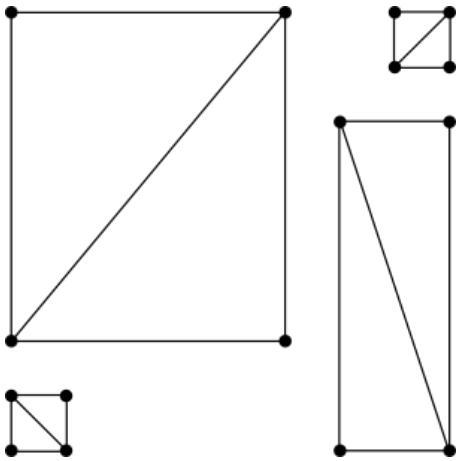
Da za deljenje na četrtine ne pišemo metod iz ničle, uporabimo metode opisane v poglavjih 4.1.1 in 4.1.2. Vsako množico in podmnožico z več kot eno točko najprej razdelimo na dve **glavni podmnožici** po osi x z uporabo prej omenjenih metod. Dobljeni podmnožici razporedimo po osi y in jih razdelimo še na tej. Pri združevanju moramo biti pazljivi, da najprej združimo podmnožici iz iste glavne podmnožice. V nasprotnem primeru lahko predvsem pri enakomernem deljenju pride do sekanja povezav med delnimi triangulacijami.

To najlažje dokažemo na podlagi primera. Vzemimo enakomerno delitev množice 16 točk, ki jo predstavlja Slika 26. V prvem koraku ustvarimo glavni podmnožici V1 in V2. Te v naslednjem koraku enakomerno razdelimo v štiri podmnožice, Q1-Q4.



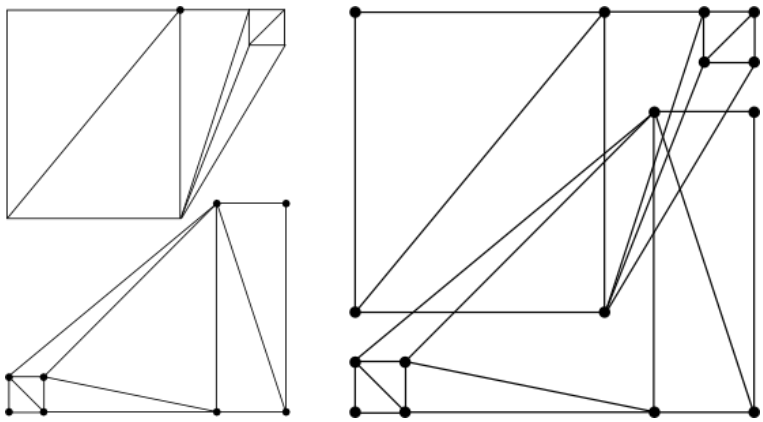
Slika 26: Enakomerna porazdelitev množice točk na četrtine.

Vsako podmnožico trianguliramo neodvisno od ostalih tako da nastane triangulacija na Slika 27.



Slika 27: Triangulacije podmnožic.

Problem lahko nastane pri navzkrižnem združevanju delnih triangulacij. V primeru, da združimo delne triangulacije znotraj njihovih lastnih glavnih množic, do sekanja ne bo prišlo, saj je algoritem ob deljenju na glavni množici že poskrbel, da sta le-ti neodvisni druga od druge. Če pa združujemo delni triangulaciji iz različnih glavnih množic, to ne nujno. Ker ob združevanju upoštevamo le povezave nastajajoče triangulacije in ne vseh povezav v globalnem pogledu (slednje bi bilo zelo časovno zahtevno), se lahko zgodi, da se nastali delni triangulaciji dejansko sekata v poljubnemu številu točk. Pojav prikazuje slika 28, v kateri je prikazano združevanje podmnožic Q1 z Q3 in Q2 z Q4.



Slika 28: Navzkrižno združevanje iz vidika posamezne nastale delne triangulacije (levo) in dejanski rezultat takega združevanja (desno).

4.2 Združevanje delnih triangulacij

Združevanje delnih triangulacij je računsko najzahtevnejši del prve faze algoritma. Vhod v fazo združevanja sta dve delni triangulaciji. Kot že omenjeno v tretjem poglavju, algoritmu za združevanje ni potrebno poznati vseh točk delnih triangulacij, niti ne vseh povezav, saj lahko

med seboj povezuje le obodne točke, pri čemer ga bodo lahko ustavile že obodne povezave. Da lahko algoritem združi delni triangulaciji, mora torej za vsako od njiju poznati vsaj naslednje podatke:

1. seznam vseh obodnih točk in
2. seznam vseh obodnih povezav.

Vsak par točk iz obodov delnih triangulacij predstavlja potencialno povezavo šiva triangulacij. Algoritem se sprehodi skozi vse take potencialne povezave in jih poskuša vstaviti v triangulacijo. Nastala triangulacija pa ni še ni pripravljena za združevanje s še eno triangulacijo – kot smo že omenili, vhod v združevanje vsebuje le ključne povezave in točke – tiste iz oboda triangulacije. Po koncu združevanja moramo torej izluščiti obod triangulacije. Miselno lahko fazo združevanja razdelimo na dve pod-fazi:

1. vstavljanje povezav in
2. izračun oboda.

Še preden pa lahko začnemo s katerokoli od njiju, moramo definirati, kako zaznamo sekanje.

4.2.1 Izračun sekanja

Da lahko algoritem ugotovi, če dano povezavo sploh sme vstaviti, mora najprej zagotoviti, da se ne seka z obstoječo povezavo triangulacije. Za par daljic e_1, e_2 ne moremo neposredno ugotoviti, ali se sekata. Lahko pa najdemo točko, v kateri se sekata premici, na katerih ležita daljici e_1 in e_1 in jo uporabimo za nadaljnje izračune. Da lahko to storimo, pa moramo najprej izračunati linearni enačbi premic.

Daljica je podana kot par točk $A(x_a,y_a)$, $B(x_b,y_b)$. Za osnovo uporabimo dvo-točkovno obliko linearne enačbe (ang. two point form):

$$y - y_a = \frac{y_b - y_a}{x_b - x_a}(x - x_a)$$

Ob tem velja omeniti, da $y_b - y_a$ ustreza y koordinati smernega vektorja premice, $AB(x_d, y_d)$, $x_b - x_a$ pa x koordinati. Ob upoštevanju te zakonitosti izgleda linearna enačba ko izrazimo y takole:

$$y = \frac{y_d}{x_d}x - x_a\frac{y_d}{x_d} + y_a$$

Obstajata 2 posebna primera. Ko je y koordinata smernega vektorja(y_d) enaka 0, je premica horizontalna črta, z enačbo oblike $y = y_a$. V primeru, da je x koordinata smernega vektorja (x_d) enaka 0, je rezultat zgornje enačbe nedefiniran, saj imamo na dveh mestih deljenje z ničlo. Razlog je v tem, da je premica vertikalna črta, torej iščemo enačbo tipa $x = n$. Če iz dvo-točkovne oblike enačbe izrazimo x, dobimo enačbo oblike:

$$x = \frac{x_d}{y_d} y - y_a \frac{x_d}{y_d} + x_a$$

Če je x koordinata smernega vektorja (x_d) enaka 0 in vstavimo podatke v zgornjo enačbo, bomo dobili enačbo oblike $x = x_a$, ki ponazarja vertikalno premico. Linearno enačbo premice zato uporabljamo v obliki $y = kx + n$, razen v primeru, ko je x koordinata smernega vektorja premice enaka 0. V tem primeru je enačba premice podana v obliki $x = n$.

Ko imamo znani linearni enačbi premic, na katerih ležita dve povezavi, lahko izračunamo njuno presečišče tako, da enačimo enačbi in izrazimo x. To storimo po naslednjem postopku:

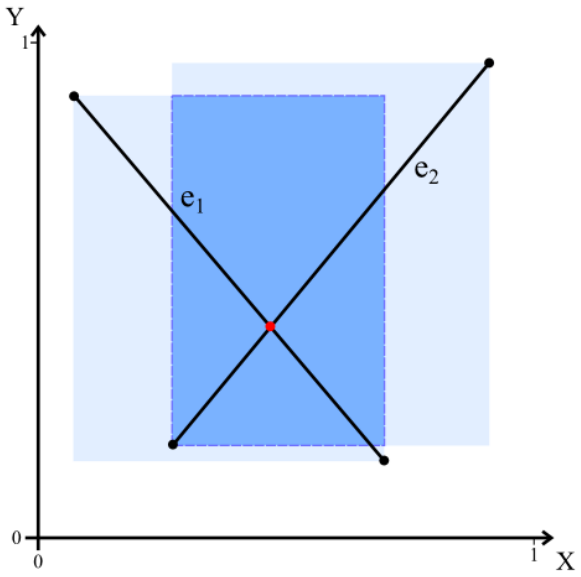
$$\begin{aligned} y &= k_1 x + n_1 \\ y &= k_2 x + n_2 \\ \hline k_1 x + n_1 &= k_2 x + n_2 \\ k_1 x - k_2 x &= n_2 - n_1 \\ x(k_1 - k_2) &= n_2 - n_1 \\ x &= \frac{n_2 - n_1}{k_1 - k_2} \end{aligned}$$

Tabela 7: Izračun x koordinate presečišča premic.

V primeru, da je ena od enačb tipa $x = n$, lahko ta korak preskočimo. V primeru, da sta obe premici vertikalni ali horizontalni, presečišča ne iščemo. Razlog za to je podrobneje predstavljen v naslednjem podpoglavju.

Ko imamo znano koordinato x, jo lahko vstavimo v katerokoli izmed enačb premic, ki ni vertikalna in dobimo koordinato y. Pridobljena točka je presečišče premic, na katerih ležita povezavi. To, da se premici sekata, pa seveda še ne pomeni, da se sekata tudi povezavi, ki ležita na njih. Vsak par ne-vzporednih premic se vedno seka v natanko eni točki. Upoštevajoč, da se nahajamo na intervalu [(0,0), (1,1)], pa lahko o sekanju povezav brez dodatnih izračunov povemo naslednje: če je katerokoli koordinata presečišča premic večja od 1 ali manjša od 0, (če se presečišče nahaja izven intervala), se povezavi ne sekata.

Povezavi **se** sekata, če izračunano presečišče premic leži na obeh daljicah. Vsako daljico, ki ni horizontalna ali vertikalna, lahko vzamemo za diagonalo pravokotnika vzporednega z osmi koordinatnega sistema. Ta pravokotnik omejuje interval, v katerem se nahaja daljica. Ker že vemo, da presečišče leži na premici, katere del je daljica, lahko predpostavimo, da leži tudi na daljici sami, v kolikor se nahaja znotraj intervala, na katerem se ta nahaja. Z drugimi besedami, daljici se sekata, v kolikor se presečišče njunih premic nahaja na preseku intervalov, na katerih se nahajata.

