Strukture podataka za efikasnu manipulaciju geometrijskim podacima

Kosir, Ines

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Ines Kosir

STRUKTURE PODATAKA ZA EFIKASNU MANIPULACIJU GEOMETRIJSKIM PODACIMA

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Zvonimir Bujanović

Zagreb, srpanj, 2022.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Zahvaljujem se mentoru na strpljenju i svim korisnim savjetima, obitelji na podršci tijekom cijelog studiranja, a posebno sestri koja je spavala na stolcu pokraj mene tijekom napornih noći punih učenja. Hvala Jurici koji je svaki projekt učinio zabavnijim te usput razmijenjivao znanje sa mnom.

Sadržaj

Uvod

Geometrijski algoritmi počeli su se značajnije proučavati u kasnim 70-im godinama prošlog stoljeća. Brz i uspješan napredak u njihovom stvaranju možemo pripisati velikoj količini raznih geometrijskih problema. S druge strane, dobiveni algoritmi koriste se u informatičkim sustavima - računalna grafika, geografski informacijski sustav (GIS), robotika i slično - gdje igraju važnu ulogu u funkcioniranju sustava u cjelosti.

U prošlosti, mnogi geometrijski problemi bili su riješeni neefikasnim i matematički kompliciranim algoritmima koje je bilo teško implementirati. Danas postoje različiti pristupi rješavanju tih problema te su razvijene različite tehnike koje su pojednostavile i poboljšale prijašnje pokušaje rješavanja problema. Mi ćemo se u ovom radu baviti s nekoliko modernih algoritama koji rješavaju određene geometrijske probleme.

Za dobro razumijevanje rada, potrebno je poznavanje osnovnih pojmova iz geometrije. Također, pretpostavljamo da je čitatelj upoznat s oblikovanjem i analizom algoritama, upotrebom veliko *O* notacije (eng. *big-O notation*), različitim algoritamskim tehnikama poput sortiranja, binarnog pretraživanja te strukturama podataka poput vektora, lista, binarnih stabala i slično.

Prvo poglavlje pruža nam uvod u razumijevanje načina pohranjivanja točaka, pretraživanja raznih raspona te također uvodimo dvije nove strukture: kd-stablo i quadtree. Nove strukture ćemo implementirati te primjerom pokazati njihovo korištenje, a služe nam za pohranjivanje skupa dvodimenzionalnih točaka.

Drugo poglavlje nastalo je u svrhu pokazivanja primjene quadtree-a kao strukture podataka koja nam može uvelike olakšati rješenje konkretnog problema. Mi se ovdje specificiramo na jednu upotrebu quadtree-a, a to je kreiranje triangulacija. Pokazujemo kako prilagođavamo izgradnju quadtree-a za potrebe našeg problema te implementiramo cijeli algoritam triangulacije.

Poglavlje 1

Algoritmi za upite pravokutnih raspona

U današnjem dobu gdje su nam informacije nadohvat ruke, podaci su srž rada svih aplikacija i portala s raznim informacijama. Kako su informacije sve dostupnije, podaci se moraju na neki način pohraniti. Baze ovdje rješavaju problem, no one pohranjuju sve podatke. Da bismo dobili samo dio koji nas zanima, podatke moramo nekako filtrirati, odnosno postaviti upite (eng. *queries*) na bazu. Upitom dobivamo određeni skup podatka koji zadovoljava restrikcije samog upita. Taj skup često možemo reprezentirati geometrijski. Isprva se može činiti da baze nemaju veze sa geometrijom, no jedan redak u bazi (ili jedan dobiveni rezultat upita) može se interpretirati kao točka u višedimenzionalnom prostoru. Isto tako, više redaka interpretiramo kao skup točaka u prostoru.

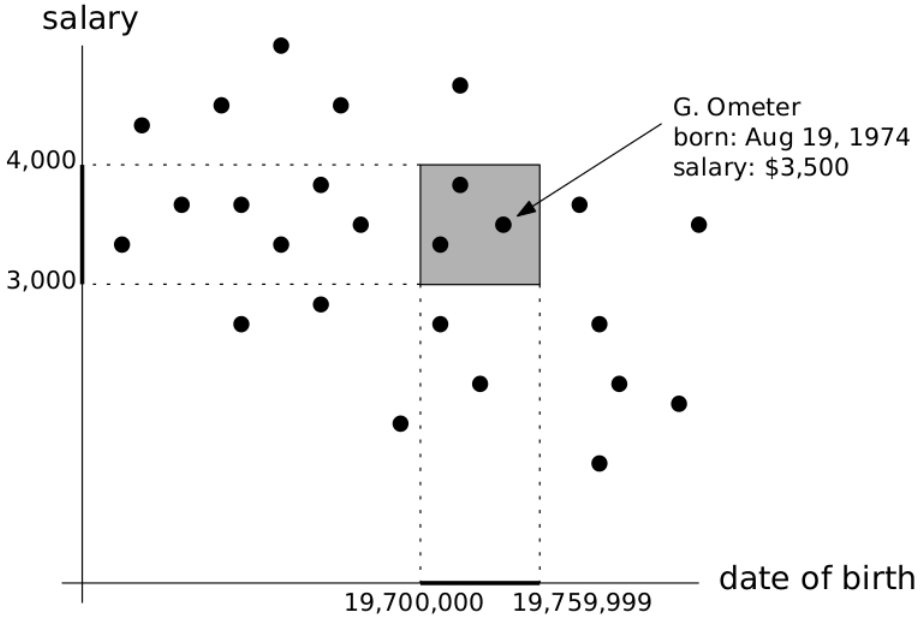
Za primjer navedenog razmišljanja, promotrimo bazu podataka za osobnu administraciju, odnosno bazu koja sadrži informacije o zaposlenicima proizvoljne imaginarne tvrtke. Potrebne informacije su podaci o imenu, prezimenu, datumu rođenja, plaći, broju djece i slično. Tipičan upit može zahtijevati ispis svih zaposlenika rođenih između 1970. i 1975. godine, čiji je raspon mjesečne plaće između $3 000 i $4 000. Kako bismo ovo reprezentirali kao geometrijski problem, svakog zaposlenika predstavljamo pomoću jedne točke. Prva koordinata točke je cijeli broj dobiven formulom

$$10\,000 \times godina + 100 \times mjesec + dan$$

koja predstavlja datum rođenja. Druga koordinata je cijeli broj koji predstavlja mjesečnu plaću zaposlenika. Uz ove koordinate, u pripadnom objektu tipa *točka* koji opisuje pojedinog zaposlenika pohranjujemo i ostale podatke poput imena, adrese i slično, no te informacije ne gledamo na koordinatnim osima jer nam nisu relevantne za dani primjer upita. Upit na bazu koji traži sve zaposlenike rođene između 1970. i 1975. godine s mjesečnom plaćom u rasponu između $3 000 i $4 000 možemo prikazati kao sljedeći geometrijski upit: Tražimo sve točke u dvodimenzionalnom prostoru čija se prva koordinata nalazi u rasponu između 19 700 000 i 19 759 999, a druga koordinata između 3 000 i 4 000. Drugim riječima, želimo pronaći sve točke u pravokutniku zadanih dimenzija - vidi sliku 1.1.

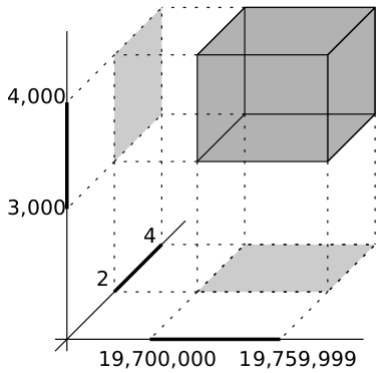
Što ako želimo napraviti upit koji sadrži i informaciju o broju djece zaposlenika? U tom slučaju, svakog zaposlenika reprezentiramo pomoću točke u trodimenzionalnom prostoru. Prve dvije koordinate točke su iste kao i u prethodnom slučaju (prva označava datum rođenja, a druga plaću). Treća koordinata je broj koji predstavlja broj djece zaposlenika. Zadani upit glasi: tražimo sve zaposlenike rođene između 1970. i 1975. godine, čiji je raspon mjesečne plaće između \$3 000 i \$4 000 te koji imaju barem dvoje, a najviše četvero djece. Odgovor na dani upit je skup svih točaka koje se nalaze u kvadru dimenzija $[19\,700\,000, 19\,759\,999] \times [3\,000, 4\,000] \times [2, 4]$. Geometrijski prikaz možemo vidjeti na slici 1.2.

Proširivanjem navedenog razmišljanja, dobivamo naputak za kreiranje upita s više potrebnih informacija. Ukoliko želimo odgovor na upit koji sadrži n validnih informacija, tada transformiramo retke pohranjene u bazi u točke n-dimenzionalnog prostora. Ukoliko želimo dobiti odgovor na upit koji ima zadani raspon za određenu informaciju, tada transformiramo retke u skup točaka koje leže u n-dimenzionalnom tijelu definiranom određenim dimenzijama. Takav upit nazivamo **upit pravokutnog raspona** (eng. *rectangular range query*).



Izvor: Preuzeto iz cjeline 5.1 iz [2]

Slika 1.1: Geometrijski prikaz jednostavnog upita na bazu



Izvor: Preuzeto iz cjeline 5.1 iz [2]

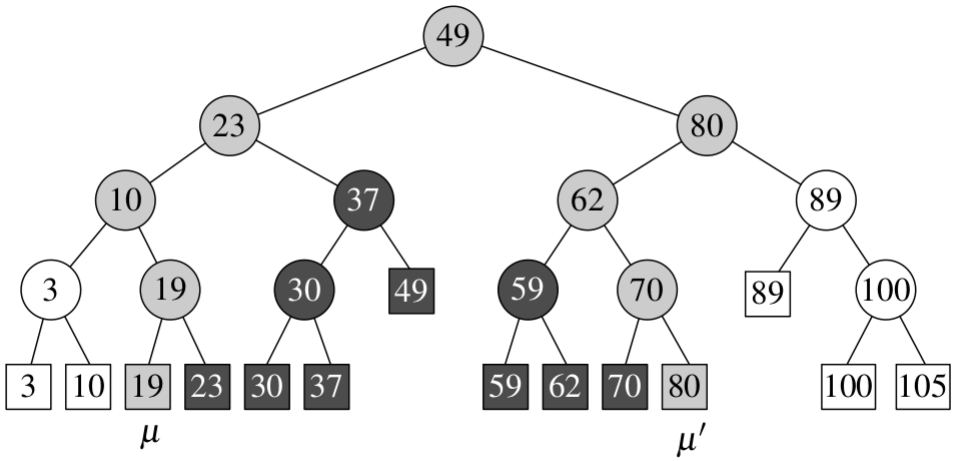
Slika 1.2: Geometrijski prikaz upita s više informacija

1.1 Pretraživanje jednodimenzionalnog raspona

Pretraživanje jednodimenzionalnog raspona uključuje pretraživanje skupa točaka u jednodimenzionalnom prostoru, tj. skupa realnih brojeva. Upitom jednodimenzionalnog pravokutnog raspona zapravo tražimo točke unutar intervala $[x, x']$.

Neka je $P := \{p_1, p_2, ..., p_n\}$ skup točaka s realnom koordinatom te $[x, x']$ zadani interval. Želimo pronaći sve točke skupa P koje pripadaju spomenutom intervalu. Naivnim algoritmom prolazimo kroz listu (ili niz) u kojoj su pohranjene sve točke te provjeravamo pripada li točka na određenoj poziciji liste našem intervalu. Kako prolazimo kroz cijelu listu, složenost naivnog algoritma je upravo $O(n)$ gdje je n kardinalitet skupa P. Za pretraživanje višedimenzionalnog raspona postoji više algoritama, no mi ćemo izdvojiti onaj koji upotrebljava balansirano binarno stablo. Vratimo se ponovo na pretraživanje jednodimenzionalnog raspona. Neka je $\mathcal{T}$ balansirano binarno stablo koje gradimo na rekurzivan način: sortiramo početni skup te mu odredimo medijan. Ukoliko skup sadrži paran broj točaka, za medijan uzimamo poziciju $\lceil n/2 \rceil$ gdje je n ukupan broj točaka skupa. Točku na poziciji medijana pospremimo u korijen stabla te skup podijelimo na dva dijela. Prvi dio skupa, nazovimo ga *firstSet*, sadrži točke od početka skupa do pozicije medijana (uključivo), dok drugi dio skupa (*secondSet*) uključuje točke od pozicije medijana (isključivo) do kraja već sortiranog skupa točaka. Kako gradimo balansirano binarno stablo, *firstSet* predstavlja točke koje će biti pohranjene u lijevom podstablu, a *secondSet* točke koje će se nalaziti u desnom podstablu. Na isti način, odredimo medijan skupa *firstSet* i točku na poziciji medijana pohranimo u korijen lijevog podstabla te podijelimo *firstSet* na dva dijela čije će točke graditi lijevo, odnosno desno podstablo korijena lijevog podstabla. Analogni postupak radimo i za skup *secondSet*. Postupak ponavljamo sve dok kardinalitet skupa

kojeg dijelimo nije jednak jedan, odnosno dok nije preostala samo jedna točka. U tom slučaju, preostalu točku napravimo listom stabla i završavamo izgradnju stabla. Konačni rezultat izgradnje stabla možemo vidjeti na slici 1.3 pri čemu zasad možemo ignorirati različito obojane čvorove stabla.



Izvor: Preuzeto iz cjeline 5.1 iz [2]

Slika 1.3: Pretraživanje 1-dimenzionalnog raspona pomoću binarnog stabla traženja

Vratimo se na pretraživanje jednodimenzionalnog raspona pomoću binarnog stabla $\mathcal{T}$. Neka je P i dalje početni skup točaka te $[x, x']$ zadani interval. Listovi stabla $\mathcal{T}$ su točke zadanog skupa P poredane na način da ih rekurzivni obilazak stabla redom *lijevo podstablo – korijen – desno podstablo* obilazi uzlazno sortiranim redoslijedom. Unutarnji čvorovi sadrže razdjelne vrijednosti koje pomažu u traženju rješenja (njihovu ulogu ćemo detaljnije objasniti u nastavku). Označimo sa v čvor u kojem je pohranjena vrijednost x_v koja dijeli pretragu na dva podstabla. Tada lijevo podstablo čvora v sadrži sve točke manje ili jednake vrijednosti x_v, a desno sve točke strogo veće od x_v. Cilj nam je pronaći točke skupa P koje se nalaze u intervalu $[x, x']$. Nalazimo ih na sljedeći način: Neka su μ i μ' listovi koji označavaju kraj pretraživanja, tj. listovi koji pripadaju redom najmanjem ("najlijevijem") i najvećem ("najdesnijem") elementu od P iz $[x, x']$. Tada su tražene točke u intervalu $[x, x']$ zapravo listovi između vrijednosti μ i μ' pri čemu su i same vrijednosti pohranjene u μ i μ' mogući kandidati pretrage. Dajemo primjer koji reprezentira navedeno razmišljanje.

Neka je $P := \{3, 10, 19, 23, 30, 37, 49, 59, 62, 70, 80, 89, 100, 105\}$, te zadani interval u kojem želimo pronaći točke $[18, 77]$. Proučavajući sliku 1.3, trebamo pronaći sve točke koje su pohranjene u tamnosivo obojanim listovima te točku koja je pohranjena u listu μ svijetlosive boje. Kako slika 1.3 nagoviješta, tražene točke su listovi određenih podstabla

na putevima traženja. Ta podstabla su označena tamno sivom bojom, dok su putevi traženja obojani svijetlo sivom bojom. Preciznije, korijeni podstabla koje odabiremo su čvorovi koji se nalaze između dva puta traženja te čiji se roditelji nalaze na istom putu traženja koji obuhvaća razdjelni čvor v_r. Kako bismo pronašli spomenute čvorove na putu traženja, prvo trebamo odrediti razdjelni čvor v_r nakon čega se pretraga dijeli na dva puta traženja. Traženi algoritam implementiran je metodom *OdrediRazdjelniČvor* te ju navodimo u nastavku:

Algorithm 1: OdrediRazdjelniČvor($\mathcal{T}, x, x'$)

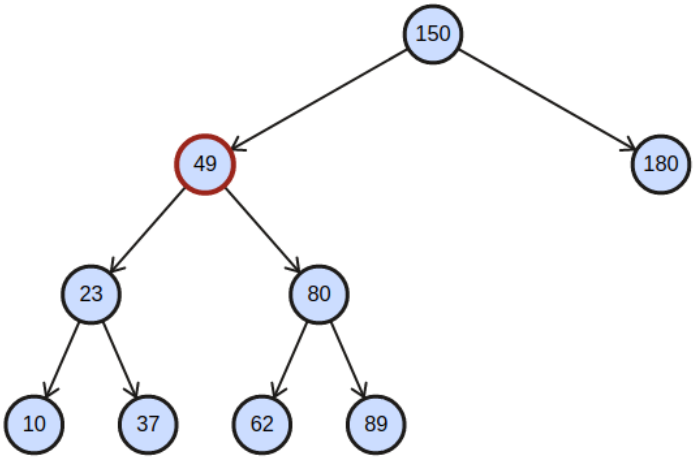
Input: Stablo $\mathcal{T}$ te dvije vrijednosti x, x' takve da je $x \leq x'$
Output: Čvor v (razdjelni čvor) gdje se pretraga dijeli na dva puta traženja, ili list gdje pretraga završava

```
1 v ← root(T)
2 while v nije list i vrijedi (x' ≤ x_v ili x > x_v) do
3     if x' ≤ x_v then
4         v ← lc(v)
5     else
6         v ← rc(v)
7 return v
```

Algoritam kao ulaz prima stablo $\mathcal{T}$ te dvije vrijednosti x, x' za koje vrijedi $x \leq x'$. U čvor v pohranjujemo korijen stabla $\mathcal{T}$. Ukoliko vrijedi tvrdnja da v nije list te je vrijednost od x' manja od x_v ili vrijednost od x strogo veća od x_v, izvršava se *while* petlja. Unutar *while* petlje, ako je desna vrijednost x' manja ili jednaka vrijednosti u čvoru v, tada u čvor v pohranjujemo lijevo dijete čvora v, odnosno kako je naznačeno u algoritmu $lc(v)$. Ako vrijedi uvjet $x > x_v$, tada u čvor v pohranjujemo desno dijete čvora v. Kada nije zadovoljen niti jedan uvjet *while* petlje, algoritam vraća čvor v.

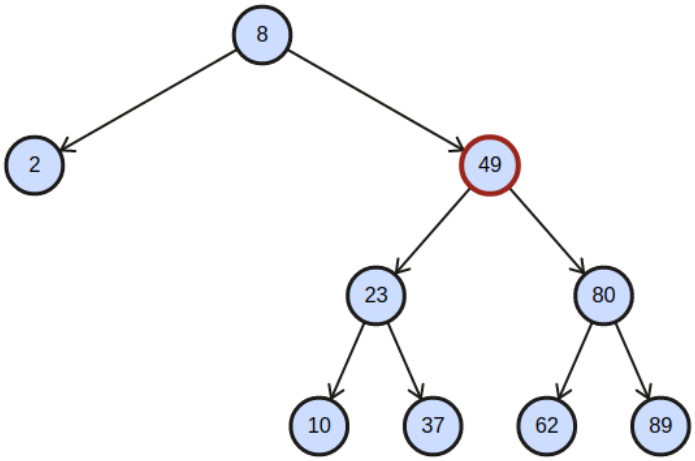
Za primjer rada algoritma, pogledajmo stablo $\mathcal{T}$ koje se sastoji od korijena s vrijednosti 150 prikazanog na slici 1.4, te vrijednosti $x = 18$ i $x' = 77$. Na početku, u čvoru v se nalazi referenca na korijen ulaznog stabla $\mathcal{T}$. Kako čvor v nije list i zadovoljen je barem jedan uvjet (u ovom slučaju samo prvi), ulazimo u *while* petlju. Prvi uvjet ($77 \leq 150$) je zadovoljen, stoga u varijablu v spremamo referencu na lijevo dijete čvora v s vrijednošću 49. Čvor v i dalje nije list, no sada nije zadovoljen niti jedan uvjet *while* petlje (77 nije manje ili jednako od 49, niti 18 nije strogo veće od 49). Došli smo do kraja algoritma, odnosno pronašli smo razdjelni čvor nakon čega se potraga za našim željenim točkama u intervalu grana na dva dijela. Pronađeni čvor je na slici 1.4 označen crvenim obrubom.

Primjer kada vrijedi uvjet $x > x_v$ označen je na slici 1.5. Na početku, ponovno ulazimo u *while* petlju, no ovaj put ne vrijedi prvi uvjet nego drugi ($18 > 8$). Zbog navedene razlike,



Slika 1.4: Algoritam OdrediRazdjelniČvor, slučaj $x' \leq x_v$, pri čemu je interval pretraživanja $[x, x'] = [18, 77]$

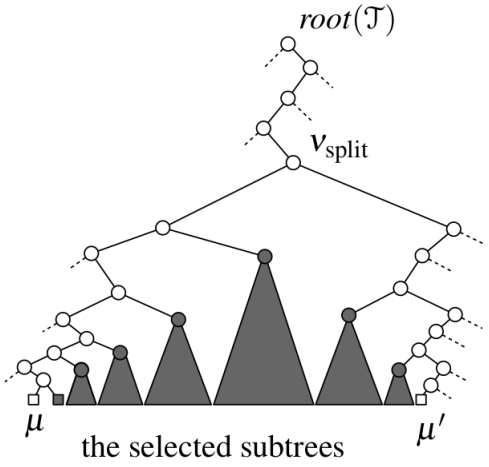
u čvor v pohranjujemo referencu na desno dijete čvora v s vrijednosti 49. Ostatak izvođenja algoritma je isti kao u prethodno navedenom primjeru.



Slika 1.5: Algoritam OdrediRazdjelniČvor, slučaj $x > x_v$, pri čemu je interval pretraživanja $[x, x'] = [18, 77]$

Sada, kad nam je poznati razdjelni čvor, počinjemo tražiti točke prema vrijednosti x, odnosno x'. Na svakom čvoru gdje put pretražuje lijevo podstablo, zapišemo točke iz svih listova desnog podstabla (jer se to podstablo nalazi između dva puta traženja). Slično, kada tražimo točke prema vrijednosti x', zapišemo točke iz svih listova lijevog podstabla u slučaju gdje put pretražuje desno podstablo. Na kraju, provjeravamo vrijednosti listova u kojima je pretraga završila jer one također mogu biti dio zadanog intervala.

Cijeli postupak pretraživanja jednodimenzionalnog raspona opisan je u *Algoritmu 2*. Taj algoritam koristi pomoćnu funkciju *DohvatiPodstablo(v)* čiji pseudokod ne navodimo, a koja prijavljuje sve listove koji se nalaze u podstablu čiji je korijen upravo ulazni čvor. Vizualni prikaz izvođenja algoritma možemo vidjeti na slici 1.6.



Izvor: Preuzeto iz cjeline 5.1 iz [2]

Slika 1.6: Algoritam PretraživanjeJednodimenzionalnogRaspona, v_{split} označava razdjelni čvor za interval pretraživanja $[\mu, \mu']$

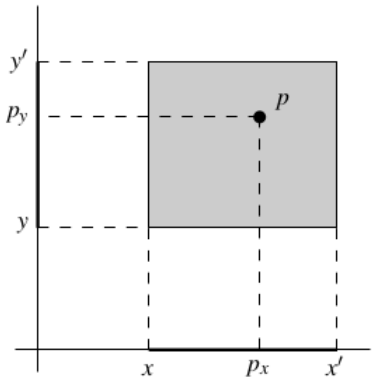
Promotrimo složenost upravo navedenog algoritma. U obzir ćemo uzeti ukupan broj točaka n te broj točaka k koje pripadaju zadanom intervalu. Metoda *DohvatiPodstablo* ima linearnu složenost u ovisnosti o broju točaka koje vraća jer prolazimo cijelim podstablom i vraćamo sve listove tog podstabla. Iz ovog slijedi da je složenost svih poziva metode *DohvatiPodstablo* upravo $O(k)$. Preostali čvorovi koje posjećujemo nalaze se na putu pretraživanja između x i x' te se u njima zadržavamo u konstantnom vremenu. Ti putevi imaju duljinu najviše $\log(n)$, stoga je ukupna složenost u tim čvorovima $O(\log n)$. Iz svega navedenog slijedi da algoritam *PretraživanjeJednodimenzionalnogRaspona* ima složenost $O(\log n + k)$.

Usporedimo složenost *Algoritma 2* i naivnog algoritma. U najgorem slučaju sve točke pripadaju zadanom intervalu, odnosno $k = n$, stoga složenost *Algoritma 2* iznosi $O(n)$.