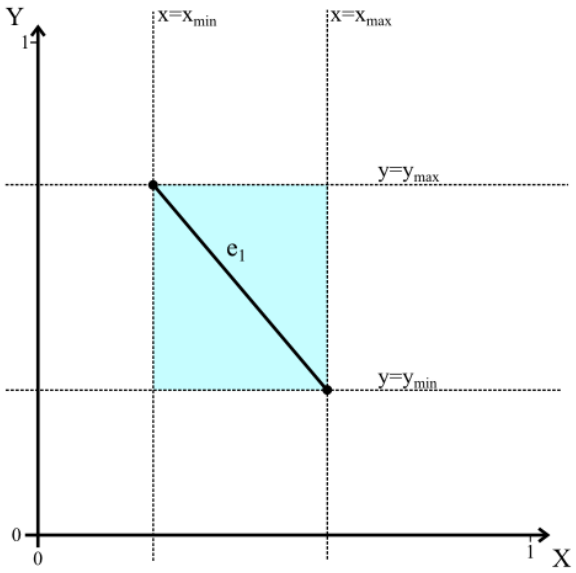


Slika 29: Presečišče premic leži na preseku intervalov, na katerih ležita daljici. Daljici se torej sekata.

V splošnem lahko za vsako daljico izpostavimo 4 lastnosti:

- najmanjša izmed x koordinat točk, ki jo označimo z x_{min},
- največja izmed x koordinat točk, ki jo označimo z x_{max},
- najmanjša izmed y koordinat točk, ki jo označimo z y_{min},
- največja izmed y koordinat točk, ki jo označimo z y_{max}.

Izpeljane koordinate so meje intervala, ki omejuje daljico. Da točka p s koordinatami x_p, y_p, ki je izračunano presečišče premic, leži na daljici, mora veljati: $x_{min} \leq x_p \leq x_{max}$ in $y_{min} \leq y_p \leq y_{max}$. Pristop deluje tudi pri horizontalni ($y_{min} = y_p = y_{max}$) ali vertikalni ($x_{min} = x_p = x_{max}$) daljici. Kljub temu pa raje uporabimo pristop, da pri horizontalnih daljicah ne preverjamo mej koordinate y in obratno pri vertikalnih ne preverjamo mej koordinate x. Razlog je nenatančnost številskega tipa Double, ki ga uporabljamo kot osnovo celotnega algoritma. Verjetnost, da se ob izračunu x ali y koordinate algoritem zmoti za majhno (v okolici 10^{-10}), a hkrati dovolj veliko število, da pogoj ne uspe, je prevelika, da bi preverjali pogoj, ki ob upoštevanju, da iskana točka zagotovo leži na premici daljice, sploh ni potreben.



Slika 30: Meje intervala, na kateri se nahaja daljica e_1.

4.2.2 Vstavljanje povezav

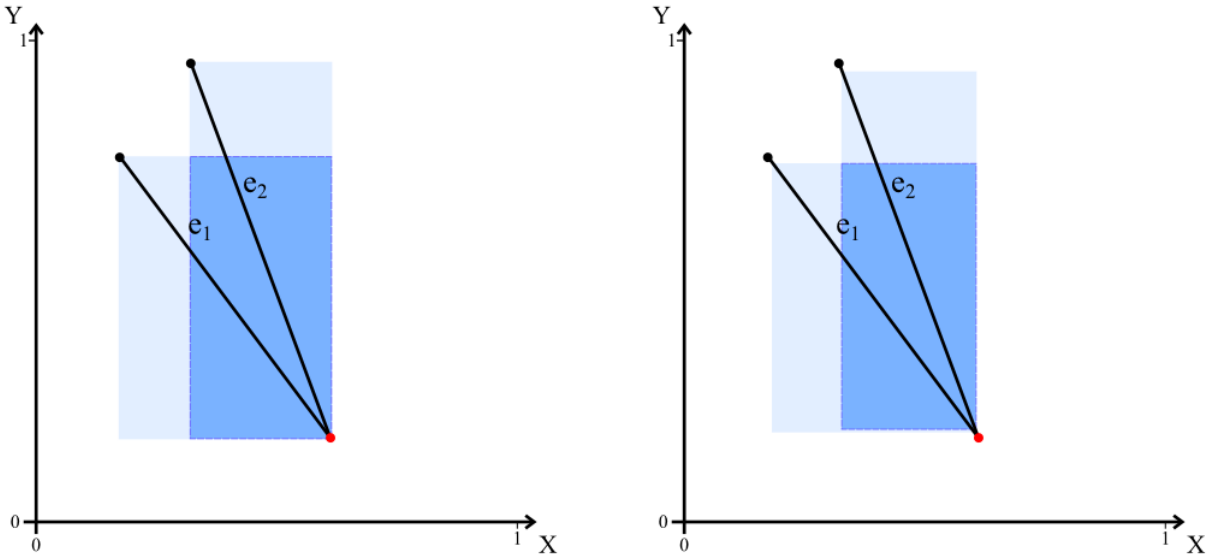
Ko imamo definiran postopek za izračun sekanja med dvema povezavama, se lahko lotimo združevanja delnih triangulacij z vstavljanjem povezav in s tem konstrukcijo šiva triangulacij. Kot smo že omenili, mora algoritem ob združevanju poznati sledeče množice podatkov:

1. množici obodnih povezav delnih triangulacij E_1 in E_2,
2. množico obodnih točk delnih triangulacij V_1 in $V_{2\ in}$
3. množico povezav šiva triangulacij S, ki je na začetku združevanja prazna

Algoritem delni triangulaciji združi tako, da se v zunanji zanki dvojne zanke sprehodi skozi vse točke $A \in V_1$, v notranji pa skozi vse točke $B \in V_1$. Za vsak tak par točk A,B skonstruira povezavo (s pripadajočo enačbo premice) in jo doda v šiv triangulacije, v kolikor se ne seka z nobeno povezavo $e_i \in S \cup E_1 \cup E_2$. Z drugimi besedami, povezava se ne sme sekati z obstoječo povezavo nastajajoče triangulacije.

Tu se pa že pojavi problem. Presečišče med premicama poljubnih povezav AB in AC, ki imata točko A skupno, bo vedno točka A. Pogoj za sekanje iz prejšnjega poglavja bi praviloma lahko spremenili iz manjše ali enako v strogo manjše, vendar zaradi že omenjene nenatančnosti številskega tipa Double to ni dovolj. Izračunano presečišče premic se od dejanskega lahko razlikuje za majhno število, ob čemer je rezultat lahko še vedno pravilen (če odmik točko premakne dlje izven preseka intervalov, na katerih se nahajata povezavi), lahko pa je napačen (če odmik točko premakne v presek intervalov povezav). Pojav imenujmo **lažno sekanje**.

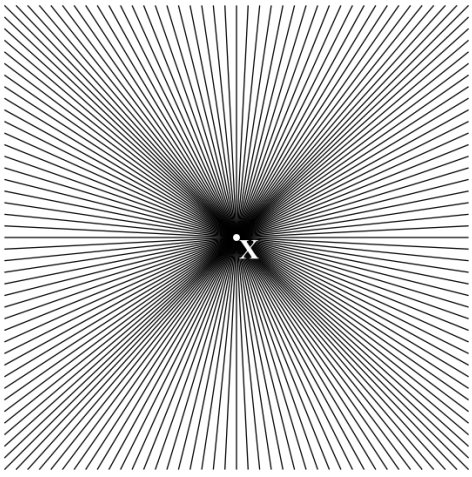
Moja prva misel je bila zmanjšati obseg intervalov na katerih se nahajata povezavi za dovolj majhno število, da se ob dejanskih sekanjih ne bi poznalo, hkrati pa ne bi prišlo do lažnih sekanj. Recimo temu številu λ. Pogoj za sekanje se tako spremeni in postane: $x_{min+}\lambda \leq x_p \leq x_{max-}\lambda$ in $y_{min+}\lambda \leq y_p \leq y_{max-}\lambda$.



Slika 31: Presek intervalov daljic skoraj vsebuje skupno točko(levo). Z zmanjšanjem intervalov (desno) zmanjšamo verjetnost, da bo prišlo do lažnega sekanja.

Žal pa s tem pristopom samo obrnemo problem, saj se bodo dogajale napake v povezavi z neodkritim sekanjem. Če se povezavi sekata v neposredni bližini točke, sekanja lahko ne odkrijemo. Namesto tega uporabimo drug pristop.

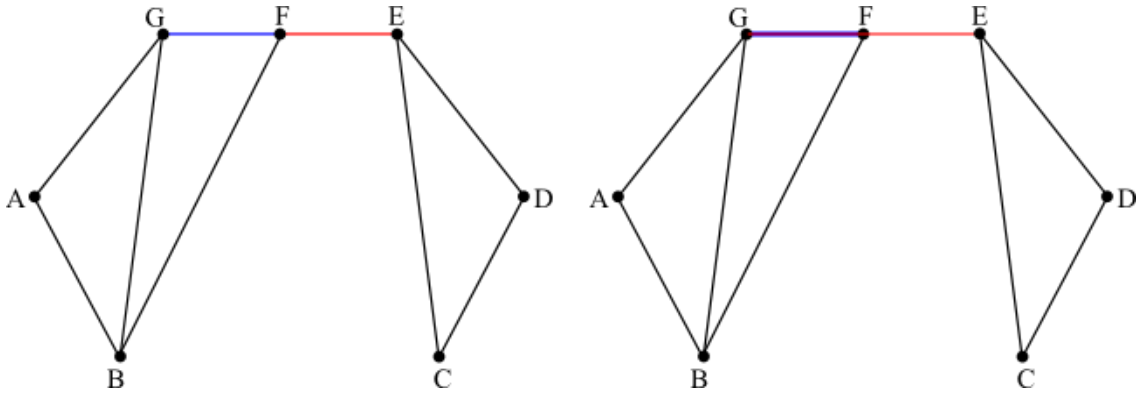
Dve premici, ki potujeta skozi isto točko, se vedno sekata v točno tej točki (slika 32). Dve različni premici imata lahko samo eno presečišče, iz česar sledi, se dve daljici z eno skupno točko ne moreta sekati. Lahko pa se prekrivata na določenemu intervalu, če ležita na isti premici.



Slika 32:Vse premice, ki potujejo skozi točko X, se sekajo v tej točki.

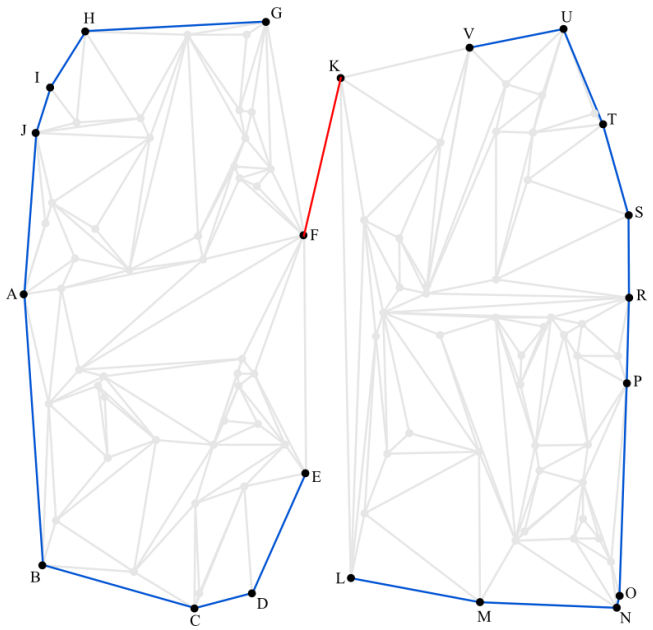
To da daljici ležita na isti premici seveda še ni zadostni pogoj za prekrivanje daljic (Slika 33). Dve daljici s skupno točko A, ki ležita na isti premici, se lahko nahajata na nasprotnih straneh točke A. V splošnem velja naslednje pravilo:

- daljici AB in AC se prekrivata na intervalu AC, če točka C leži na daljici AB, in
- daljici AB in AC se prekrivata na intervalu AB, če točka B leži na daljici AC



Slika 33: Združevanje delnih triangulacij ABFG in CDE. Daljici GF in FE ležita na isti premici a se ne prekrivata (levo). Daljici FG in EG se prekrivata na intervalu FG (desno).