Algorithm 2: PretraživanjeJednodimenzionalnogRaspona($\mathcal{T}$, $[x, x']$)

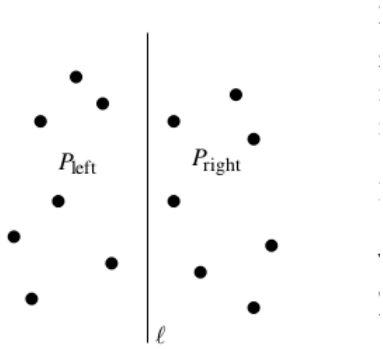
Input: Binarno stablo $\mathcal{T}$ i interval $[x, x']$
Output: Lista L svih točkaka koje se nalaze u stablu $\mathcal{T}$, a pripadaju intervalu $[x, x']$

```
1  v_r ← OdrediRazdjelniCvor(T, x, x')
2  if v_r je list then
3      if v_r ∈ [x, x'] then
4          L.insert(v_r)
5
6  else
       /* Kreći se po putu prema x i dodaj u listu točaka L sve
        točke podstabla koje se nalaze desno od puta traženja. */
7      v ← lc(v_r)
8      while v nije list do
9          if x ≤ x_v then
10             L.insert(DohvatiPodstablo(rc(v)))
11             v ← lc(v)
12         else
13             v ← rc(v)
14     if v ∈ [x, x'] then
15         L.insert(v)
16
       /* Kreći se po putu prema x' i dodaj u listu točaka L sve
        točke podstabla koje se nalaze lijevo od puta traženja. */
17     v ← rc(v_r)
18     while v nije list do
19         if x > x_v then
20             L.insert(DohvatiPodstablo(lc(v)))
21             v ← rc(v)
22         else
23             v ← lc(v)
24     if v ∈ [x, x'] then
25         L.insert(v)
26
27 return Listu točaka L
```



Izvor: Preuzeto iz cjeline 5.2 iz [2]

Slika 1.7: Točka u ravnini i 2-dimenzionalni interval u kojem leži



Izvor: Preuzeto iz cjeline 5.2 iz [2]

Slika 1.8: Podjela skupa P vertikalnom linijom na dva podskupa

Istu složenost ima i naivni algoritam pa ovdje ne možemo uočiti napredak. No, ukoliko postoje točke koje ne pripadaju zadanom intervalu, složenost *Algoritma 2* ima prednost nad naivnim algoritmom.

1.2 Kd-stablo

Pretraživanje jednodimenzionalnog raspona ima svoje prednosti, no problemi koji zahtijevaju višedimenzionalno pretraživanje puno su raširenija. Jedan primjer takvog višedimenzionalnog pretraživanja je problem pretraživanja 2-dimenzionalnog pravokutnog raspona. Neka je P ponovo skup točaka, ali ovaj put neka su točke dvodimenzionalne. Upitom dvodimenzionalnog pravokutnog raspona na skupu P tražimo točke koje pripadaju skupu P, a istovremeno se nalaze u pravokutniku određenih dimenzija $[x, x'] \times [y, y']$, pri čemu su $x, x', y, y' \in \mathbb{R}$. Točka $p := (p_x, p_y)$ se nalazi u pravokutniku $[x, x'] \times [y, y']$ ako i samo ako vrijedi $p_x \in [x, x']$ i $p_y \in [y, y']$ kako je i prikazano na slici 1.7.

Možemo reći da se upit 2-dimenzionalnog pravokutnog raspona sastoji od dva upita 1-dimenzionalnog pravokutnog raspona, jednog po x-koordinati danih točaka, a drugog po y-koordinati.

U prethodnom poglavlju, koristili smo određenu strukturu podataka za upite pravokutnog raspona - binarno stablo pretraživanja. Sada želimo generalizirati tu strukturu za upite 2-dimenzionalnog pravokutnog raspona. U tu svrhu, promotrimo rekurzivnu definiciju binarnog stabla traženja: skup (jednodimenzionalnih) točaka razdvojimo na dva podskupa

približno iste veličine. Jedan skup sadrži sve točke čija je vrijednost manja ili jednaka razdjelnoj vrijednosti, dok drugi sadrži sve točke čija je vrijednost strogo veća. Razdjelna vrijednost pohranjena je u korijenu stabla, dok su podskupovi rekurzivno pohranjeni u lijevo i desno podstablo.

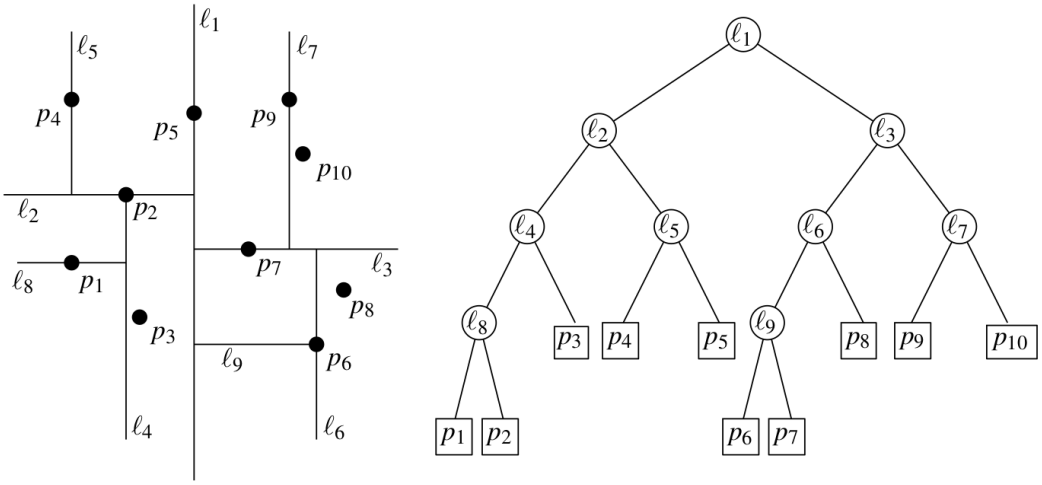
U dvodimenzionalnom slučaju, točke imaju dvije vrijednosti: x-koordinatu i y-koordinatu. Dakle, prvo dijelimo skup po x-koordinati, zatim po y-koordinati, pa ponovo po x i tako dalje. Dajemo precizniji postupak u nastavku. Zadani skup P dijelimo vertikalnom linijom ℓ (dakle po x-koordinati) na dva podskupa približno iste veličine. U korijen stabla pohranjujemo vertikalnu liniju ℓ, dok podskup P_{left} koji sadrži točke koje se nalaze lijevo od vertikalne linije (i na vertikalnoj liniji) ℓ pohranjujemo u lijevo podstablo, a podskup P_{right} koji sadrži sve točke koje se nalaze desno od linije ℓ pohranjujemo u desno podstablo (vidi sliku 1.8). U lijevom djetetu korijena podijelimo skup P_{left} na dva podskupa horizontalnom linijom. Sve točke koje se nalaze ispod ili na horizontalnoj liniji spremamo u lijevo podstablo lijevog djeteta, dok sve točke koje se nalaze iznad horizontalne linije stavljamo u desno podstablo istog djeteta. Lijevo dijete također pohranjuje i samu horizontalnu liniju. Slično, skup P_{right} također dijelimo horizontalnom linijom na dva podskupa koji su pohranjeni u lijevo i desno podstablo desnog djeteta. Kod unuka početnog čvora, ponovno dijelimo skupove vertikalnom linijom. Općenito, skup dijelimo vertikalnom linijom u čvorovima čija je dubina paran broj, dok u čvorovima neparne dubine skup dijelimo horizontalnom linijom. Na slici 1.9 možemo vidjeti podjelu početnog skupa i odgovarajuće binarno stablo. Takvo stablo nazivamo **kd-stablo** (eng. *kd-tree*). Općenito, to je binarno stablo u kojem je svaki list k-dimenzionalna točka. Unutarnji čvorovi dijele skup na dva podskupa u ovisnosti o vrijednosti pohranjenoj u čvoru, te ostatku pri dijeljenju dubine tog čvora s k. U upravo opisanom razmatranju, dan je primjer 2-d stabla. Danas izvorni naziv nije toliko u upotrebi pa umjesto 2-d stabla jednostavno koristimo naziv 2-dimenzionalno kd-stablo.

Implementacija kd-stabla

Kd-stablo implementiramo u programsku jeziku C++ te krajnji rezultat testiramo na slučajno generiranim skupovima dvodimenzionalnih točaka. Detaljnu implementaciju navodimo u nastavku.

Kako bismo efikasno implementirali kd-stablo, koristimo pomoćne strukture. Jedna od takvih struktura je `Range` koja reprezentira raspon, odnosno dvodimenzionalni interval $[x_1, x_2] \times [y_1, y_2]$ u kojem tražimo točke.

```
struct Range
{
    double x1, x2, y1, y2;
```



Izvor: Preuzeto iz cjeline 5.2 iz [2]

Slika 1.9: Točke u ravnini i pripadno kd-stablo

```
    Range(double, double, double, double);
    virtual ~Range();
};
```

Nadalje, struktura koja reprezentira dvodimenzionalnu točku zove se `Point`. Sastoji se od dvije vrijednosti `x` i `y` koje predstavljaju x i y koordinatu. Struktura `Point` sadrži metodu `isInRange` koja određuje nalazi li se točka u zadanom rasponu. Tu provjeru radimo tako da uspoređujemo vrijednosti koordinata s vrijednostima rubova zadanog intervala reprezentiranog varijablom tipa `Range`.

```
struct Point
{
    double x, y;

    Point();
    Point(double, double);
    virtual ~Point();

    bool operator==(Point);
    bool isInRange(Range *);
};
```

```
bool Point::isInRange(Range *range)
{
    if (this->x <= range->x2 && this->x >= range->x1 && this->y <=
    ↪  range->y2 && this->y >= range->y1)
    {
        return true;
    }

    return false;
}
```

Iduća važna struktura je `Region` koja se sastoji od četiri varijable p_1, p_2, p_3 i p_4 pomoću kojih ćemo definirati raspon točaka stabla. Ukoliko su sve vrijednosti definirane dobivamo pravokutnik (koji nije zatvoren sa svih strana, tj. ne sadrži čitav njegov rub), no ukoliko nisu, struktura predstavlja podskup ravnine ograničen pravcima. Točka se nalazi unutar regije (koja je reprezentirana strukturom `Region`) ukoliko je njena x-koordinata veća od x-koordinate varijable p_1 i manja od x-koordinate varijable p_2, no ako je x-koordinata jednaka barem jednoj x-koordinati pripadnih varijabli, onda gledamo y-koordinatu. Y-koordinata mora biti strogo veća od y-koordinate varijable p_1, te manja ili jednaka od varijable p_2. Analogno vrijedi za varijable p_3 i p_4 no tamo gledamo prvo po y-koordinati, a zatim po x-koordinati. Na taj način dobivamo pravokutnik čije sve stranice nisu nužno zatvorene. Na primjer, neka su definirane samo vrijednosti p_2 i p_3. Tada je naša struktura zapravo podskup ravnine koji obuhvaća sve točke čija je x-koordinata manja od x-koordinate točke p_2. Ukoliko su x-koordinate jednake, onda ravnina sadrži sve točke čija je y-koordinata manja ili jednaka y-koordinati točke p_2. Kako varijabla p_1 nije definirana, tada ne postoji minimalna vrijednost za x-koordinatu, stoga ona može biti proizvoljna. Isto tako, kako je definirana varijabla p_3, ravnina obuhvaća sve točke čija je y-koordinata veća od y-koordinate točke p_3. Također, ukoliko su y-koordinate jednake, onda ravnina obuhvaća točke čija je x-koordinata strogo veća od x-koordinate točke p_3. Ponovno, varijabla p_4 nije definirana pa ne postoji gornja granica za y-koordinatu točaka koje pripadaju danoj ravnini.

```
struct Region
{
    Point *p1, *p2, *p3, *p4;

    Region();
    Region(Point *, Point *, Point *, Point *);
    virtual ~Region();

    bool isContainedInRange(Range *);
```

```
    bool intersectsRange(Range *);
};
```

Navedena struktura ima dvije metode koje određuju je li regija sadržana u nekom 2-dimenzionalnom intervalu, te siječe li regija zadani interval. Za određivanje je li regija sadržana u dvodimenzionalnom rasponu služi metoda `isContainedInRange`. U toj metodi provjeravamo jesu li sve četiri varijable definirane. Ukoliko barem jedna nije, regija nije sadržana u intervalu (jer regija nije ograničena barem s jedne strane). Ako su sve varijable definirane, onda uspoređujemo granice intervala sa x i y koordinatama točaka p_1, p_2, p_3 i p_4. Metoda `intersectsRange` služi za provjeru presjeka sa zadanim intervalom. Regija siječe interval ako podskup ravnine definiran regijom ima presjek s podskupom ravnine definiranim intervalom. Ukoliko niti jedna od četiri varijable nije definirana, onda regija predstavlja cijelu ravninu pa također i siječe zadani interval. Upravo opisana struktura omogućit će nam efikasno pohranjivanje vertikalnih, odnosno horizontalnih linija.

Iduća pomoćna struktura je struktura `Node` koja predstavlja čvor stabla. Čvor se sastoji od pokazivača na lijevo i desno dijete (varijable `left` i `right`), vertikalne, tj. horizontalne linije tipa `Region` (varijabla `line`) te same vrijednosti točke (varijabla `point`). Linija čvora sadrži informaciju u kojem se rasponu mogu nalaziti točke podstabla što olakšava pretraživanje samog stabla. U ovoj strukturi postoje dva konstruktora od kojih jedan kao argument prima varijablu tipa `Point` koja služi za inicijalizaciju vrijednosti točke. Drugi konstruktor ne prima argumente te on postavlja obje koordinate točke na NaN, odnosno vrijednosti točke u tom slučaju nisu definirane. U oba konstruktora se sve četiri vrijednosti varijable `line` postavljaju na NaN.

```
struct Node
{
    Node *left, *right;
    Point *point;
    Region *line;

    Node();
    Node(Point);
    virtual ~Node();
};
```

Naša definicija kd-stabla sastoji se od korijena stabla, metoda za kreiranje i pretraživanje stabla, te metode za dohvaćanje točaka iz listova stabla. Definicija koristi navedene strukture, a samo kreiranje kd-stabla ostvarujemo primjenom rekurzivne metode.

```
class KDTree
{
```

```
    Node *root;

public:
    KDTree(std::vector<Point>);
    virtual ~KDTree();

    // Getter
    Node *getRoot()
    {
        return root;
    }

    // Setter
    void setRoot(Node *v)
    {
        root = v;
    }

    Node *buildKDTree(vector<Point>, vector<Point>, Region*, int);
    std::vector<Point> searchKDTree(Node *, Range *);
    std::vector<Point> reportSubtree(Node *);
};
```

Sada nam je cilj konstruirati 2-dimenzionalno kd-stablo što ćemo ostvariti primjenom metode pod nazivom `buildKDTree`. Metoda prima četiri argumenta, a vraća korijen kreiranog kd-stabla. Prva dva argumenta su skupovi, treći argument predstavlja granice pomoću kojih kreiramo liniju, a četvrti argument je broj koji nazivamo ***diskriminator*** (eng. *discriminator*). Prvi skup čine točke od kojih želimo izgraditi kd-stablo te su sortirane uzlazno po x-koordinati. Slično, drugi skup sačinjavaju iste točke kao i prethodni skup, samo su ovaj put sortirane uzlazno po y-koordinati. Treći argument su granice koje su reprezentirane pomoćnom strukturom `Region`, tj. sadrže četiri vrijednosti: lijevu granicu po x, desnu granicu po x, te lijevu i desnu granicu po y. Te četiri vrijednosti su reprezentirane strukturom `Point` te su koordinate točaka inicijalizirane na *NaN*. Četvrti argument je nenegativan cijeli broj koji označava dubinu rekurzije, odnosno dubinu korijena stabla u rekurzivnom pozivu metode. Taj broj nam je važan jer on označava trebamo li skup dijeliti vertikalnom ili horizontalnom linijom. Na početku algoritma, diskriminator je jednak nuli.

```
Node *KDTree::buildKDTree(const std::vector<Point> xSortedPoints, const
↪  std::vector<Point> ySortedPoints, Region *boundaries, const int depth)
{
```

```
if (xSortedPoints.size() == 1)
{
    return new Node(xSortedPoints.front());
}

int median = std::ceil(static_cast<double>(xSortedPoints.size()) / 2);

Node *node = new Node();
node->line = boundaries;

std::vector<Point> xPointsLeft;
std::vector<Point> xPointsRight;

std::vector<Point> yPointsLeft;
std::vector<Point> yPointsRight;
int subsetSizeCounter = 0;

Region *leftChildBoundaries;
Region *rightChildBoundaries;

if (depth % 2 == 0)
{
    node->point = new Point(xSortedPoints[median - 1].x,
    ↪  xSortedPoints[median - 1].y);

    xPointsLeft.assign(xSortedPoints.begin(), xSortedPoints.begin() +
    ↪  median);
    xPointsRight.assign(xSortedPoints.begin() + median,
    ↪  xSortedPoints.end());

    for (int i = 0; i < ySortedPoints.size(); ++i)
    {
        if (ySortedPoints[i].x <= xSortedPoints[median - 1].x &&
        ↪  subsetSizeCounter != median)
        {
            yPointsLeft.push_back(ySortedPoints[i]);
            subsetSizeCounter++;
        }
        else
        {
            yPointsRight.push_back(ySortedPoints[i]);
        }
```

```
        }

        leftChildBoundaries = new Region(boundaries->p1, node->point,
        ↪  boundaries->p3, boundaries->p4);

        rightChildBoundaries = new Region(node->point, boundaries->p2,
        ↪  boundaries->p3, boundaries->p4);
    }
    else
    {
        node->point = new Point(ySortedPoints[median - 1].x,
        ↪  ySortedPoints[median - 1].y);

        yPointsLeft.assign(ySortedPoints.begin(), ySortedPoints.begin() +
        ↪  median);
        yPointsRight.assign(ySortedPoints.begin() + median,
        ↪  ySortedPoints.end());

        for (int i = 0; i < xSortedPoints.size(); ++i)
        {
            if (xSortedPoints[i].y <= ySortedPoints[median - 1].y &&
            ↪  subsetSizeCounter != median)
            {
                xPointsLeft.push_back(xSortedPoints[i]);
                subsetSizeCounter++;
            }
            else
            {
                xPointsRight.push_back(xSortedPoints[i]);
            }
        }

        leftChildBoundaries = new Region(boundaries->p1, boundaries->p2,
        ↪  boundaries->p3, node->point);

        rightChildBoundaries = new Region(boundaries->p1, boundaries->p2,
        ↪  node->point, boundaries->p4);
    }

    node->left = buildKDTree(xPointsLeft, yPointsLeft, leftChildBoundaries,
    ↪  depth + 1);
```

```
    node->right = buildKDTree(xPointsRight, yPointsRight,
    ↪  rightChildBoundaries, depth + 1);

    return node;
}
```

Metoda na početku provjerava ukoliko dobiveni skup točaka sortiranih po x-koordinati sadrži samo jednu točku (skup točaka sortiranih po y-koordinati ima istu veličinu jer sadrži iste točke) te ako je navedeni uvjet istinit, metoda vraća novokreirani čvor u kojem je pohranjena vrijednost preostale točke skupa. Ako skupovi sadrže više od jedne točke, u varijablu `median` pohranjujemo medijan definiran formulom $\lceil n/2 \rceil$, pri čemu je n veličina skupa `xSortedPoints` koji je isti kao veličina skupa `ySortedPoints`. U varijablu `line` spremamo dobivenu varijablu `boundaries` iz argumenta funkcije, a ta varijabla predstavlja u kojem rasponu će se nalaziti točke podstabla čiji korijen (`node`) upravo kreiramo.

Dobiveni skup točaka dijelimo vertikalnom, odnosno horizontalnom linijom u ovisnosti o dubini rekurzije koju predstavlja varijabla `depth`. Ukoliko je `depth` parna, skup točaka dijelimo vertikalnom linijom, a u suprotnom slučaju skup dijelimo horizontalnom linijom. Pogledajmo slučaj dijeljenja skupa vertikalnom linijom. U varijablu `node->point` kreiranog čvora pohranjujemo točku koja se nalazi na indeksu medijan niza skupa `xPointsSorted`. Neka skup `xPointsLeft` sadrži sve točke sortirane po x-koordinati početnog skupa `xSortedPoints` koje se nalaze do medijana (isključivo), a `xPointsRight` sve točke od medijana (uključivo) do kraja zadanog skupa. Isto tako, skupovi `yPointsLeft` i `yPointsRight` sadrže iste točke kao skupovi `xPointsLeft` i `xPointsRight`, samo su točke sortirane po y-koordinati. Nadalje, određujemo raspon točaka djece koji pohranjujemo u varijablama `leftChildBoundaries` i `rightChildBoundaries`. Obje spomenute varijable sadrže isti raspon kao i trenutni čvor roditelj, no taj raspon se profinjuje ovisno o varijabli `node->point`. Kako lijevo dijete sadrži sve točke čija je x-koordinata manja od x-koordinate točke `node->point` (u slučaju jednakosti gledamo relaciju $\leq$ na y-koordinatama), tada varijabla `leftChildBoundaries` ima isti raspon kao i čvor roditelj, samo je desna granica za x-koordinatu (varijabla p_2) upravo varijabla `node->point`. Analogno, kako desno dijete sadrži sve točke čija je x-koordinata veća ili jednaka x-koordinati varijable `node->point` (u slučaju jednakosti x-koordinati, desno dijete sadrži sve točke čija je y-koordinata strogo veća od y-koordinate varijable `node->point`), varijabla `rightChildBoundaries` sadrži također isti raspon kao i roditeljski čvor, samo je lijeva granica za x-koordinatu (varijabla p_1) upravo točka `node->point`.

U slučaju dijeljenja horizontalnom linijom, varijabla `depth` je neparan broj, te je razmatranje analogno upravo navedenom, samo se sada u obzir uzima y-koordinata i skup sortiran po y-koordinati.

Na kraju, gradimo lijevo podstablo metodom `buildKDTree` te joj kao parametre prosljeđujemo kreirane skupove `xPointsLeft`, `yPointsLeft`, `leftChildBoundaries` te varijablu `depth` uvećanu za jedan. Rezultat izgradnje lijevog stabla je korijen te ga pohranjujemo u varijablu `left` koja reprezentira lijevo dijete trenutnog čvora. Analogno gradimo desno podstablo (sa skupovima `xPointsRight`, `yPointsRight`, `rightChildBoundaries` te brojem `depth` + 1) čiji čvor pohranjujemo u varijablu `right` koja predstavlja desno dijete čvora.

Kako je već ranije navedeno, metoda `buildKDTree` prima dva sortirana skupa prema x i y koordinati. Općenito, točke skupa P na početku nisu nužno sortirane redom koji zahtijeva gore opisana procedura te postoji mogućnost generiranja točaka s identičnim x i y koordinatama. Kako bi spriječili dupliciranje i unaprijed sortirali proizvoljan skup točaka, obavljamo pripremu prije samog poziva metode `buildKDTree`. Tu pripremu radimo u konstruktoru stabla tipa `KDTree` koji kao argument prima proizvoljni skup točaka `points`. Prvo sortiramo dobiveni skup prema x-koordinati (u čemu nam pomaže funkcija `sortByX`), zatim uklanjamo duplikate iz njega te kreiramo drugi skup `_points` koji je identičan upravo modificiranom skupu `points`. Nadalje, skup `_points` sortiramo prema y-koordinati, a uklanjanje točaka nije potrebno jer ne sadrži duplicirane točke. Varijabla `plane` predstavlja raspon u kojem se nalaze točke stabla koje upravo kreiramo, a kako smo na početku kreiranja, raspon je neograničen i predstavlja cijelu ravninu. Pozivamo metodu `buildKDTree` s navedenim varijablama te rezultat pohranjujemo u varijabli `root` čime završava izgradnja kd-stabla.

```
KDTree::KDTree(std::vector<Point> points)
{
    std::sort(points.begin(), points.end(), sortByX);
    points.erase(std::unique(points.begin(), points.end()),
    ↪  points.end());

    std::vector<Point> _points = points;
    std::sort(_points.begin(), _points.end(), sortByY);

    Region *plane = new Region();

    this->root = KDTree::buildKDTree(points, _points, plane, 0);
}

bool sortByX(const Point &a, const Point &b)
{
    if (a.x == b.x)
    {
```

```
        return a.y < b.y;
    }

    return (a.x < b.x);
}
```

Analizirajmo vrijeme potrebno za izgradnju kd-stabla. Ukoliko skup sadrži samo jednu točku, vraća se novo kreirani čvor te složenost cijelog algoritma iznosi $O(1)$. Ako algoritam prima više točaka, složenost je veća. Kako unaprijed imamo sortirane skupove točaka, medijan možemo pronaći u konstantnom vremenu. Nadalje, konstruiranje novih sortiranih lista potrebnih za dva rekurzivna poziva može se izvršiti u linearnom vremenu. Ukoliko je diskriminator paran, u skup `xPointsLeft` pohranjujemo točke od početka skupa `xSortedPoints` do medijana što se izvršava u linearnom vremenu. Analogno vrijedi i za skup `xPointsRight`. Kako bismo rasporedili točke u skupove `yPointsLeft` i `yPointsRight`, prolazimo kroz skup `ySortedPoints` u linearnom vremenu te u ovisnosti o vrijednost y-koordinate varijable `ySortedPoints[median - 1]`, pohranjujemo točke u ranije spomenute skupove. Analogni postupak se provodi u slučaju kada je diskriminator neparan. Iz svega navedenog, dobivamo:

$$T(n) = \begin{cases} O(1), & n = 1 \\ O(n) + 2T(\lceil n/2 \rceil), & n > 1, \end{cases}$$

odnosno

$$T(n) = \begin{cases} a, & n = 1 \\ b \cdot n + 2T(\lceil n/2 \rceil), & n > 1, \end{cases}$$

gdje su a i b konstante. Ova rekurzija dobro je definirana za svaki $n \in \mathbb{N}$, no teže se rješava zbog izraza $\lceil n/2 \rceil$. Ukoliko promatramo samo slučajeve u kojima je n potencija broja 2, rekurzija postaje:

$$T(n) = \begin{cases} a, & n = 1 \\ b \cdot n + 2T(n/2), & n > 1, n = 2^k, k > 0. \end{cases}$$