Pravilo lahko uporabimo tako, da preprečimo lažen izračun sekanja: algoritem ob vstavljanju povezave AB v šiv triangulacij preverja morebitno sekanja z vsemi relevantnimi povezavami triangulacije **razen povezav, ki se začnejo ali končajo v točki A ali B**.



Slika 34: Vstavljanje povezave KF. Povezave, ki jih preverja za sekanje, so označene z modro.

Če algoritem ugotovi, da se povezava AB ne seka z nobeno povezavo, s katero je opravljal preverjanje, pa to še ne pomeni, da jo lahko vstavi v triangulacijo. Najprej pregleda vsako točko $X \in (V_1 \cup V_2) \setminus \{A, B\}$. Algoritem predpostavi, da točka X leži na daljici AB če velja, da je kot med vektorjema $\overrightarrow{XA}$ in $\overrightarrow{XB}$ enak 180°. Če tako točko X najde, povezave ne vstavi v šiv triangulacije.

Za izračun kota med daljicami uporabimo skalarni produkt vektorjev $\overrightarrow{XA}$ in $\overrightarrow{XB}$. Iz formul izrazimo kosinus kota med vektorjema na sledeč način:

$$\overrightarrow{XA}(x_a, y_a), \overrightarrow{XB}(x_b, y_b)$$
$$\overrightarrow{XA} \cdot \overrightarrow{XB} = x_a * x_b + y_a * y_b$$
$$\overrightarrow{XA} \cdot \overrightarrow{XB} = \left|\overrightarrow{XA}\right| * \left|\overrightarrow{XB}\right| * \cos\varphi$$
$$x_a * x_b + y_a * y_b = \left|\overrightarrow{XA}\right| * \left|\overrightarrow{XB}\right| * \cos\varphi$$
$$\cos\varphi = \frac{x_a * x_b + y_a * y_b}{\left|\overrightarrow{XA}\right| * \left|\overrightarrow{XB}\right|}$$

Kosinus 180° kota je enak -1. Povezava AB se prekriva z obstoječo povezavo na intervalu AX ali BX, če obstaja taka točka X, da je kosinus kota med daljicama AX in BX enak -1. Ker pa je celoten razlog za nov pristop nenatančnost številskega tipa Double, je dovolj če je kosinus kota približno enak -1. Z drugimi besedami, algoritem povezave AB, ki se ne seka z obstoječo povezavo triangulacije, ne vstavi v šiv triangulacij, če obstaja taka točka X, da za kot φ med daljicama AX in BX,velja: $|\cos\varphi + 1| \leq \lambda$, kjer je λ največji dovoljeni odmik od dejanske vrednosti. Pravkar opisani pristop se uporablja za obravnavanje kolinearnosti v vseh delih algoritma.

Uporaba tega pristopa je tudi razlog, da v primerih, ko so obe premici horizontalni ali vertikalni, ne obravnavamo sekanja. Premici sta v tem primeru zagotovo vzporedni, ne vemo pa še, ali sta isti. Da zagotovimo konsistentnost algoritma, take primere preverimo s pravkar izpeljano metodo.

Končni algoritem za združevanje delnih triangulacij je predstavljen spodaj.

```
private void Merge() {
   foreach (Point p1 in V1) {
       foreach (Point p2 in V2) {
           //vsak par točk predstavlja potencialno povezavo triangulacije.
           LineSegment candidate = new LineSegment(p1, p2);
           bool doable = true;

           for (int i = 0; i < allConnections.Count; i++) {
               line = allConnections[i];

               /*Če je si nova povezava deli točko z obstoječo povezavo,
                 bo preverjanje opravljeno kasneje*/
               if (line.isBoundedBy(p1) || line.isBoundedBy(p2)) {
                   continue;
               }
               //presečišče premic je točka
               Point intersect = line.lineEquation.equals(candidate.lineEquation);

               /*če presečišče premic leži na obeh povezavah, lahko s
                 preverjanjem prenehamo, algoritem povezave ne bo vstavil*/
               if (!candidate.contains(intersect) && line.contains(intersect)) {
                   doable = false;
                   break;
               }
           }
           //Preveri še, če obstaja točka, ki leži na predlagani povezavi
           if (doable) {
               foreach (Point x in points) {
                   if (!candidate.isBoundedBy(x)) {
                       //Če so točke A,X,B kolinearne, povezave ne smemo vstaviti
                       if (Collinear(temp.a, x, temp.b)) doable = false;
                   }
               }
           }
           if (doable) Add(temp); //točko dodaj v triangulacijo.
       }
   }
```

Algoritem 5: Metoda za združevanje delnih triangulacij..

4.2.3 Izračun oboda triangulacije

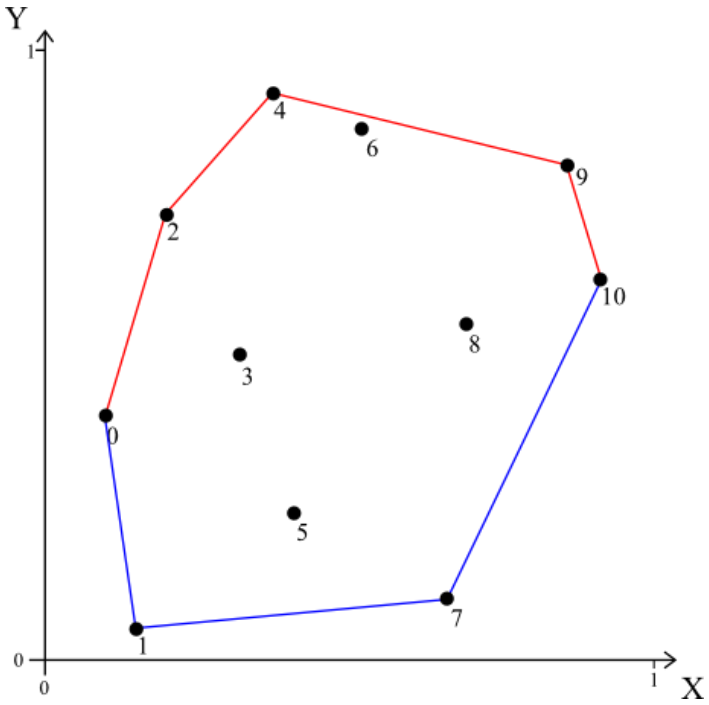
Kot smo že omenili v poglavju 3, je pri združevanju triangulacij T1 in T2 preizkušanje vseh povezav z izvorom v T1 in ponorom T2 izjemno časovno potratno, predvsem pa popolnoma nepotrebno. Isto velja tudi za preverjanje sekanja – iskanje morebitnega sekanja z vsemi povezavami triangulacije hitro postane ozko grlo algoritma in že razmeroma majhne množice točk (~3000) lahko algoritmu vzamejo več kot uro.

Ker tega nočemo (osnovno triangulacijo bi se moralo izračunati hitreje, kot se jo izboljša) algoritem po vsakem končanem združevanju poišče obod nastale triangulacije in iz triangulacije odvzame vse točke in povezave, ki ne ležijo na njem. Da najdemo obod pa uporabimo koncept konveksne ovojnice.

Konveksna ovojnica množice točk Q je najmanjši konveksni poligon P, za katerega velja da vse točke množice Q ležijo na obodu ali znotraj ploščine P [5]. Konveksno ovojnico si lahko predstavljamo z analogijo elastike. Če elastiko raztegnemo okoli množice točk in jo nato spustimo, so točke konveksne ovojnice tiste, ki držijo in napenjajo elastiko. Obstaja več

algoritmov za izračun konveksne ovojnice množice točk, zaradi lahke implementacije in določenih dobrih lastnosti, ki bodo predstavljene v nadaljevanju, sem se odločil za implementacijo algoritma monotone verige (ang. Andrew's monotone chain).

Preden govorimo o samem delovanju algoritma, predstavimo koncept zavoja. Za primer vzemimo 3 točke, A, B in C in narišimo smerna vektorja AB in AC. S p označimo premico, na kateri leži daljica AB. Zaradi pravila desne roke je Z koordinata vektorskega produkta vektorjev $\overrightarrow{AB}$ in $\overrightarrow{AC}$ manjša od 0, če se točka C nahaja desno od premice p, in večja od 0, če se točka nahaja levo od premice p. Če je Z koordinata vektorskega produkta $\overrightarrow{AB} \times \overrightarrow{AC}$ enaka 0, so točke kolinearne. Iz tega sledi naslednje pravilo v povezavi z zavojem, ki ga napravita daljici AB in BC: Če se točka C nahaja levo od premice p, potem povezavi AB BC napravita zavoj v nasprotni smeri urinega kazalca. Nasprotno povezavi AB BC napravita zavoj v smeri urinega kazalca, če se točka C nahaja desno od premice p. Pristop je uporabljen v algoritmu monotone verige, kasneje pa ga uporabljamo za izboljševanje triangulacije.



Slika 35: Algoritem monotone verige. Modre povezave prikazujejo spodnjo verigo, rdeče zgornjo.

Algoritem monotone verige konveksno ovojnico izračuna v dveh delih: spodnja veriga in zgornja veriga [6]. Vhod v algoritem je po osi x urejena množica točk (v primeru, da imata dve točki isto koordinato x, sta urejeni po osi y). Algoritem se sprehodi od prve do zadnje točke in jih dodaja v seznam. Če sta v seznamu več kot dve točki in zadnje 3 točke naredijo zavoj v smeri urinega kazalca, iz seznama odstrani predzadnjo točko. Rezultat je spodnja veriga konveksne ovojnice. Nato se sprehodi od zadnje do prve točke in z uporabo istega

principa poišče zgornjo verigo ovojnice. Končni rezultat algoritma je seznam točk, kot si sledijo na ovojnici od prve do prve v krogu v nasprotni smeri urinega kazalca. Sam algoritem je prikazan v nadaljevanju.

```
public List<Point> ConvexHull(List<Point> entryPoints) {

    Point[] hull = new Point[2 * entryPoints.Count];
    Double tmp;

    int n = entryPoints.Count, k = 0;

    //spodnja veriga
    for (int i = 0; i < n; i++) {
        /*Briši predzadnjo točko, dokler zadnje 3 točke ne naredijo zavoja v nasprotni
          smeri urinega kazalca, ali dokler ne ostanejo samo 2 točki.*/
        while (k >= 2 && (crossProduct(hull[k - 2], hull[k - 1], entryPoints[i])) <= 0){
            k--;
        }
        hull[k] = entryPoints[i];
        k++;
    }
    //zgornja veriga
    for (int i = n - 2, t = k + 1; i >= 0; i--) {
        /*Briši predzadnjo točko, dokler zadnje 3 točke ne naredijo zavoja v nasprotni
         smeri urinega kazalca, ali dokler ne ostanejo samo 2 točki.*/
        while (k >= t && (crossProduct(hull[k - 2], hull[k - 1], entryPoints[i])) <= 0){
            k--;
        }
        hull[k] = entryPoints[i];
        k++;
    }
    //prva točka se ponovi dvakrat
    Hull[k]=entryPoints[0];
}
```

Algoritem 6: Konveksna ovojnica.