Supstitucijom $t_k = T(n) = T(2^k)$ dobivamo:

$$t_k = 2 \cdot t_{k-1} + b \cdot 2^k,$$

čime smo dobili nehomogenu rekurziju oblika:

$$a_m \cdot t_k + a_{m-1} \cdot t_{k-1} + ... + a_0 \cdot t_{k-m} = b_1^k \cdot p_{d_1}(k) + b_2^k \cdot p_{d_2}(k) + ... + b_l^k \cdot p_{d_l}(k).$$

Ovdje su b_i međusobno različite konstante, a $p_{d_i}(k)$ polinomi zadani u varijabli k stupnja d_i. Detalji su opisani u [4]. Ovakva rekurzija može se homogenizirati te dobivena homogena rekurzija ima karakterističnu jednadžbu oblika:

$$(a_m \cdot x^m + a_{m-1} \cdot x^{m-1} + ... + a_0) \cdot (x - b_1)^{d_1+1} \cdot ... \cdot (x - b_l)^{d_l+1} = 0.$$

Iz upravo opisanog možemo dobiti našu karakterističnu jednadžbu koja glasi

$$(x - 2) \cdot (x - 2) = 0,$$

odnosno

$$(x - 2)^2 = 0.$$

Opće rješenje je:

$$t_k = c_1 \cdot 2^k + c_2 \cdot k \cdot 2^k,$$

gdje su c_1 i c_2 konstante. Vratimo li u termin od n izraz $k = \log_2 n$, dobivamo:

$$T(n) = c_1 \cdot n + c_2 \cdot n \cdot \log_2 n.$$

Odavde zaključujemo $T(n) = O(n \cdot \log n)$, što vrijedi u slučaju kada je n potencija broja 2. Ovaj rezultat možemo generalizirati pod određenim uvjetima tako da vrijedi za svaki $n \in \mathbb{N}$. Traženi uvjeti su da $T(n)$ mora biti asimptotski rastuća te $f(n) = n \cdot \log n$ glatka funkcija. Prvi uvjet možemo dokazati indukcijom, dok je za drugi uvjet dovoljno dokazati da je $f(n)$ 2-glatka. Detalji i primjeri ovakvih dokaza mogu se naći u [4]. Iz svega prikazanog, dobivamo da je složenost izgradnje kd-stabla jednaka $O(n \cdot \log n)$, $\forall n \in \mathbb{N}$.

Kako bismo odredili prostornu složenost, primijetimo da svaki list stabla pohranjuje jednu točku, tj. imamo ukupno n listova. Kd-stablo je binarno stablo stoga svaki list i unutarnji čvor koristi $O(1)$ memorije iz čega slijedi da je ukupna prostorna složenost upravo $O(n)$.

Pretraživanje kd-stabla

Kako smo već najavili, jedan od ciljeva nam je pretražiti kd-stablo, odnosno pronaći točke koje pripadaju određenom rasponu. U svakom rekurzivnom pozivu prilikom kreiranja stabla, linijom dijelimo ravninu na dva dijela te jedan dio točaka pohranjujemo u lijevo, a ostatak u desno podstablo. 2-dimenzionalni interval koji označava koje se točke nalaze u podstablu smo pohranili u varijablu `line` čvora stabla, a proizvoljna točka se nalazi u podstablu ako i samo ako se nalazi u granicama tog intervala. Algoritam pretraživanja kd-stabla posjećuje samo one čvorove čiji interval siječe zadani raspon. Ako se interval čvora *node* u potpunosti nalazi u zadanom rasponu, onda sve točke podstabla od *node* pripadaju rasponu te ih pridodajemo rezultatu algoritma. U slučaju da smo dosegnuli list stabla,

provjeravamo spada li točka pohranjena u listu u zadani raspon. Navedeni algoritam implementiramo metodom `searchKDTree` koja prima čvor stabla koje pretražuje te raspon `range` koji predstavlja zatvoreni pravokutnik $[x_1, x_2] \times [y_1, y_2]$.

```
std::vector<Point> KDTree::searchKDTree(Node *node, Range *range)
{
    std::vector<Point> points;

    if (node->left == NULL && node->right == NULL)
    {
        if (node->point->isInRange(range))
        {
            points.push_back(*node->point);
        }

        return points;
    }

    Region *region = node->line;

    if (region->isContainedInRange(range))
    {
        std::vector<Point> subtreePoints = reportSubtree(node);
        points.insert(points.end(), subtreePoints.begin(),
        ↪  subtreePoints.end());
    }
    else if (region->intersectsRange(range))
    {
        std::vector<Point> subtreePointsLeft =
        ↪  searchKDTree(node->left, range);
        std::vector<Point> subtreePointsRight =
        ↪  searchKDTree(node->right, range);

        points.insert(points.end(), subtreePointsLeft.begin(),
        ↪  subtreePointsLeft.end());
        points.insert(points.end(), subtreePointsRight.begin(),
        ↪  subtreePointsRight.end());
    }
```

```
    return points;
}
```

Na samom početku metode, provjeravamo je li čvor list, odnosno ima li djecu. Ukoliko nema, metodom `isInRange` utvrđujemo pripada li njegova točka zadanom rasponu `range`. U slučaju da čvor nije list, dohvaćamo raspon točaka koji on sadrži te ga pohranjujemo u varijablu `region`. Ako je raspon sadržan u `range`, tada dohvaćamo sve točke listova stabla metodom `reportSubtree`. U slučaju da raspon siječe `range`, rekurzivno dohvaćamo rezultantne točke istoimenom metodom u dva poziva. Kao argumente, rekurzivnim metodama prosljeđujemo čvor koji reprezentira lijevo/desno dijete, zajedno sa zadanim rasponom `range`. Za provjeru sadrži li `range` raspon trenutnog čvora, odnosno postoji li presjek, pomažu nam već spomenute metode `isContainedInRange` te `intersectsRange`.

Za dohvaćanje svih točaka podstabla služi nam rekurzivna metoda `reportSubtree`. Metoda implementira algoritam koji se kreće po stablu, pronalazi sve listove te vraća točke pohranjene u njima kao rezultat. Na početku, provjeravamo jesmo li došli do lista te ukoliko jesmo, vraćamo točku koja je pohranjena u njemu. Zatim istoimenom metodom tražimo listove lijevog djeteta i rezultat pridružujemo skupu već pronađenih točaka `points`. Isti postupak radimo i za desno dijete te na kraju vraćamo sve pronađene točke.

```
std::vector<Point> KDTree::reportSubtree(Node *node)
{
    std::vector<Point> points;

    if (node->left == NULL && node->right == NULL)
    {
        points.push_back(*node->point);
        return points;
    }

    auto leftPoints = reportSubtree(node->left);
    points.insert(points.end(), leftPoints.begin(),
    ↪  leftPoints.end());

    auto rightPoints = reportSubtree(node->right);
    points.insert(points.end(), rightPoints.begin(),
    ↪  rightPoints.end());

    return points;
}
```

Navedena razmišljanja mogu se generalizirati na veće dimenzije. Tada čvorovi sadrže k-dimenzionalne točke, pri čemu je k dimenzija prostora čije stablo želimo kreirati, a pretraživanje se provodi na sličan način kao u 2-dimenzionalnom slučaju.

Promotrimo sada složenost pretraživanja kd-stabla. Ukoliko stablo sadrži samo jednu točku, algoritam vraća traženu točku te je složenost konstantna. Ako stablo sadrži više točaka, potrebna je detaljnija analiza. Otprije nam je poznata složenost metode `reportSubtree` te ona iznosi $O(k)$ pri čemu je k ukupan broj vraćenih točaka podstabla. Ovime smo odredili složenost metode u čvorovima čije su regije u potpunosti sadržane u zadanom intervalu. Preostaje nam odrediti broj čvorova koji nisu dio metode `reportSubtree`. Za svaki takav čvor v, zadani pravokutni raspon `range` siječe regiju od v, odnosno presjek postoji, no regija čvora nije u potpunosti sadržana u rasponu. Kako bismo analizirali broj takvih čvorova, potrebno je odrediti broj regija presječenih bilo kojom vertikalnom linijom. Tako ćemo dobiti gornju granicu broja regija presječenih lijevom, odnosno desnom stranicom zadanog pravokutnog raspona. Broj regija presječenih gornjom, odnosno donjom stranicom raspona možemo dobiti na sličan način (gledanjem horizontalne linije).

Neka je l proizvoljna vertikalna linija te neka je $\mathcal{T}$ kd-stablo s n čvorova u čijem korijenu prvo pohranjujemo vertikalnu liniju. Neka je $l(root)$ linija pohranjena u korijenu kd-stabla. Linija l siječe ili regiju lijevog djeteta, ili regiju desnog djeteta korijena stabla, ali nikako oboje. Isto tako, linija l siječe obje regije djece presječenog djeteta. Na primjer, ako l siječe regiju lijevog djeteta, onda siječe i obje regije djece lijevog djeteta jer lijevo dijete ima pohranjenu horizontalnu liniju u čvoru. Definirajmo $T(n)$ kao broj svih regija kd-stabla koje siječe l. Kako bismo napisali rekurziju za $T(n)$, moramo otići dvije razine niže u stablu jer gledamo čvorove koji imaju pohranjenu vertikalnu liniju. Svaki od četiri čvora na razini 2 stabla sadrži $n/4$ točaka, pri čemu se korijen inicijalnog čvora nalazi na razini 0. (Preciznije, svaki čvor je podstablo koje sadrži najviše $\lceil \lceil n/2 \rceil /2 \rceil = \lceil n/4 \rceil$ točaka, ali to ne utječe na konačni rezultat.) Regije dva od četiri čvora su presječene, stoga rekurzivno određujemo broj presječenih regija u tim podstablima. Također, l siječe regiju korijena i regiju jednog djeteta koje ima pohranjenu horizontalnu liniju, zbog čega imamo konstantu 2 u dobivenoj rekurziji:

$$T(n) = \begin{cases} O(1), & n = 1 \\ 2 + 2T(n/4), & n > 1. \end{cases}$$

Supstituiramo $n = 2^k$ te dobivamo rekurziju

$$T(2^k) = 2 + 2T(2^{k-2}),$$

gdje je $k > 2$. Uz oznaku $t_k = T(2^k) = T(n)$ slijedi,

$$t_k = 2 + 2t_{k-2},$$

odnosno

$$t_k - 2t_{k-2} = 2.$$

Karakteristična jednadžba glasi:

$$(x^2 - 2) \cdot (x - 1) = 0,$$

tj.

$$(x - \sqrt{2}) \cdot (x + \sqrt{2}) \cdot (x - 1) = 0.$$

Opće rješenje rekurzije je

$$c_1 \cdot (\sqrt{2})^k + c_2 \cdot (-\sqrt{2})^k + c_3 \cdot 1^k.$$

Vraćajući n, dobivamo

$$T(n) = c_1 \cdot \sqrt{n} + c_2 \cdot \sqrt{n} + c_3,$$

čime dobivamo $T(n) = O(\sqrt{n})$. Ta tvrdnja vrijedi za svaki $n \in \mathbb{N}$ uz zadovoljene uvjete da je $T(n)$ asimptotski rastuća, te $f(n) = \sqrt{n}$ glatka funkcija čiji se detalji dokazivanja mogu naći u [4].

Na sličan način dobivamo da ista složenost vrijedi i u slučaju horizontalne linije. Naime, neka je l proizvoljna horizontalna linija, a $\mathcal{T}$ kd-stablo s n čvorova u čijem korijenu prvo pohranjujemo vertikalnu liniju. $T(n)$ ovaj puta definiramo kao broj svih regija kd-stabla koje siječe l. Ponovo moramo otići dvije razine niže kako bismo obuhvatili ispravne čvorove. Isto tako, linija l siječe regiju korijena te regije oba djeteta (radimo u korijenu vertikalnu podjelu, no l je horizontalna linija stoga siječe obje regije). Zbog svega navedenog, rekurzija glasi:

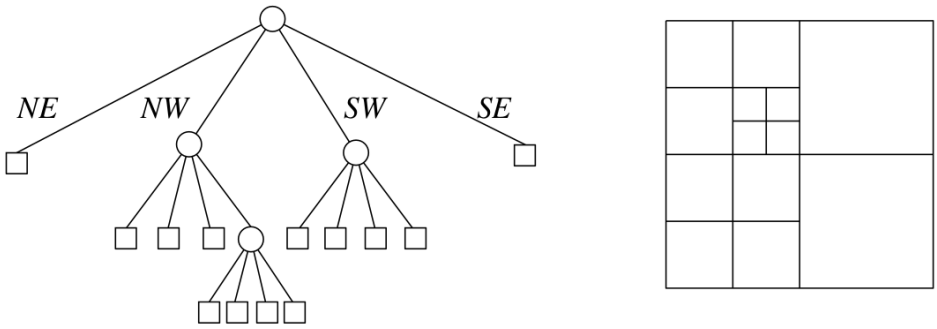
$$T(n) = \begin{cases} O(1), & n = 1 \\ 3 + 2T(n/4), & n > 1, \end{cases}$$

te se rješava na način koji je sličan prethodno opisanom postupku gdje je l bio vertikalna linija, a konačni rezultat je identičan. Iz svega dobivenog slijedi da je složenost pretraživanja kd-stabla upravo $O(\sqrt{n} + k)$.

Kao što smo mogli primijetiti u upravo navedenom objašnjenju, varijablu k dobivamo zbog činjenice da moramo pronaći i vratiti sve točke u navedenom rasponu. U slučaju da bismo trebali vratiti samo broj točaka u rasponu (ne i stvarne točke), tada bi složenost iznosila samo $O(\sqrt{n})$. U tom slučaju bismo u svakom čvoru pohranili broj koji označava koliko listova stablo ima, stoga bismo tu informaciju dohvaćali u konstantnom vremenu.

1.3 Quadtree

Iduća struktura koju promatramo je **quadtree**. Quadtree je stablo čiji unutarnji čvor ima točno četiri djeteta, a svaki čvor predstavlja kvadrat. Ukoliko čvor ima djecu, tada njihovi kvadrati odgovaraju podjeli roditeljskog kvadrata na četiri kvadranta, iz čega možemo vidjeti podrijetlo naziva stabla. Odavde možemo primijetiti kako listovi stabla zajedno formiraju subdiviziju roditeljskog čvora. Takvu subdiviziju nazivamo **quadtree subdivizija** (eng. *quadtree subdivision*), a primjer možemo vidjeti na slici 1.10. Djeca korijena imaju oznake *NE*, *NW*, *SW* i *SE* koje nagoviještavaju koji kvadrant im pripada. *NE* označava sjeveroistočni kvadrant (eng. *north-east quadrant*), *NW* sjeverozapadni (eng. *north-west quadrant*), *SW* jugozapadni, a *SE* jugoistočni kvadrant.



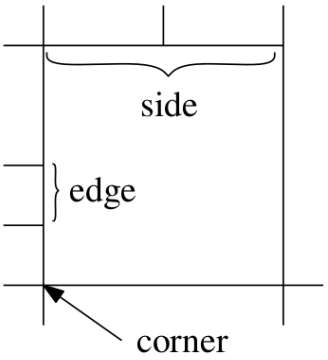
Izvor: Preuzeto iz cjeline 14.2 iz [2]

Slika 1.10: Quadtree i pripadna subdivizija

Quadtree subdivizijom dobivamo manje kvadrate. Struktura kvadrata prikazana je na slici 1.11. Svaki kvadrat ima četiri vrha koji se nazivaju **vrhovi kuta** (eng. *corner vertices*), skraćeno kutevi. Dužine koje spajaju kuteve kvadrata nazivamo **stranice kvadrata** (eng. *sides of squares*). Rubovi subdivizije koji pripadaju nekom kvadratu zovu se **rubovi kvadrata** (eng. *edges of the square*), stoga stranica kvadrata sadrži barem jedan, a može sadržavati i više rubova. Kažemo da su dva kvadrata **susjedna** (eng. *neighbors*) ukoliko dijele rub.

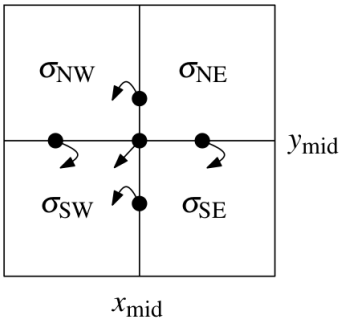
Kao i kd-stablo, quadtree služi za pohranu točaka ravnine. U ovom slučaju, podjela kvadrata na četiri dijela događa se sve dok kvadrat sadrži više od jedne točke. Neka je $\mathcal{T}$ quadtree te P skup točaka unutar kvadrata $\sigma := [x_\sigma, x'_\sigma] \times [y_\sigma, y'_\sigma]$.

Ako je $|P| \leq 1$, tada se quadtree sastoji od samo jednog lista u kojem je pohranjen skup P i kvadrat σ. Ukoliko skup P sadrži više točaka, definiramo središnju točku s koordinatama $x_{mid} := (x_\sigma + x'_\sigma)/2$ i $y_{mid} := (y_\sigma + y'_\sigma)/2$ te označimo sa σ_{NE}, σ_{NW}, σ_{SW} i σ_{SE} četiri



Izvor: Preuzeto iz cjeline 14.2 iz [2]

Slika 1.11: Dijelovi kvadrata



Izvor: Preuzeto iz cjeline 14.2 iz [2]

Slika 1.12: Subdivizija kvadrata i pripadnost točaka pojedinom kvadrantu

kvadranta kvadrata σ. Definiramo skupove točaka za pripadne kvadrante:

$$P_{NE} := \{\, p := (p_x, p_y) \in P \mid p_x > x_{mid} \quad \& \quad p_y > y_{mid} \,\},$$

$$P_{NW} := \{\, p := (p_x, p_y) \in P \mid p_x \leq x_{mid} \quad \& \quad p_y > y_{mid} \,\},$$

$$P_{SW} := \{\, p := (p_x, p_y) \in P \mid p_x \leq x_{mid} \quad \& \quad p_y \leq y_{mid} \,\},$$

$$P_{SE} := \{\, p := (p_x, p_y) \in P \mid p_x > x_{mid} \quad \& \quad p_y \leq y_{mid} \,\}.$$

Quadtree se sada sastoji od korijena v u kojem je pohranjen kvadrat σ kojeg ćemo označiti sa $\sigma(v)$. Nadalje, v ima četvero djece:

- *NE*-dijete je korijen quadtree-a za skup točaka P_{NE} unutar kvadrata σ_{NE},
- *NW*-dijete je korijen quadtree-a za skup točaka P_{NW} unutar kvadrata σ_{NW},
- *SW*-dijete je korijen quadtree-a za skup točaka P_{SW} unutar kvadrata σ_{SW},
- *SE*-dijete je korijen quadtree-a za skup točaka P_{SE} unutar kvadrata σ_{SE}.

Korištenje operatora $\leq$, $<$ i $>$ u definiciji skupova P_{NE}, P_{NW}, P_{SW} i P_{SE} određuje kojem kvadrantu pripadaju točke na rubovima subdivizije. Na ovaj način, vertikalne linije pripadaju lijevim, a horizontalne linije donjim kvadrantima kao što možemo vidjeti na slici 1.12.

Quadtree je rekurzivno definiran iz čega dobivamo rekurzivan algoritam za njegovu izgradnju: podijelimo kvadrat na četiri manja kvadranta, particioniramo skup točaka na

skupove za određeni kvadrant, te rekurzivno izgradimo quadtree za svaki kvadrant s njegovim pripadnim skupom točaka. Rekurzija staje kada je kardinalitet zadanog skupa točaka strogo manji od dva. Primijetimo kako kvadrat ne mora biti zadan na početku algoritma. U tom slučaju, izračunamo dimenzije najmanjeg kvadrata koji sadrži sve točke dobivenog skupa točaka.

Implementacija strukture Quadtree

Uz već definirane strukture koje smo koristili u implementaciji kd-stabla, definirat ćemo još jednu pomoćnu strukturu pod nazivom `QuadData` koja će nam olakšati implementaciju strukture Quadtree. Navedena struktura pohranjuje informacije specifične za svaki čvor stabla, poput dimenzije kvadrata te točku u kvadratu (ukoliko kvadrat sadrži točku). Struktura ima dva konstruktora. Prvi konstruktor prima dva argumenta od kojih je jedan tipa `Point` i služi za pohranjivanje točke, a drugi je pokazivač na `Range` i pomoću njega inicijaliziramo kvadrat čvora stabla. Drugi konstruktor prima samo jedan argument, tj. pokazivač na `Range` te su koordinate točke postavljene na NaN. Prvim konstruktorom definiramo potrebne informacije za list stabla, dok drugim definiramo informacije za unutarnji čvor. Struktura sadrži metodu `findMiddlePoint` koja vraća točku u sredini kvadrata.

```
struct QuadData
{
    Point *point;
    Range *square;

    QuadData(Point, Range *);
    QuadData(Range *);
    virtual ~QuadData();

    Point findMiddlePoint();
};

Point QuadData::findMiddlePoint()
{
    double xMid = (this->square->x1 + this->square->x2) / 2;
    double yMid = (this->square->y1 + this->square->y2) / 2;

    return Point(xMid, yMid);
}
```

Sada smo spremni za implementaciju strukture Quadtree. Definiramo novu klasu `QuadTree` koja predstavlja korijen stabla te sadrži nekoliko varijabli od kojih je jedna tipa

QuadData. Ona pohranjuje informacije o kvadratu korijena stabla te ukoliko se u tom kvadratu nalazi samo jedna točka, ona je također pohranjena u varijabli data. Nadalje, kao što je navedeno u definiciji quadtree-a, klasa sadrži i četiri varijable NE, NW, SW, SE koje predstavljaju djecu korijena stabla te varijablu parent kojom označavamo roditeljski čvor kojem trenutni čvor pripada. Klasa sadrži jedan konstruktor, nekoliko getter-a i setter-a, te metode čije ćemo funkcionalnosti opisati u nastavku.

```
class QuadTree
{
    QuadData *data;

    QuadTree *NE, *NW, *SW, *SE;
    QuadTree *parent;

public:
    QuadTree(std::vector<Point>, Range *, QuadTree *);
    virtual ~QuadTree();

    // Getter
    QuadData *getData()
    {
        return data;
    }

    // Setter
    void setData(QuadData *v)
    {
        data = v;
    }

    QuadTree *getNE()
    {
        return NE;
    }

    QuadTree *getNW()
    {
        return NW;
    }
```

```
    QuadTree *getSW()
    {
        return SW;
    }
    QuadTree *getSE()
    {
        return SE;
    }

    void divideNode(std::vector<Point>);
    QuadTree *findNorthNeighbor(QuadTree *);
    QuadTree *findSouthNeighbor(QuadTree *);
    QuadTree *findEastNeighbor(QuadTree *);
    QuadTree *findWestNeighbor(QuadTree *);
    void balanceQuadtree();
    std::list<QuadTree *> getAllLeaves();
    bool isSubdivisionNeeded(QuadTree *);
    bool hasLargerSquare(Range *);
};
```

Rekurzivno kreiranje quadtree-a odvija se pomoću konstruktora koji prima tri argumenta. Skup točaka iz kojih gradimo quadtree dobivamo u prvom argumentu, drugi argument predstavlja kvadrat u kojem se nalaze točke te treći predstavlja pokazivač na roditeljsko stablo. Na početku, u varijablu `this->parent` spremamo dobiveni roditeljski čvor, a zatim provjeravamo kardinalitet dobivenog skupa točaka. Ukoliko taj skup sadrži samo jednu točku, došli smo do kraja rekurzivnog algoritma za kreiranje quadtree-a te pohranjujemo točku i dobiveni kvadrat u varijablu `this->data`. Kako više nema dijeljenja kvadrata, sve vrijednosti djece trenutnog čvora postavimo na `NULL`. Ukoliko je skup točaka prazan, tada smo također došli do kraja rekurzivnog algoritma, no ovaj puta ne pohranjujemo točku, odnosno naše trenutno stablo predstavlja list bez točke. Ako skup točaka sadrži dvije ili više točaka, potrebno je podijeliti dobiveni kvadrat na četiri dijela te rekurzivno kreirati quadtree za svako dijete. Prvo, u varijablu `this->data` pohranjujemo kvadrat za trenutno stablo te pozivamo metodu `divideNode` koja obavlja upravo spomenutu podjelu.

```
QuadTree::QuadTree(std::vector<Point> points, Range *square,
↪   QuadTree *parent)
{
    this->parent = parent;

    if (points.size() == 1)
```

```
    {
        this->data = new QuadData(points.front(), square);
        this->NE = NULL;
        this->NW = NULL;
        this->SW = NULL;
        this->SE = NULL;

        return;
    }

    if (points.size() == 0)
    {
        this->data = new QuadData(square);
        this->NE = NULL;
        this->NW = NULL;
        this->SW = NULL;
        this->SE = NULL;

        return;
    }

    this->data = new QuadData(square);

    this->divideNode(points);
}
```

Metoda `divideNode` dijeli trenutni kvadrat stabla na četiri manja kvadrata, particionira skup točaka za pripadne kvadrate te svakom djetetu pridjeljuje vrijednost dobivenu kreiranjem novog quadtree-a sa pripadnim skupom točaka, kvadratom i roditeljskim stablom. Na početku, definiramo točku u središtu kvadrata pomoću već spomenute metode `findMiddlePoint`. Zatim, pomoću metode `partitionPoints` čije implementacijske detalje izostavljamo, rasporedimo točke dobivenog skupa `points` na skupove točaka za svaki od četiri kvadrata. Također, definiramo dimenzije svakog od četiri kvadrata pomoću varijable `middlePoint` te u `this->data->point` pohranjujemo točku koja ima nedefinirane koordinate čime naznačujemo da se radi o unutrašnjem čvoru, a ne listu stabla. Na kraju, svakom djetetu pridjeljujemo vrijednost dobivenu kreiranjem novog quadtree-a sa pripadnim argumentima. Na primjer, `this->NE` će postati novi quadtree koji obuhvaća sve točke navedene u `pointsNE`, kvadrat mu je definiran varijablom `squareNE`, a roditeljsko stablo je upravo ono čiju podjelu trenutno radimo.