Seznam točk konveksne ovojnice sovpada z obodom triangulacije, če so vse točke v splošnem položaju. Ko ima pred seboj seznam točk oboda, lahko algoritem odstrani vse povezave AB, pri katerih točki A in B nista neposredni sosedi na seznamu obodnih točk.

Ker pa je konveksna ovojnica minimalna množica, v primeru, da obstaja točka X, ki leži na eni od povezav konveksne ovojnice (torej točke niso v splošnem položaju), X ne bo vsebovana v ovojnici, čeprav vemo, da je del oboda. Algoritem se lahko spremeni, da je zavoj neveljaven le, če je z koordinata strogo manjša od 0 (torej dovoli kolinearne točke). Vendar pa lahko to prinese različne rezultate od dejanskega stanja povezav. Ker pa smo v prejšnjem podpoglavju že definirali formulo za kolinearnost, jo lahko uporabimo tudi tukaj. S tem tudi zagotovimo konsistentnost čez celoten algoritem. Pogoj je torej spremenjen: če imamo na seznamu več kot dve točki in zadnje 3 točke na seznamu niso kolinearne ***in*** naredijo zavoj v smeri urinega kazalca, iz seznama izbrišemo predzadnjo točko.

Da zmanjšamo število potrebnih izračunov kota pa algoritem raje pustimo v svoji originalni obliki. Če med dvema sosednjima točkama AB iz konveksne ovojnice ne obstaja povezava

vzamemo vse točke med in vključno z AB in del verige, na kateri se nahajata, izračunamo znova, tokrat z upoštevanjem kolinearnosti.

S tem je zaključeno poglavje o združevanju delnih triangulacij. Sledi algoritem za postopno približevanje triangulaciji z najmanjšo dolžino povezav.

4.3 Postopno izboljševanje triangulacije

Kot že večkrat omenjeno, triangulacija, ki jo izračuna algoritem opisan v poglavjih 3.1 in 4.2, po vsej verjetnosti ne bo izražala želenih lastnosti. Zato ima algoritem drugo fazo: postopno izboljševanje rezultata z izmenjevanjem povezav 1 za 1. Da algoritem ugotovi, katere povezave lahko izmenja, poišče vse konveksne štirikotnike, ki imajo v notranjosti daljšo izmed možnih diagonal, in nato vsem zamenja diagonalo z drugo možno. Algoritem lahko razdelimo na dve pod-fazi in sicer:

1. osnovna analiza v prvem krogu in
2. izmenjava diagonal in omejena ponovna analiza.

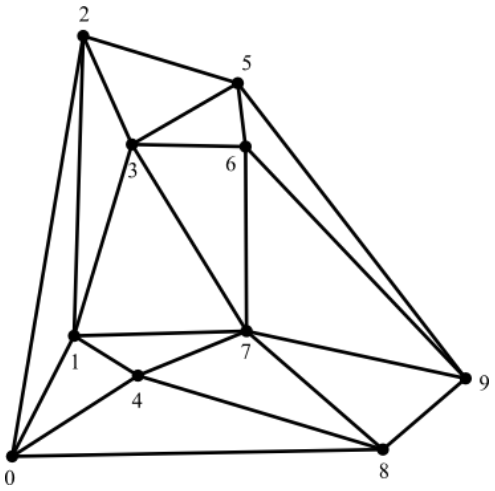
Algoritem omogoča dva različna načina izboljševanja: izboljševanje končne triangulacije in sprotno izboljševanje vsake delne triangulacije. Optimiziran algoritem za osnovno analizo pri sprotnemu izboljševanju delnih triangulacij je predstavljen v zadnjem podpoglavju. Najprej pa si poglejmo objekt, s katerim shranjujemo stanje triangulacije – seznam sosednosti.

4.3.1 Seznam sosednosti

V prejšnjem poglavju smo omenili, da algoritem po vsaki združitvi odstrani vse povezave, ki niso obodne. Končnemu rezultatu pa bi težko rekli triangulacija, če bi vseboval le obodne povezave največjih delnih triangulacij in povezave šiva. Ne le to, tako »triangulacijo« bi težko kaj izboljšali. Zato velja omeniti naslednje: algoritem res ne shranjuje objekta tipa povezava (ki vsebuje dva objekta tipa točka, dolžino, enačbo premice….) za druge povezave kot obodne. To bi bilo potratno, sploh pa bi si z njimi težko pomagali. Namesto tega ima vsaka triangulacija zgrajen svoj **seznam sosednosti**.

Seznam sosednosti je način predstavitve vseh točk in povezav v grafu. Načeloma gre za tabelo velikosti n, kjer je n število vseh točk (V) v grafu, ki na vsakem mestu vsebuje seznam točk. Seznam v tabeli na mestu i vsebuje vse točke, s katerimi je povezana točka V_i.

Ker vsak indeks tabele sovpada z natančno eno točko grafa, je seznam sosednosti lahko implementiran izključno na podlagi indeksov. Povezavo med točkama na indeksih 0 in 7 v tabeli lahko označimo kot 0-7. V seznamu sosednosti za neusmerjen graf se to odraža tako, da seznam v tabeli na indeksu 0 vsebuje število 7 kot indeks točke, s katero je povezana točka 0, in enako seznam na indeksu 7 vsebuje število 0.



Slika 36:Naključna triangulacija 10 točk.

0:	1,2,4,8
1:	0,2,3,4,7
2:	0,1,3,5
3:	1,2,5,6,7
4:	0,1,7,8
5:	2,3,6,9
6:	3,5,7,9
7:	1,3,4,6,8,9
8:	0,4,7,9
9:	5,6,7,8

Tabela 8: Seznam sosednosti za triangulacijo iz slike 36.

Potrebno pa je omeniti, da moramo ob združevanju delnih triangulacij združiti tudi njuna seznama sosednosti. Po tem bo v primeru, da smo izvajali deljenje na dva dela, seznam pravilen do tam, do koder segajo točke prve polovice, indeksi druge polovice pa bodo nepravilni. Zato uporabimo naslednji princip. Razred triangulacija hrani seznam vseh točk, ki so podane kot vhod v algoritem, razporejenih po X osi. Za vsako točko lahko potem definiramo dva indeksa. **Globalni indeks** je indeks točke v seznamu vseh povezav. **Lokalni indeks** je indeks točke v seznamu sosednosti triangulacije, ki ji točka pripada.

Ker je seznam vseh točk urejen, lahko globalni indeks točke najdemo v logaritemskem času. Seznam sosednosti lahko zato implementiramo kot slovar, ki uporablja globalne indekse točk tako za ključe kot za vrednosti v seznamih. Ker pa je lahko iskanje velikem slovarju lahko dolgotrajno, namesto tega uporabimo namenski objekt, ki hrani globalni indeks pripadajoče točke in seznam globalnih indeksov za vse točke, s katerimi je točka povezana. Seznam takih

objektov lahko urejamo glede na globalni indeks in z uporabo dvojiškega iskanja najdemo lokalni indeks točke z določenim globalnim indeksom v logaritemskem času.

Ob vsakem združevanju delnih triangulacij bo algoritem torej združil seznama sosednosti in ju ponovno uredil. Ker sta pod-seznama že urejena, je to najhitreje storjeno v le enem (zadnjem) koraku merge-sort algoritma. Za vsako dodano povezavo mora poiskati globalna in lokalna indeksa za obe točki ter jo nato zabeleži v seznam.

4.3.2 Osnovna analiza

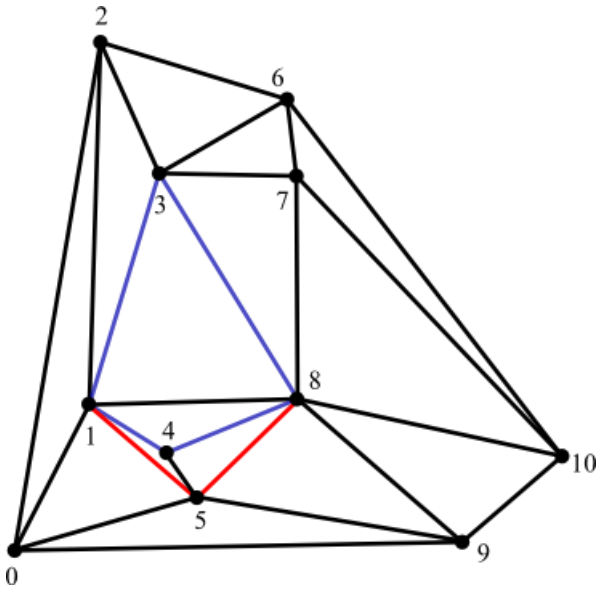
V fazi osnovne analize algoritem gradi seznam vseh štirikotnikov Q, ki jim lahko zamenja diagonalo in s tem zmanjša skupno dolžino povezav triangulacije. Preden lahko razložimo postopek analize, pa si moramo znova pogledati pomembno lastnost povezav v triangulaciji.

Vzemimo povezavo AB iz notranjosti triangulacije, ki leži na premici *p*. Povezava AB na vsaki strani premice *p* omejuje lice triangulacije. Ker je množica triangulirana sta obe lici, ki ju omejuje AB, trikotnika. Ob upoštevanju tega dejstva lahko vsak par notranjih lic triangulacije, ki se stikata v daljici AB, vzamemo za štirikotnik z diagonalo AB.

V primeru, da daljica AB leži na obodu triangulacije, je eno izmed lic, ki ju omejuje, zunanje ali neskončno lice. Iz tega sledi, da obodna povezava na more biti diagonala štirikotnika v triangulaciji.

Če vzamemo množico vseh točk X, s katerimi sta povezana tako A kot B, lahko z gotovostjo trdimo, da točka X_1 z najmanjšo vsoto dolžin povezav $AX + BX$ omejuje lice triangulacije. Istega zaradi primera na sliki 37 ne moremo trditi tudi za točko z drugo najmanjšo vsoto dolžin povezav, lahko pa trdimo, da točka z najmanjšo vsoto dolžin povezav, ki se nahaja na **drugi strani premice *p*** kot točka X_1, prav tako omejuje lice triangulacije.

Če uporabimo izpeljani princip, lahko vsako povezavo v triangulaciji z izjemo obodnih obravnavamo kot **potencialno diagonalo štirikotnika**. Algoritem se v dvojni zanki sprehodi po seznamu sosednosti, ki ga je zgradil prvi del algoritma, in za potencialno diagonalo štirikotnika vzame vsako povezavo AB, kjer sta A in B globalna indeksa pripadajočih točk, v kolikor je A manjši ob B. Razlog tiči v tem, da je seznam sosednosti urejen in je torej algoritem vsako povezavo AB kjer je B manjši od A že preveril, ko je iskal povezave z izvorom v točki z indeksom B.



Slika 37: Najbližja skupna povezana točka daljici 1-8 je točka 4. Druga najbližja je točka 5, iščemo pa točko 3.

Ostali dve točki štirikotnika dobimo tako, da preiščemo seznam sosednosti na mestih A in B in vse točke, s katerimi sta povezana oba, vstavimo v seznam skupnih točk X, kjer $\{X_0 \ldots X_n\} \in X$. Če sta A in B povezana z le eno skupno točko, lahko algoritem nadaljuje z naslednjo povezavo (točki imata praviloma lahko le eno skupno sosedo samo, če omejujeta povezavo oboda). V nasprotnem primeru dobljene točke razporedimo glede na vsoto dolžin povezav $\overline{X_1A}$ in $\overline{X_1B}$. Prva točka iz seznama, X_0, postane točka C štirikotnika ABCD. Ko imamo znano točko C, lahko izračunamo Z-indeks vektorskega produkta med vektorjema AB in AC – recimo mu Z_1. Ta nam pove smer, v kateri se nahaja točka C glede na daljico AB. Po tem se sprehodimo po urejenem seznamu točk X in vzamemo prvo točko X_k, za katero velja, da ima Z-koordinata vektorskega produkta med $\overrightarrow{AB}$ in $\overrightarrow{AX_k}$ nasproten predznak kot Z_1. Ta postane točka D štirikotnika ABCD.

Da lahko štirikotnik ABCD sploh uporabimo za zmanjšanje dolžine povezav, morajo zanj veljati naslednje lastnosti:

- **Povezava CD mora biti krajša od povezave AB**. V nasprotnem primeru bi bilo povezavi nesmiselno zamenjati, saj bi s tem povečali skupno dolžino povezav. Dolžino povezave določimo po formuli $d = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2}$.
- **Točke CAB in CBD ne smejo biti kolinearne**. To je prvi izmed pogojev ki zagotovi, da je štirikotnik **strogo konveksen**. Če pogoj ne velja, nova diagonala leži na isti premici kot dve izmed povezav oboda in je torej ne moremo vstaviti. Za izračun kolinearnosti uporabimo metodo ki je predstavljena v poglavju 4.2.2.

- **Štirikotnik mora biti konveksen**. Za preverjanje konveksnosti uporabimo algoritem za izračun konveksne ovojnice iz poglavja 4.2.3, in sicer tako, da ga poženemo na množici točk ABCD – če nam vrne 4 točke ali manj (prva točka se v rezultatu ponovi na zadnjem mestu, torej so na ovojnici le 3 izmed 4 vhodnih točk), niso vse v konveksni poziciji, torej je štirikotnik konkaven.

V poglavju 3.1 pa smo omenili tudi to, da pri vsaki zamenjani povezavi obstaja možnost, da nekateri iz štirikotnikov, ki smo jih našli v analizi, ne obstajajo več. To se zgodi zato, ker je diagonala enega štirikotnika obodna stranica drugega. Ob zamenjavi diagonale prvega štirikotnika drugi štirikotnik ne obstaja več. Ker pa je preverjanje vseh štirikotnikov iz analize po vsaki zamenjani diagonali izredno časovno potratno, uporabimo drugo metodo.

Predpostavimo, da je seznam Q urejen glede na indeksa točk diagonale. V takem seznamu lahko za obodno povezavo štirikotnika z uporabo dvojiškega iskanja preverimo, če je že vstavljena v seznam kot diagonala obstoječega štirikotnika. Operacijo moramo ponoviti 4x, enkrat za vsako obodno povezavo (AC, CB, BD, AD), torej je skupna časovna zahtevnost O(4 * log n), kar se še vedno dobro obnese tudi pri večjih vhodnih množicah. V primeru, da je ena izmed obodnih povezav diagonala obstoječega štirikotnika v seznamu, novi štirikotnik preprosto preskočimo.

Šele ko štirikotnik prestane vse preizkuse, ga vstavimo v seznam Q. Da seznama ne ponovno urejamo vsakič, ko vstavimo nov štirikotnik, namesto tega z podobnim pristopom dvojiškega iskanja, kot smo ga uporabili v deljenju triangulacije, poiščemo mesto, na katerega lahko vstavimo nov štirikotnik, da bo seznam še vedno urejen.

4.3.3 Zamenjava diagonal in omejena ponovna analiza triangulacije

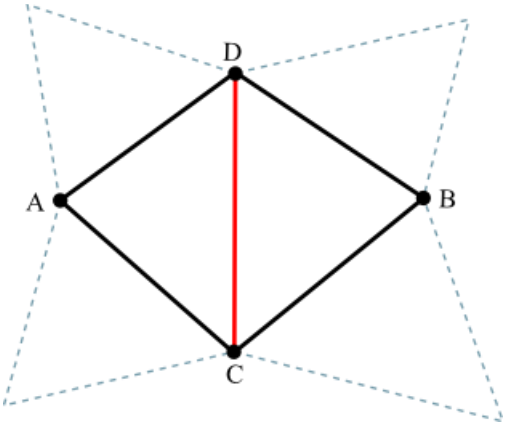
Ko imamo pred seboj seznam vseh štirikotnikov z izmenljivo diagonalo, je sama zamenjava zelo preprost postopek. Algoritem iz seznama Q odstrani prvi štirikotnik ABCD, kjer je daljica AB diagonala štirikotnika. Nato iz seznama sosednosti na indeksu točke A odstrani točko B in na indeksu točke B odstrani točko A. Nasprotno točki C in D doda drugo drugi v seznam sosednosti.

Kot pa smo dokazali v poglavju 3.1, lahko vsaka zamenjana diagonala ustvari nov štirikotnik z zamenljivo diagonalo. Ponovitev celotnega postopka analize, opisanega v prejšnjem podpoglavju, je izredno počasna in povsem nepotrebna. O pravkar vstavljeni povezavi vemo to, da je najmanjša možna diagonala znotraj svojega lastnega štirikotnika. Lahko pa omejuje drug štirikotnik, za katerega to ne velja.

V omejeni ponovni analizi triangulacije torej iščemo vse štirikotnike s pravkar vstavljeno povezavo na obodu. Postopek moramo ponoviti **po vsaki zamenjani diagonali**, sam postopek pa je zelo podoben postopku analize.

Če uporabimo notacijo iz prejšnjega podpoglavja, smo z zamenjavo diagonale štirikotnik ABCD spremenili na tak način, da je nova diagonala daljica CD. Ta štirikotnik ABCD deli na dva trikotnika, ACD in CDB. Če vsakega od njiju vzamemo za potencialno polovico štirikotnika z izmenljivo diagonalo, nam to da skupno 4 možnosti, kot prikazuje slika 38:

1. Štirikotnik ACDX z diagonalo AC,
2. Štirikotnik ACDX z diagonalo AD,
3. Štirikotnik CDBX z diagonalo CB in
4. Štirikotnik CDBX z diagonalo DB.



Slika 38: Vsi možni štirikotniki, ki imajo pravkar vstavljeno povezavo CD na obodu.

Pri tem je X točka, ki jo moramo najti za vsakega od navedenih štirikotnikov. Daljice AC, AD, CB in DB so obodne povezave štirikotnika, ki smo mu zamenjali diagonalo, iz česar sledi da je ob zamenjavi diagonale štirikotnika vsaka njegova obodna povezava potencialno diagonala štirikotnika z izmenljivo diagonalo.

Od tu je postopek skoraj enak kot pri osnovni analizi: algoritem za vsako izmed obodnih povezav $d_i \in \{AC, AD, CB, DB\}$ poišče vse točke, do katerih imata povezavo obe točki, ki povezavo d_i omejujeta. Ker že vemo, da tista izmed točk C in D, ki ne omejuje povezave d_i, skupaj z le-to omejuje prvo polovico iskanega štirikotnika, moramo le še poiskati točko, ki se nahaja na nasprotni strani povezave d_i. Postopek po katerem ugotovimo, če je štirikotnik uporaben, je isti kot v osnovni analizi.

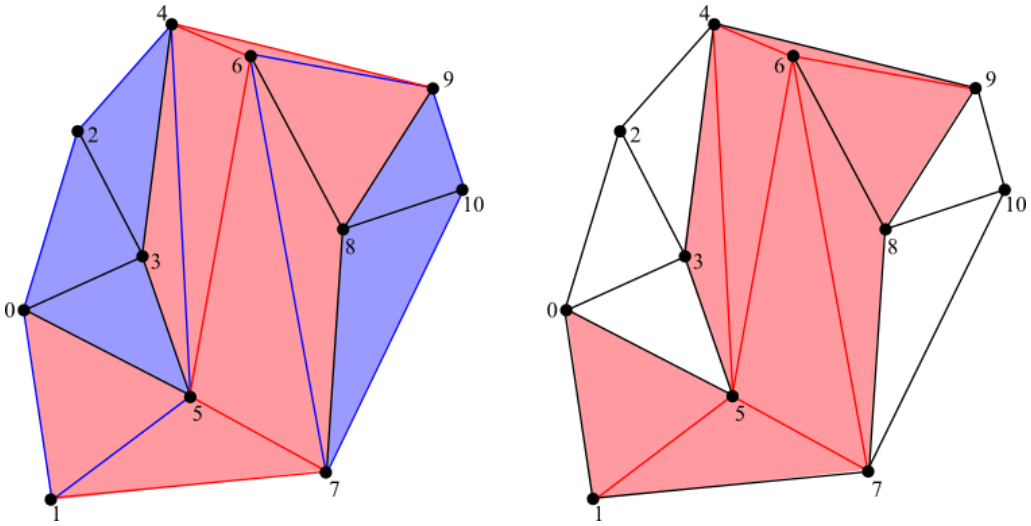
Dobljene štirikotnike vstavljamo v urejen seznam Q, pri čemer spet pazimo, da katera izmed obodnih povezav ni že vstavljena kot diagonala. Poseben algoritem za ponovno analizo je pomemben predvsem zato, da zmanjšamo količino potrebnega iskanja.

Ker se seznam štirikotnikov Q avtomatsko dopolnjuje z vsako zamenjano diagonalo, je drugi del algoritma samo-vzdržujoč. Zaključi se šele, ko na seznamu Q ni več niti enega štirikotnika.

4.3.4 Optimizacija algoritma za osnovno analizo pri izboljševanju delnih triangulacij

Omenili smo, da algoritem omogoča dva načina izboljševanja triangulacije – izboljševanja končnega rezultata in postopno izboljševanje vsake delne triangulacije. Algoritem za analizo iz poglavja 4.3.2 bo sicer že v svoji originalni obliki lahko opravil tudi s drugim primerom, vendar pa bo moral zelo veliko časa zapravljati na nepotrebnem iskanju štirikotnikov.

Če uporabljamo algoritem za postopno izboljševanje delnih triangulacij, potem ob združevanju dveh delnih triangulacij vemo, da sta le-ti že optimizirani. Ne-optimiziran je kvečjemu tisti del triangulacije, kjer leži **šiv**. Štirikotniki, ki nimajo na obodu ali kot diagonale vsaj ene izmed novih povezav šiva, so bili že preverjeni in optimizirani v delnih triangulacijah.



Slika 39: Združeni delni triangulaciji 0-5 in 6-10. Z modro so označena področja, kjer iskanje ni potrebno (levo). Rdeče povezave na sliki desno prikazujejo potencialne diagonale štirikotnikov z izmenljivo diagonalo.

Iz tega sledi, da bomo v osnovni analizi po združitvi dveh optimiziranih delnih triangulacij našli vse štirikotnike z izmenljivo diagonalo, če za potencialno diagonalo vzamemo vse **povezave šiva, ki ne ležijo na obodu nastale triangulacije** (obodne povezave navzven ne omejujejo trikotnika) in vse **povezave iz starih obodov delnih triangulacij, ki niso na obodu** trenutne triangulacije.