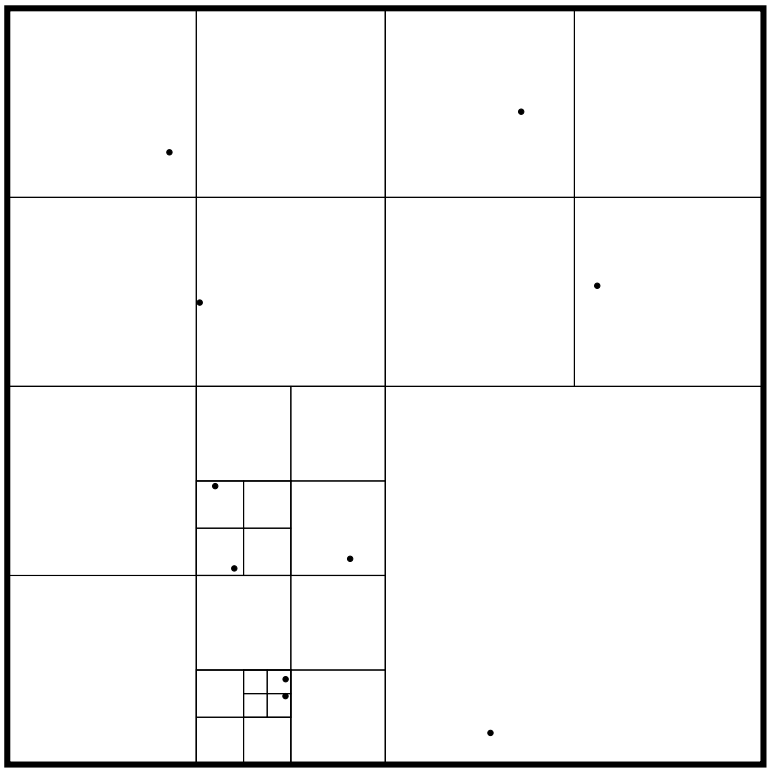
```
void QuadTree::divideNode(std::vector<Point> points)
{
    Point middlePoint = this->data->findMiddlePoint();
    std::vector<Point> pointsNE;
    std::vector<Point> pointsNW;
    std::vector<Point> pointsSW;
    std::vector<Point> pointsSE;
    partitionPoints(middlePoint, points, pointsNE, pointsNW,
    ↪  pointsSW, pointsSE);

    Range *squareNE = new Range(middlePoint.x,
    ↪  this->data->square->x2, middlePoint.y,
    ↪  this->data->square->y2);
    Range *squareNW = new Range(this->data->square->x1,
    ↪  middlePoint.x, middlePoint.y, this->data->square->y2);
    Range *squareSW = new Range(this->data->square->x1,
    ↪  middlePoint.x, this->data->square->y1, middlePoint.y);
    Range *squareSE = new Range(middlePoint.x,
    ↪  this->data->square->x2, this->data->square->y1,
    ↪  middlePoint.y);

    this->NE = new QuadTree(pointsNE, squareNE, this);
    this->NW = new QuadTree(pointsNW, squareNW, this);
    this->SW = new QuadTree(pointsSW, squareSW, this);
    this->SE = new QuadTree(pointsSE, squareSE, this);
    this->data->point = new Point();
}
```

Kako bismo konstruirali quadtree, potreban nam je samo skup točaka. U metodi prije kreiranja stabla izbacimo duplicirane točke (ukoliko one postoje) te izračunamo najmanji kvadrat koji sadrži sve točke iz zadanog skupa. U tome nam pomaže metoda `computeSquare` koja vraća takav kvadrat. Sada imamo sve potrebne argumente za konstruktor klase `QuadTree`. Naravno, u inicijalnom pozivu roditeljski čvor ima vrijednost `NULL`.

```
Range *computeSquare(std::vector<Point> &points)
{
    if (points.size() == 0)
    {
        return new Range();
```

```
    }

    double xMin = points.front().x;
    double xMax = points.front().x;
    double yMin = points.front().y;
    double yMax = points.front().y;

    for (const auto &value : points)
    {
        if (value.x < xMin)
        {
            xMin = value.x;
        }
        if (value.x > xMax)
        {
            xMax = value.x;
        }
        if (value.y < yMin)
        {
            yMin = value.y;
        }
        if (value.y > yMax)
        {
            yMax = value.y;
        }
    }

    double lengthX = xMax - xMin;
    double lengthY = yMax - yMin;

    if (lengthX > lengthY)
    {
        return new Range(xMin, xMax, yMin, yMin + lengthX);
    }
    else
    {
        return new Range(xMin, xMin + lengthY, yMin, yMax);
    }
}
```

Primjenom upravo opisanih metoda na deset slučajno generiranih točaka, dobivamo quadtree koji možemo vidjeti na slici 1.13.



Slika 1.13: Quadtree

Promotrimo sada složenost izgradnje quadtree-a. Jedina operacija koju bismo trebali detaljnije pogledati unutar jednog rekurzivnog poziva jest podjela zadanog skupa točaka na četiri skupa točaka za svaki kvadrant (ostale operacije izvode se u konstantnom vremenu). U metodi koja dijeli točke zapravo prolazimo kroz zadani skup točaka i, u ovisnosti o x-koordinati i y-koordinati točke, raspoređujemo točku u pripadni kvadrant, odnosno u skup točaka za pripadni kvadrant. Podjela se odvija samo u unutarnjim čvorovima, a oni sadrže dvije ili više točaka. Štoviše, kvadrati čvorova na istoj dubini međusobno su disjunktni i njihovom unijom dobivamo početni, inicijalni kvadrat. Zaključujemo da je ukupan broj točaka na određenoj dubini koje pripadaju unutarnjim čvorovima najviše n iz čega slijedi da je složenost quadtree izgradnje upravo $O(d \cdot n)$, gdje je d dubina stabla.

Također, uočimo da je dubina stabla za skup točaka P najviše $\log(s/c) + 3/2$, gdje je c najmanja udaljenost između bilo koje dvije točke skupa P te s duljina dužine stranice incijalnog kvadrata koji sadrži P. Naime, kada se spustimo razinu niže, sa čvora na njegovo dijete, duljina odgovarajuće stranice kvadrata se prepolovi iz čega slijedi da je duljina

stranice kvadrata čvora na dubini i jednaka $s/2^i$. Najveću udaljenost između točaka u kvadratu možemo izračunati korištenjem dijagonale kvadrata koja iznosi $s\sqrt{2}/2^i$ za kvadrat čvora na dubini i. Kako svaki unutarnji čvor stabla ima najmanje dvije točke u kvadratu te je minimalna udaljenost točaka jednaka c, dubina i unutarnjeg čvora mora zadovoljavati nejednakost

$$s \cdot \sqrt{2}/2^i \geq c.$$

Iz nejednakosti slijedi

$$i \leq \log(s \cdot \sqrt{2}/c) = \log(s/c) + 1/2.$$

Iz činjenice da je dubina quadtree-a za jedan veća od maksimalne dubine unutarnjeg čvora, dobivamo da je dubina stabla jednaka upravo $\log(s/c) + 1/2 + 1 = \log(s/c) + 3/2$.

Često korištena metoda na strukturi quadtree je metoda traženja susjeda. Zadan nam je čvor v (zapravo referenca na stablo u našem slučaju) i smjer koji može biti *sjever*, *jug*, *istok*, *zapad*. Označimo sa $\sigma(v)$ kvadrat od v. Metoda pokušava pronaći susjedni čvor v' takav da je $\sigma(v')$ susjedni kvadrat kvadratu $\sigma(v)$ te da se v i v' nalaze na istoj dubini. Ako takav čvor ne postoji, onda nastoji pronaći najdublji čvor čiji je kvadrat susjedan kvadratu $\sigma(v)$. Postoji mogućnost da niti takav čvor ne postoji i to se događa u slučaju da je rub kvadrata čvora v sadržan u rubu inicijalnog kvadrata te tada algoritam vraća `NULL`.

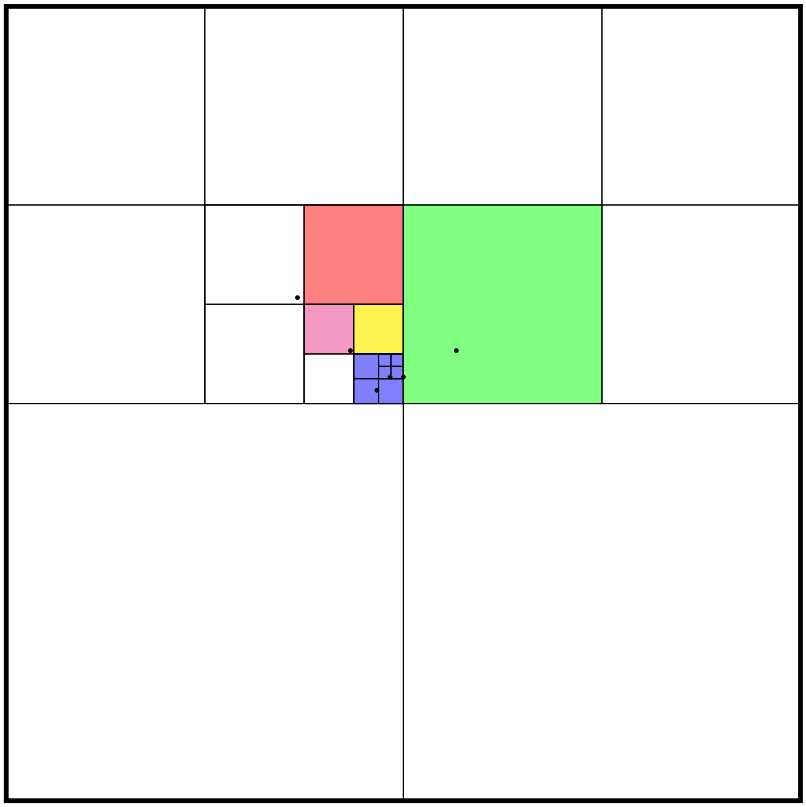
U ovisnosti o smjeru u kojem želimo pronaći susjeda, implementirali smo traženje u četiri zasebne metode. Te metode su `findNorthNeighbor`, `findSouthNeighbor`, `findEastNeighbor` i `findWestNeighbor`. Mi ćemo objasniti `findNorthNeighbor`, tj. objasnit ćemo traženje sjevernog susjeda zadanog čvora, a ostale metode se izvode na sličan način. Metoda prima samo jedan argument `node` i to je upravo referenca na čvor čiji susjedni kvadrat želimo pronaći. Ukoliko je `node` korijen zadanog stabla na kojem pozivamo metodu, tada vraćamo `NULL`. U varijablu `parentTree` spremamo roditeljsko stablo od varijable `node`. Ukoliko je `node` upravo `SW` dijete od `parentTree`, tada je sjeverni susjed `parentTree->NW`. Isto tako, ukoliko je `node` jednak `parentTree->SE`, tada je traženi susjed varijabla `parentTree->NE`. U slučaju da je `node` jednak `parentTree->NW` ili `parentTree->NE`, tada želimo pronaći sjevernog susjeda od roditeljskog čvora jer će jedno njegovo dijete biti traženi rezultat. U varijablu `northNeighbor` spremimo sjevernog susjeda od `parentTree`. Ukoliko on ne postoji, znači da je gornji rub od kvadrata varijable `node` zapravo dio ruba inicijalnog kvadrata cijelog quadtree-a te njegov sjeverni susjed ne postoji. Ukoliko je `northNeighbor` list, tada vraćamo njega jer je on najdublji čvor koji se nalazi sjeverno od `node`. U slučaju da `northNeighbor` ima djecu, tada vraćamo `northNeighbor->SW` ako je `node` jednak `parentTree->NW`, ili `northNeighbor->SE` u slučaju da je `node` sjeverozapadno dijete svog roditelja.

```
QuadTree *QuadTree::findNorthNeighbor(QuadTree *node)
{
    if (node == this)
```

```
    {
        return NULL;
    }

    QuadTree *parentTree = node->parent;

    if (parentTree->SW == node)
    {
        return parentTree->NW;
    }

    if (parentTree->SE == node)
    {
        return parentTree->NE;
    }

    QuadTree *northNeighbor = this->findNorthNeighbor(parentTree);

    if (northNeighbor == NULL || (northNeighbor->NE == NULL &&
    ↪  northNeighbor->NW == NULL && northNeighbor->SW == NULL &&
    ↪  northNeighbor->SE == NULL))
    {
        return northNeighbor;
    }
    else if (node == parentTree->NW)
    {
        return northNeighbor->SW;
    }
    else
    {
        return northNeighbor->SE;
    }
}
```

Na slici 1.14 možemo vidjeti zadani čvor obojan žutom bojom te sve njegove susjede. Sjeverni susjed obojan je crvenom bojom, južni plavom, istočni zelenom, a zapadni rozom bojom.