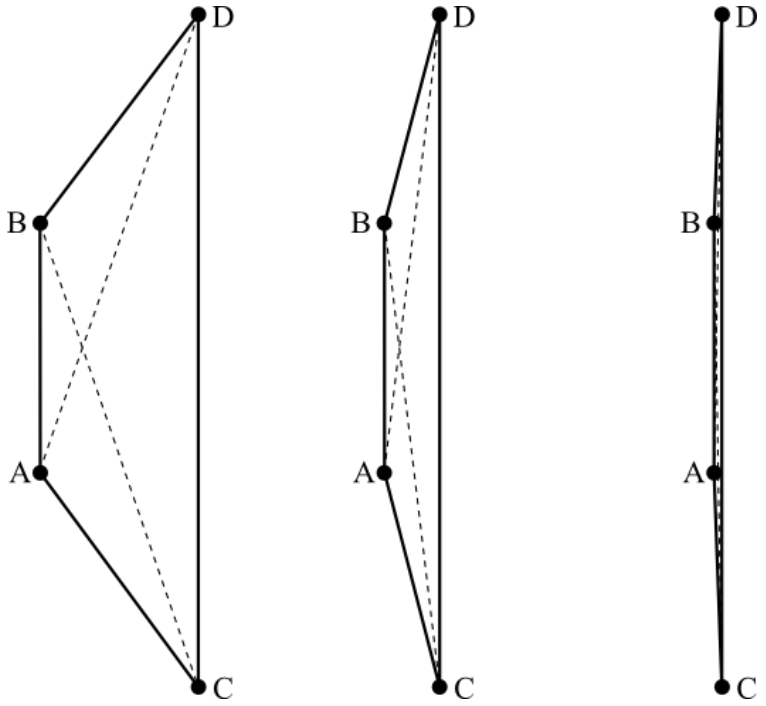
Ker v fazi združevanja algoritem že vsebuje optimizirano metodo za ponovno analizo štirikotnikov, nam faze združevanja ni potrebno spreminjati.

5. Težave in napake

Večino težav mi je ob pisanju algoritma povzročala uporaba številskega tipa Double. Ta je sicer nenatančen, je pa računanje z njim veliko hitreje od alternativ. Večino težav in pristopov k rešitvi sem že opisal v poglavju o implementaciji, ob daljšem testiranju algoritma pa se je pojavil problem, ki ga je težko preprečiti brez temeljitih sprememb samega algoritma.

Ob zagonu algoritma na seriji vhodnih množic se je pokazalo, da se število vseh povezav v izračunani triangulaciji pogosto ne ujema s številom, ki bi jih ta glede na število obodnih povezav (glej formulo v poglavju 3.1) morala imeti. Poleg tega je po izboljšavi take triangulacije na določenem mestu ostal navpičen šiv, sestavljen iz zelo dolgih povezav, ki v triangulaciji z najmanjšo dolžino povezav nimajo kaj početi. Izkaže se, da je problem povezan s kolinearnostjo.

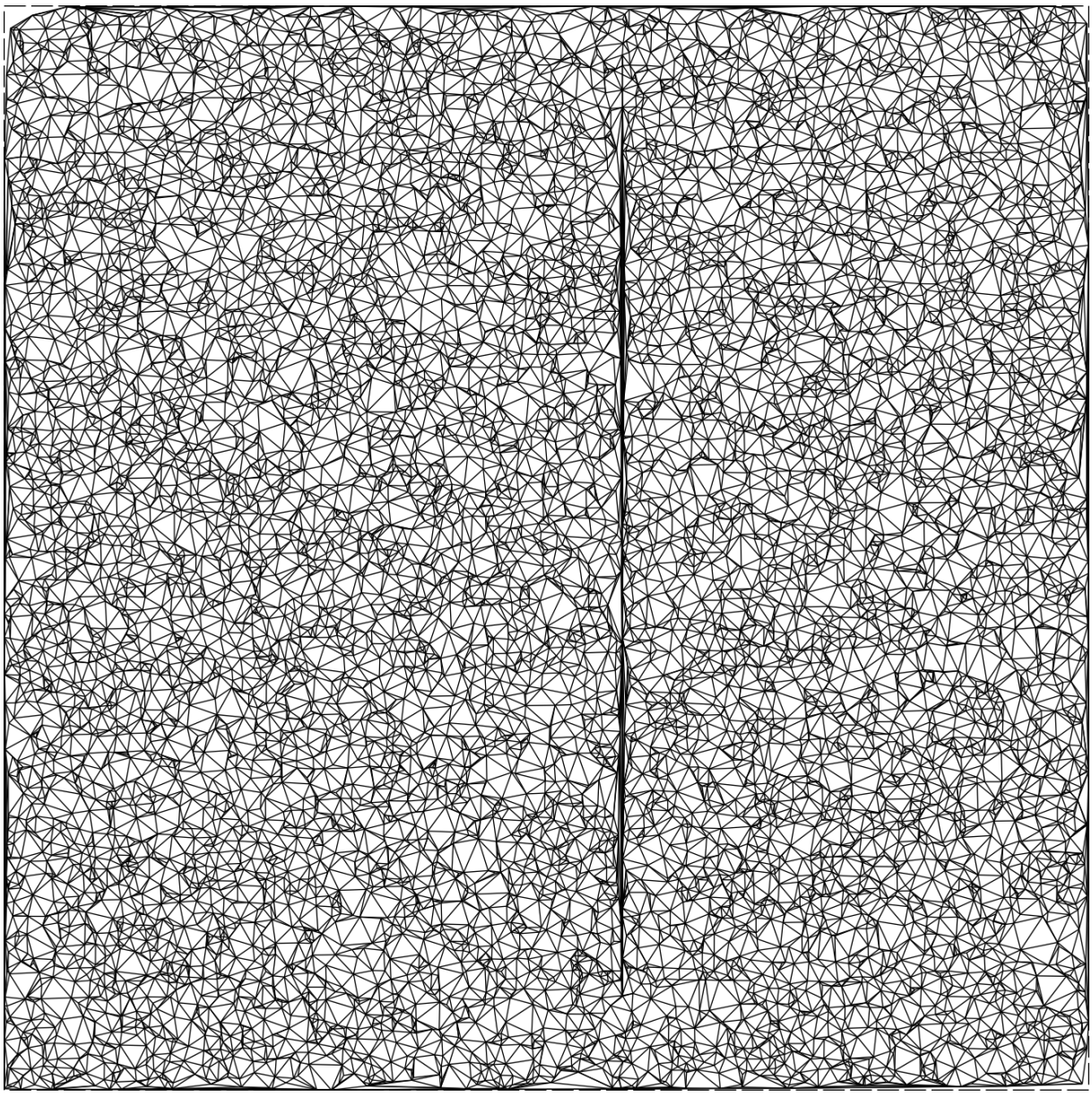
Način, na katerega obravnavamo kolinearnost točk pomeni, da lahko algoritem točko X najde za kolinearno z daljico AB, tudi če se ta nahaja zelo blizu na levi ali desni strani daljice. Za primer vzemimo sliko 40.



Slika 40: Štirikotnik ABCD z obema možnima diagonalama. Ko štirikotnik stiskamo po Y osi, kota med daljicama BA in BD, ter AB in AC postajata čedalje bližja 180° kotu.

Če štirikotnik iz slike stisnemo do te meje, da z uporabo formule za izračun kolinearnosti točki A in B za las še ne ležita na daljici CD, se lahko zgodi, da sodeč po isti formuli točka B leži na diagonali AD in točka A na diagonali BC. Ker algoritem dovoli vse 4 obodne

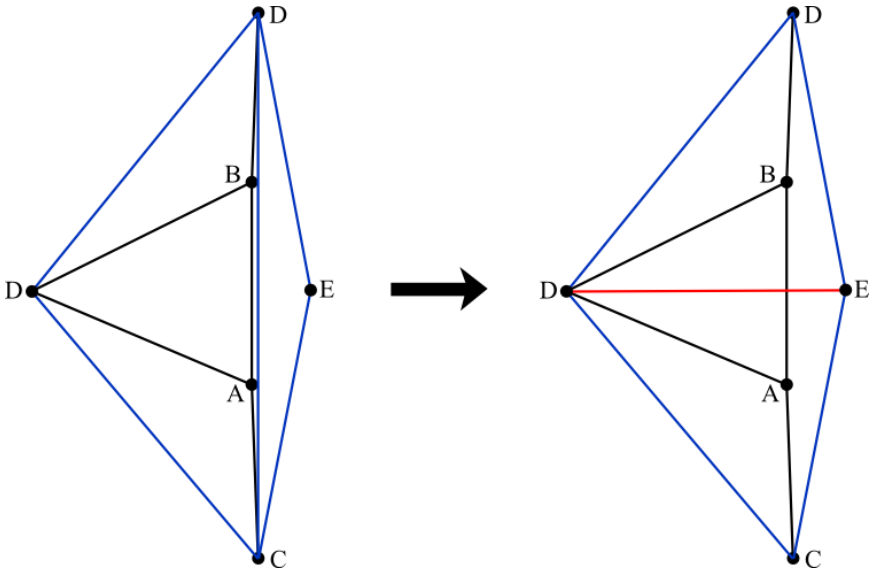
povezave, območje štirikotnika zapre, manjkajoča diagonala pa se odraža v manjkajoči povezavi triangulacije.



Slika 41: Optimizirana triangulacije množice 10000 točk. Napaka v triangulaciji se odraža v obliki navpičnega šiva.

Ker takemu štirikotniku manjka diagonala, ga algoritem ne more izboljšati. Prav tako tudi ne obstaja štirikotnik z eno od obodnih povezav problematičnega štirikotnika za diagonalo, saj je eno izmed lic, ki ga obada taka povezava, štirikotnik. Problem je še hujši, saj algoritem za iskanje štirikotnikov z izmenljivo diagonalo sklepa da, lahko vsako povezavo uporabi kot diagonalo štirikotnika, če sta točki, ki jo omejujeta, povezani z vsaj dvema skupnima točkama na nasprotnih straneh povezave. V primeru, da imata točki daljše daljice problematičnega

štirikotnika na vsaki strani eno točko, s katero sta povezani obe, lahko algoritem zato ustvari povezavo, ki seka poljubni del triangulacije. Pojav prikazuje slika 42.



Slika 42: Triangulacija 6 točk s problematičnim štirikotnikom ABCD. Ker povezava CD na levi in desni strani omejuje trikotnik, lahko algoritem za izboljševanje triangulacije ustvari povezavo DE, ki seka obstoječo povezavo AB.

Za rešitev problema uporabimo naslednji pristop. Ker poznamo formulo za izračun števila povezav triangulacije v odvisnosti od števila vseh točk in števila obodnih točk, lahko algoritem v vsakem koraku združevanja delnih triangulacij ugotovi, ali imamo manjkajočo povezavo in je torej verjetno prišlo do pravkar opisanega problema. Besedo verjetno uporabimo zato, ker formula za izračun števila točk triangulacije predpostavi, da so točke v konveksni poziciji in se zato število vseh povezav ne bo ujemalo s izračunanim tudi, če so vse točke triangulacije kolinearne. V primeru, da se pojavi verjetnost napake, se algoritem sprehodi po seznamu sosednosti in poišče naslednje povezave:

1. obodne povezave, ki na notranjo stran omejuje lice z več kot tremi stranicami,
2. notranje povezave, ki na vsaj eno stran omejuje lice z več kot tremi stranicami.

Te povezave predpostavi za obodne povezave problematičnega štirikotnika. Poišče jih tako, da poišče vse povezave, ki jim ustreza eden od sledečih pogojev:

- točki, ki omejujeta povezavo, nista povezani z nobeno skupno točko, ali
- točki, ki omejujeta notranjo povezavo, sta povezani z le eno skupno točko, ali
- vse točke, s katerimi sta povezani točki, ki omejujeta notranjo povezavo triangulacije, se nahajajo na isti strani notranje povezave, ali
- eno izmed dobljenih lic, ki naj bi jih omejevala povezava, ni dejansko lice triangulacije.

Za preizkus prvih treh uporabimo principe za iskanje štirikotnikov z izmenljivo daljico. Za preizkus zadnjega pogoja uporabimo metodo za izračun konveksne ovojnice, in sicer na sledeč način: najprej vzamemo vse točke X_i, ki se nahajajo na istem intervalu, kot točke trikotnega lica, ki ga omejuje povezava. Algoritem za konveksno ovojnico poženemo na točkah trikotnika in točki X_i. Če se točka X_i ne nahaja na konveksni ovojnici, se nahaja v notranjosti trikotnika, iz česar sledi, da trikotnik ni lice triangulacije.

Ko imamo vse povezave, ki ustrezajo omenjenim pogojem, sestavimo podgraf triangulacije, ki sestoji iz omenjenih povezav in točk, ki jih omejujejo. Po grafu se sprehodimo s DFS (Depth first search) algoritmom in poiščemo vse cikle z dolžino večjo od 3. V take cikle vstavljamo povezave med ne-sosednimi pari točk, s čimer praviloma odpravimo napako. Pri vstavljanju povezav še vedno upoštevamo sekanje, ne upoštevamo pa kolinearnosti, zaradi katere je do napake sploh prišlo.

6. Rezultati

Predstavljene meritve predstavljajo zmogljivosti algoritma tako v odvisnosti od velikosti problema kot tudi od načina, na katerega algoritem opravlja deljenje na podprobleme. Ob merjenju se osredotočimo na naslednje faktorje, po katerih ocenjujemo zmogljivost algoritma:

- skupna dolžina povezav osnovne triangulacije,
- skupna dolžina povezav izboljšane triangulacije z različnimi načini izboljšave (izboljšava rezultata, sprotna izboljšava),
- čas, ki ga algoritem porabi za izračun osnovne triangulacije in
- skupni čas, ki ga porabi algoritem za izračun izboljšane triangulacije (osnovna triangulacija + izboljšava) z različnimi načini izboljšave (izboljšava rezultata, sprotna izboljšava).

Testiranje je bilo izvajano na procesorju Core 2 Duo E8500, 3.14 GHz. Test za vsako velikost problema je bil izveden velikokrat, vsakič na drugi naključno generirani množici točk. Iz dobljenih rezultatov so izvlečena povprečja za vsako skupino, ki so predstavljena v podpoglavjih.

6.1 Osnovna triangulacija

Kot smo že povedali, algoritem pozna štiri načine, na katere množico točk razdeli na lažje obvladljive podmnožice. Povedali smo tudi, da se rezultat triangulacije z uporabo različnih načinov deljenja razlikuje. Pogledali si bomo za koliko in kakšen je vpliv na dolžino povezav.

Sledijo rezultati testiranja. Rezultati za različne načine deljenja so predstavljeni v lastnih tabelah, povzeti pa so na grafu 1.

Velikost problema	10	100	1000	10000
Skupna dolžina povezav	8.69204004132	81.61124924	775.2127722	7715.576588

Tabela 9: Dolžina povezav v odvisnosti od velikosti problema pri uporabi enakomernega deljenja na dva dela.

Velikost problema	10	100	1000	10000
Skupna dolžina povezav	8.71522630000	83.59958	802.278535	7954.54

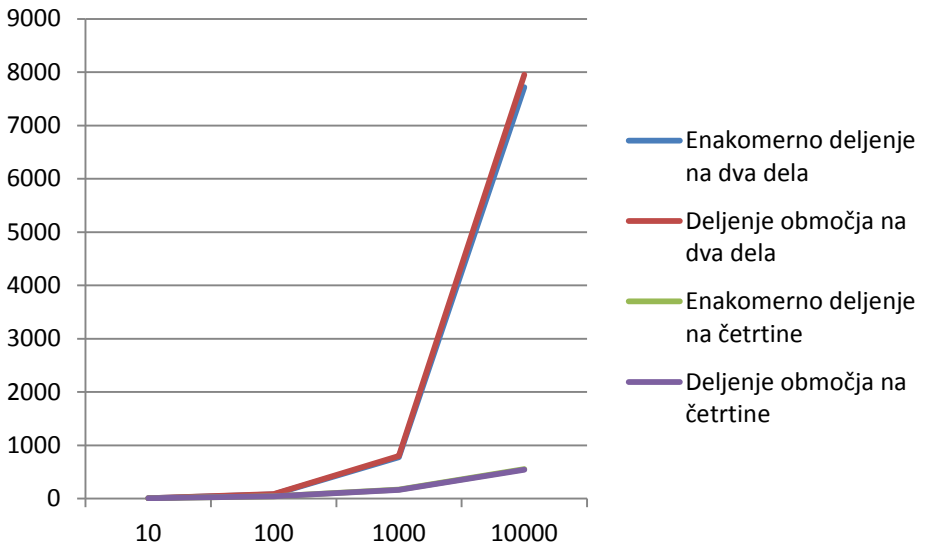
Tabela 10: Dolžina povezav pri uporabi deljenja območja na dva dela.

Velikost problema	10	100	1000	10000
Skupna dolžina povezav	8.48552101940	46.3977	168.7	556.248

Tabela 11: Dolžina povezav pri uporabi enakomernega deljenja na četrtine.

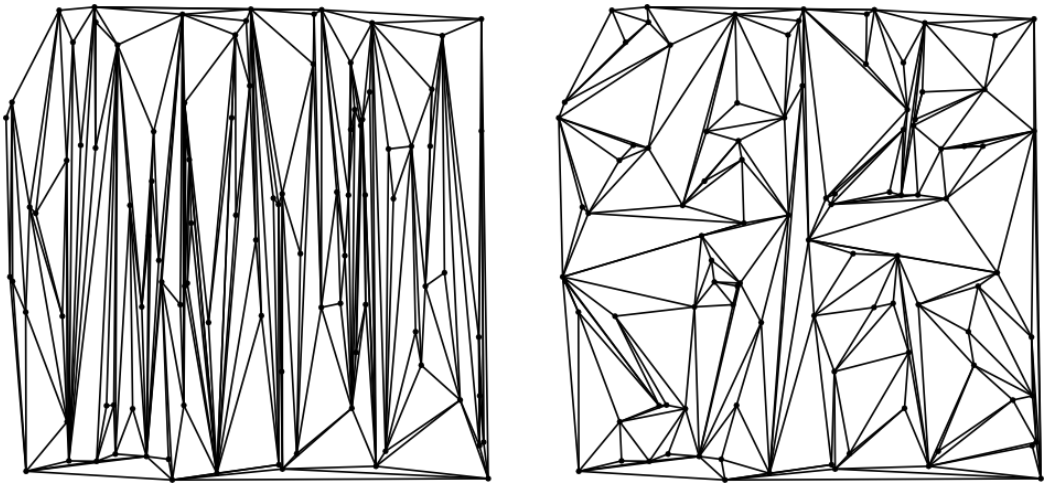
Velikost problema	10	100	1000	10000
Skupna dolžina povezav	8.30000000000	44.89	163.4	539.118

Tabela 12: Dolžina povezav pri uporabi deljenja območja na četrtine.



Graf 1: Odvisnost dolžine povezav od števila točk z različnimi načini deljenja problema.

Kot je razvidno iz rezultatov, je končna triangulacija pri obeh načinih deljenja na četrtine veliko bližje želenemu rezultatu, kot pri deljenju na polovice. Opazimo tudi, da dolžina povezav triangulacij, ki uporabljajo deljenje na polovice, hitro narašča z velikostjo problema. Razlog tiči v tem, da pri deljenju na četrtine v vsakem koraku deljenja omejimo največjo povezavo v delni triangulaciji na četrtino trenutnega območja. Pri deljenju na dva dela to ne velja in povezave so pogosto zelo dolge, kot prikazuje slika 43.



Slika 43: Triangulacije iste množice stotih točk pri deljenju na polovice po X osi(levo) in deljenju na četrtine(desno). Kot je opaziti is slike, se pri deljenju na polovice po X osi pojavijo dolge navpične povezave, ki močno vplivajo na skupno dolžino povezav.

Tako kot dolžina povezav osnovne triangulacije je tudi čas izvajanja algoritma za izračun le-te odvisen od izbranega načina deljenja problema. Kot smo že omenili, mora algoritem za deljenje na četrtine točke ponovno urejati ob vsakem deljenju. Prav tako smo omenili, da moramo ob deljenju območja najprej poiskati zadnjo točko, ki še spada v prvo polovico območja. Preden povzamemo, kakšen vpliv imajo ti faktorji na čas izvajanja algoritma za osnovno triangulacijo, si poglejmo rezultate za posamezen način.

Velikost problema	**10**	**100**	**1000**	**10000**
Čas izvajanja (s)	0.000142965	0.002445616	0.0342243	0.412427356

Tabela 13: Čas izvajanja osnovne triangulacije v odvisnosti od velikosti problema pri uporabi enakomernega deljenja na dva dela.

Čase izvajanja iz tabele 13 uporabimo kot osnovo, na podlagi katere ocenjujemo vse ostale. Ker pri enakomernemu deljenju na polovice ne izvajamo nobene dodatne operacije razen začetne ureditve vhodnih točk, bi moral biti algoritem v tem primeru najhitrejši.

Velikost problema	**10**	**100**	**1000**	**10000**
Čas izvajanja (s)	0.000166078	0.00271539	0.036397984	0.4379

Tabela 14: Čas izvajanja pri uporabi deljenja območja na dva dela.

Kot je razvidno iz tabele 14, je čas izvajanja algoritma daljši kot pri enakomernemu deljenju. To je bilo tudi pričakovano, saj pri enakomernem deljenju algoritem preprosto vzame polovico elementov tabele za vsak podproblem, pri deljenju območja pa mora mesto delitve

vsakič znova poiskati. Poleg tega bo algoritem pri enakomernem deljenju vedno porabil najmanjše možno število delitev, pri deljenju na območja pa ne, saj se lahko v katerikoli izmed polovic intervala, ki ga delimo, ne nahaja nobena točka.

Velikost problema	10	100	1000	10000
Čas izvajanja (s)	0.000171354	0.002934176	0.0378562	0.48006

Tabela 15: Čas izvajanja pri uporabi enakomernega deljenja na četrtine.

Tabela 15 prikazuje čas izvajanja pri enakomernem deljenju na četrtine. Izvajanje je še počasnejše kot pri deljenju območja na polovice, saj je tam edina dodatna operacija iskanje sredinskega elementa – ki z uporabo dvojiškega iskanje vzame O(log n) časa, pri deljenju na četrtine pa mora algoritem pri vsakem deljenju pognati algoritem quicksort, ki vzame O(n log n) časa.

Velikost problema	10	100	1000	10000
Čas izvajanja (s)	0.0001875	0.0030426	0.04157	0.50633716

Tabela 16: Čas izvajanja pri uporabi deljenja območja na četrtine.

Deljenje območja na četrtine predstavlja kombinacijo prejšnjih dveh pristopov in zato podeduje vse hibe prejšnjih pristopov. Kot tak je tudi opazno najpočasnejši, kot pa smo že opazili, je končni rezultat najboljši prav pri uporabi tega pristopa.

6.1 Izboljšava triangulacije

Algoritem omogoča dva načina izboljšave triangulacije – izboljšava rezultata, ki vzame izhod prvega dela algoritma in ga približa želenemu rezultatu, in sprotna izboljšava, ki izboljša vsak delni rezultat. Rezultati testiranja zmogljivosti so predstavljeni v tabelarični obliki, pri čemer za vsak način deljenja uporabimo novo tabelo.

Najprej si poglejmo rezultate za skupno dolžino povezav izboljšane triangulacije. Kot smo omenili v prejšnjem poglavju, so triangulacije ustvarjene z deljenjem na četrtine veliko bližje želenemu rezultatu kot tiste, ki so ustvarjene z deljenjem na polovice. Poglejmo si vpliv te lastnosti na končni rezultat.

Velikost problema	10	100	1000	10000
Dolžina povezav (izboljšava rezultata)	7.49	35.33553744	114.153	349.5
Dolžina povezav (sprotna izboljšava)	7.48	35.275	114.16	349.489

Tabela 17: Skupna dolžina povezav optimizirane triangulacije z uporabo enakomernega deljenja na dva dela.

Kot je razvidno iz tabele 17, način izboljšave nima velikega vpliva na končni rezultat. Odstopanja so dovolj majhna, da jih lahko pripišemo variaciji med testnimi primeri.

Velikost problema	10	100	1000	10000
Dolžina povezav (izboljšava rezultata)	7.47468	35.34101247	114.2306401	349.542
Dolžina povezav (sprotna izboljšava)	7.483652	35.34467454	114.2465737	349.476433

Tabela 18: Skupna dolžina povezav optimizirane triangulacije z uporabo deljenja območja na dva dela.

Rezultati pri deljenju območja so podobni kot pri enakomernemu deljenju, kar je bilo pričakovati. Pri večjih testnih primerih opazimo, da je skupna dolžina rahlo daljša kot v prejšnjem primeru, vendar za dovolj majhno količino, da je rezultat še vedno dober.

Velikost problema	10	100	1000	10000
Dolžina povezav (izboljšava rezultata)	7.486	35.372	114.265	349.879
Dolžina povezav (sprotna izboljšava)	7.477	35.3735536	114.3026	349.8521

Tabela 19: Skupna dolžina povezav optimizirane triangulacije z uporabo enakomernega deljenja na četrtine.

Rezultat iz tabele 19 predstavlja nenavaden pojav – kljub temu, da je bil rezultat osnovne triangulacije s pristopom enakomernega deljenja na četrtine opazno boljši kot pri pristopih deljenja na dva dela, je končni rezultat po izboljšavi take triangulacije v povprečju slabši. Med vrstama izboljšave iz vidika dolžine povezav spet ni velikih razlik.

Velikost problema	10	100	1000	10000
Dolžina povezav (optimizacija rezultata)	7.471638	35.35228	114.21062	349.7321
Dolžina povezav (sprotna optimizacija)	7.475347	35.345712	114.24508	349.671983

Tabela 20: Skupna dolžina povezav optimizirane triangulacije z uporabo deljenja območja na četrtine.

Kot je razvidno iz tabele 20, je končni rezultat sicer rahlo boljši kot pri pristopu enakomernega deljenja na četrtine, vendar še vedno slabši kot pri uporabi pristopov deljenja na dva dela.

Končni rezultat testiranja kaže na to, da sicer boljša vhodna triangulacija, ki jo pridobimo z deljenjem na četrtine, po izboljšavi v povprečju postane do 0.1% slabša od triangulacije, ki jo pridobimo z deljenjem na dva dela. Izbran način izboljšave pa v povprečju ni imel nikakršnega vpliva na dolžino povezav triangulacije. Poglejmo si, ali ima vpliv na čas izvajanja.

Velikost problema	10	100	1000	10000
Čas izvajanja pri izboljšavi rezultata (s)	0.0002475	0.006365099	0.115658603	2.09
Čas izvajanja pri sprotni izboljšavi (s)	0.00027868	0.0064924	0.10838	1.51768

Tabela 21: Skupni čas izvajanja osnovne triangulacije in izboljšav pri uporabi enakomernega deljenja na dva dela.

Kot je razvidno iz tabele 21, se algoritem ob uporabi sprotne izboljšave izvaja dlje kot z uporabo izboljšave rezultata, ko je vhod majhen, vendar pa se izvaja precej hitreje (do 37% na 10000 točkah) na velikih vhodih. Za ta pojav obstajata dva verjetna razloga. Prvi je ta, da algoritem ob sprotni izboljšavi večino časa upravlja z majhnimi seznami sosednosti, kar mu pospeši iskanje sosednih točk in posredno delovanje celotnega algoritma.

Ker algoritem na seznam štirikotnikov z izmenljivo diagonalo ne sme vnesti štirikotnika, ki ima za obodno stranico diagonalo obstoječega štirikotnika v seznamu, mora v vsakem koraku preveriti cel seznam štirikotnikov. Ta je v splošnem opazno manjši, ko uporabljamo sprotno izboljšavo, kar je tudi drugi razlog, da je algoritem ob sprotnem izboljševanju hitrejši.

Velikost problema	10	100	1000	10000
Čas izvajanja pri izboljšavi rezultata (s)	0.000259687	0.006573852	0.118472496	2.11724
Čas izvajanja pri sprotni izboljšavi (s)	0.000290472	0.006823562	0.111077395	1.585319554

Tabela 22: Skupni čas izvajanja osnovne triangulacije in izboljšave pri uporabi deljenja območja na dva dela.

Čas izvajanja v primeru iz tabele 22 ni bistveno daljši kot pri enakomernem deljenju na dva dela. Velik del razlike nastane že v prvi fazi, torej pri samem izboljševanju ni bistvenih razlik.

Velikost problema	10	100	1000	10000
Čas izvajanja pri izboljšavi rezultata (s)	0.000259872	0.004996537	0.069021597	0.948424128
Čas izvajanja pri sprotni izboljšavi (s)	0.000286722	0.005496508	0.07125	0.86004761

Tabela 23: Skupni čas izvajanja osnovne triangulacije in izboljšave pri uporabi enakomernega deljenja na četrtine.

Večje razlike opazimo pri izboljšavi triangulacije, ki smo jo pridobili z uporabo enakomernega deljenja na četrtine (tabela 23). Kljub temu, da je bil osnovni algoritem pri deljenju na četrtine počasnejši kot pri deljenju na dva dela, je skupni čas izvajanja pri uporabi deljenja na četrtine dvakrat manjši kot pri uporabi deljenja na dva dela. Razlog je v tem, da je osnovna triangulacija, ki jo pridobimo z deljenjem na četrtine, že razmeroma blizu želenemu rezultatu. Algoritem mora zato v fazi izboljšave izmenjati veliko manj povezav kot pri deljenju na dva dela. Podoben rezultat opazimo tudi pri izboljšavi triangulacij, pridobljenih z deljenjem območja na četrtine, kot prikazuje tabela 24.

Velikost problema	10	100	1000	10000
Čas izvajanja pri izboljšavi rezultata (s)	0.000276041	0.005149772	0.0710337	0.949694528
Čas izvajanja pri sprotni izboljšavi (s)	0.00030952	0.005625098	0.074882605	0.87080322

Tabela 24: Skupni čas izvajanja osnovne triangulacije in izboljšave pri uporabi deljenja območja na četrtine.

7. Zaključek

Cilj diplomskega dela je bil implementirati algoritem, ki bo za vhodno množico točk izračunal čim boljši približek triangulaciji z najmanjšo dolžino povezav. Algoritem naj bi imel časovno zahtevnost manjšo od $O(n^2)$ in teoretično zmogel izračunati triangulacijo za poljubno končno število točk.

Rezultati kažejo, da je algoritem daleč najhitrejši, ko problem deli na četrtine, in izračuna nekoliko boljšo rešitev, ko problem deli na polovici. Osebno menim, da je pohitritev z uporabo v prvem primeru dovolj velika, da poraba dodatnega časa za izračun 0.1% boljše rešitve ni smiselna. Izmed načinov za izboljšavo rešitve, ki jih algoritem ponuja, se je najbolje izkazalo sprotno izboljševanje, ki v povprečju izračuna enako rešitev kot alternativa, je pa opazno hitrejše.

Med implementacijo algoritma sem se prvič seznanil s problemom natančnosti števil s plavajočo vejico. Problem se je v različnih oblikah pojavil v skoraj vseh delih algoritma, in čeprav upam, da sem jih na tak ali drugačen način odpravil ali se jim izognil, v končni fazi preprosto ni zagotovila, da se ne bo našel vhod, ki ga program ne bo zmogel pravilno triangulirati. Menim, da je v tem trenutku najbolj vprašljiv del algoritma, ki išče obod triangulacije z algoritmom za izračun konveksne ovojnice točk. Čeprav se v praksi napaka ni pojavila, bi, v kolikor bi imel več časa, vsekakor poskušal implementirati algoritem, ki obod išče na podlagi povezav in ne točk, in ki bi pri izračunu oboda združene triangulacije upošteval obode delnih triangulacij.

Poleg izboljšav na področju zanesljivosti bi se algoritem dalo dodatno optimizirati z vpeljavo niti. Ker se po razdelitvi problema na podprobleme vsak izmed teh triangulira neodvisno od drugega, je algoritem že skoraj pripravljen na izvajanje na več nitih. To je še dodaten razlog, da je sprotno izboljševanje triangulacije najboljša rešitev, saj je v tem primeru za poganjanje programa na več nitih dovolj, da znamo podprobleme pravilno razdeliti med niti.

Kljub vsem problemom, s katerimi sem se soočil ob implementaciji algoritma, pa mislim, da sem dosegel cilj diplomskega dela. Algoritem, ki sem ga implementiral, ne glede na način uporabe izkazuje časovno zahtevnost $O(n)$, v praksi pa se izkazal za dovolj zanesljivega, da tudi po več urah zaporednega izvajanja ni spodletel.

Slike

Literatura

[1] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co, New York, NY, USA, 1979.

[2] T. Lambert, The Delaunay triangulation maximizes the mean inradius, Proceedings of the 6th Canadian Conference on Computational Geometry, Saskatoon, SK, Kanada, pp. 201-206, 1994.

[3] W. Mulzer, G. Rote, Minimum-weight triangulation is NP-hard, Journal of the ACM (JACM), Vol. 55, No. 11, pp. 1-29, 2008.

[4] (2012) Delaunay triangulation. Dostopno na: http://en.wikipedia.org/wiki/Delaunay_triangulation.

[5] (2012) Convex Hull. Dostopno na: http://www.cs.kzoo.edu/cs215/lectures/m3-convex-hull.pdf

[6] (2012) Monotone Chain Convex Hull. Dostopno na: http://www.algorithmist.com/index.php/Monotone_Chain_Convex_Hull