U svakom pozivu, složenost metode `findNorthNeighbor` iznosi $O(1)$ ukoliko izuzmemo rekurzivne pozive. Označimo sa d dubinu stabla. Korijen stabla ima dubinu jednaku nuli. Svakim rekurzivnim pozivom, dubina argumenta snizi se za jedan. Iz svega spome-



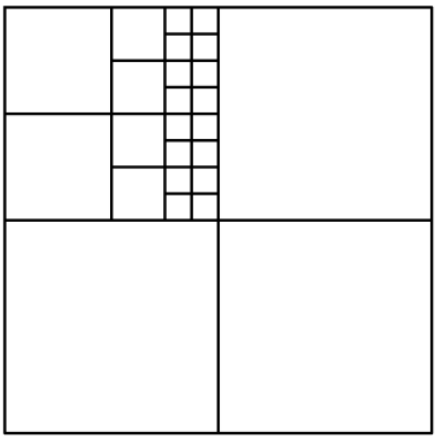
Slika 1.14: Susjedi čvora zadanog žutom bojom

nutog slijedi da složenost spomenute metode ovisi samo o dubini quadtree-a, tj. složenost metode za pronalazak sjevernog susjeda iznosi $O(d)$. Ista složenost vrijedi i za metode `findSouthNeighbor`, `findEastNeighbor` te `findWestNeighbor`.

Promatranjem slike 1.15 možemo primijetiti kako quadtree može biti poprilično neuravnotežen. Zbog toga, veliki kvadrati mogu biti susjedi kvadratima koji su nekoliko puta manji od njih. Ovo svojstvo je često neželjeno (pogotovo u kreiranju triangulacija, čime se bavimo u idućem poglavlju) pa se pojavila potreba za kreiranjem **balansiranog quadtree-a** (eng. *balanced quadtree*).

Implementacija balansiranog quadtree-a

Kažemo da je subdivizija quadtree-a balansirana ako se dužine rubova svaka dva susjedna kvadrata razlikuju najviše za faktor 2. Quadtree je balansiran ukoliko ima balansiranu subdiviziju. Svaka dva lista balansiranog quadtree-a, čiji su kvadrati susjedni, mogu se razlikovati najviše za jedan u dubini stabla.



Izvor: Preuzeto iz cjeline 14.2 iz [2]

Slika 1.15: Nebalansirani quadtree

Unatoč možda nepovoljnom rasporedu točaka, moguće je dobiti balansirano stablo. Dijeljenjem kvadrata čija je dubina bitno manja od susjednog kvadrata, postupno uravnotežujemo stablo te smanjujemo razlike u dubini, odnosno balansiramo stablo.

Prije opisivanja implementacije balansiranog quadtree-a, navodimo metodu `getAllLeaves` kojom dohvaćamo sve listove stabla. List prepoznajemo po tome što nema definiranu djecu, odnosno varijable `NE`, `NW`, `SW` i `SE` postavljene su na `NULL`. Ukoliko je čvor list stabla, stavljamo ga u listu svih listova pod nazivom `leaves` te istoimenu listu vraćamo. U suprotnom, pronalazimo sve listove koje sadrže djeca trenutnog stabla. Njihove pronađene listove također dodajemo u `leaves` te ju vraćamo kao konačni rezultat.

```
std::list<QuadTree *> QuadTree::getAllLeaves()
{
    std::list<QuadTree *> leaves;

    if (this->NE == NULL)
    {
        leaves.push_back(this);
        return leaves;
    }

    auto leavesNE = this->NE->getAllLeaves();
    leaves.insert(leaves.end(), leavesNE.begin(), leavesNE.end());
```

```
    auto leavesNW = this->NW->getAllLeaves();
    leaves.insert(leaves.end(), leavesNW.begin(), leavesNW.end());

    auto leavesSW = this->SW->getAllLeaves();
    leaves.insert(leaves.end(), leavesSW.begin(), leavesSW.end());

    auto leavesSE = this->SE->getAllLeaves();
    leaves.insert(leaves.end(), leavesSE.begin(), leavesSE.end());

    return leaves;
}
```

Balansirani quadtree kreiramo u metodi `balanceQuadTree` iz već postojećeg quadtree-a. Na početku dohvaćamo sve listove koje sadrži već kreirani (nebalansirani) quadtree te ih pospremamo u `leaves`. Dokle god je lista `leaves` neprazna, provodimo sljedeći postupak: uzmemo list koji se nalazi na početku liste `leaves`, pohranimo ga u varijablu `leaf` te ga izbacimo iz `leaves`. Zatim provjeravamo trebamo li podijeliti kvadrat lista na manje dijelove pomoću metode `isSubdivisionNeeded`. Ta metoda provjerava graniči li `leaf` barem s jednim kvadratom čija je duljina ruba manja ili jednaka četvrtini duljine ruba kvadrata varijable `leaf`. Ukoliko takav kvadrat postoji, `isSubdivisionNeeded` vraća `true`. Naime, ako je subdivizija potrebna, `leaf` postaje korijen novog quadtree-a sa četiri djeteta, a novokreirana djeca su zapravo listovi. Isto tako, ako je `leaf` sadržavao točku, određeno dijete ima pohranjenu točku u skladu s dimenzijama kvadrata. Sva četiri djeteta ubacimo u `leaves` te pronađemo sve susjede varijable `leaf`. Za svakog pronađenog susjeda provjerimo pomoću metode `hasLargerSquare` treba li on sada biti podijeljen. Susjed treba biti podijeljen u slučaju da mu je dužina ruba kvadrata bila veća od dužine ruba kvadrata varijable `leaf` (prije subdivizije). U tom slučaju, kvadrat od varijable `leaf` sada smo podijelili na četiri manja, stoga se dužine susjednih kvadrata razlikuju barem za faktor 4 što je nedopustivo u balansiranom stablu. Ako takav susjed postoji, ubacimo ga u listu `leaves` te postupak ponavljamo dokle god ona sadrži listove.

```
void QuadTree::balanceQuadtree()
{
    std::list<QuadTree *> leaves = this->getAllLeaves();

    while (leaves.size())
    {
        QuadTree *leaf = leaves.front();
        leaves.pop_front();
```

```
if (this->isSubdivisionNeeded(leaf))
{
    std::vector<Point> points;
    if (!std::isnan(leaf->getData()->point->x))
    {
        points.push_back(*leaf->getData()->point);
    }

    leaf->divideNode(points);

    leaves.push_back(leaf->NE);
    leaves.push_back(leaf->NW);
    leaves.push_back(leaf->SW);
    leaves.push_back(leaf->SE);

    QuadTree *northNeighbor = this->findNorthNeighbor(leaf);
    QuadTree *southNeighbor = this->findSouthNeighbor(leaf);
    QuadTree *eastNeighbor = this->findEastNeighbor(leaf);
    QuadTree *westNeighbor = this->findWestNeighbor(leaf);

    if (northNeighbor &&
    ↪  northNeighbor->hasLargerSquare(leaf->getData()->square))
    {
        leaves.push_back(northNeighbor);
    }
    if (southNeighbor &&
    ↪  southNeighbor->hasLargerSquare(leaf->getData()->square))
    {
        leaves.push_back(southNeighbor);
    }
    if (eastNeighbor &&
    ↪  eastNeighbor->hasLargerSquare(leaf->getData()->square))
    {
        leaves.push_back(eastNeighbor);
    }
    if (westNeighbor &&
    ↪  westNeighbor->hasLargerSquare(leaf->getData()->square))
    {
```

```
                leaves.push_back(westNeighbor);
            }
        }
    }
}
```

Objasnimo detaljnije `isSubdivisionNeeded` koju smo koristili u kreiranju balansiranog stabla. Metoda odlučuje trebamo li rastaviti kvadrat na četiri manja kvadrata. Zadani kvadrat trebamo rastaviti ukoliko graniči s kvadratom čija je duljina stranice manja ili jednaka četvrtini duljine stranice zadanog kvadrata. Pretpostavimo da je duljina kvadrata sa sjeverne strane kvadrata zadanog čvora jednaka četvrtini duljine zadanog kvadrata. Naša metoda traži sjevernog susjeda zadanog čvora. Ukoliko on postoji te ima jugoistočno ili jugozapadno dijete koje nije list, tada zadani kvadrat graniči s manjim kvadratom čija je duljina jednaka četvrtini ili manje duljine stranice zadanog kvadrata te je subdivizija potrebna. Primjer takvog slučaja možemo vidjeti na slici 1.16. Kvadrat kojemu je potrebno napraviti subdiviziju označen je žutom bojom. Njegov sjeverni susjed prikazan je zelenom bojom. Sjeverni susjed ima jugoistočno dijete koje nije list (sadrži djecu) stoga rastavljamo početni kvadrat, a podjela na manje kvadrate prikazana je točkastim linijama. Sličan postupak se primjenjuje u slučaju kada se manji kvadrati nalaze s južne, istočne, odnosno zapadne strane.

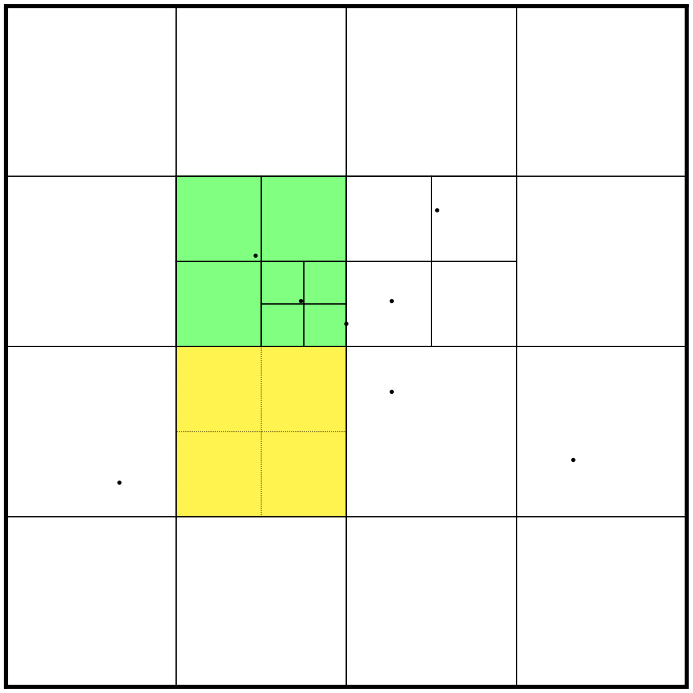
```
bool QuadTree::isSubdivisionNeeded(QuadTree *node)
{
    QuadTree *northNeighbor = this->findNorthNeighbor(node);
    QuadTree *southNeighbor = this->findSouthNeighbor(node);
    QuadTree *eastNeighbor = this->findEastNeighbor(node);
    QuadTree *westNeighbor = this->findWestNeighbor(node);

    if ((northNeighbor && northNeighbor->SW &&
    ↪  northNeighbor->SW->NE) || (northNeighbor &&
    ↪  northNeighbor->SE && northNeighbor->SE->NE))
    {
        return true;
    }

    if ((southNeighbor && southNeighbor->NW &&
    ↪  southNeighbor->NW->NE) || (southNeighbor &&
    ↪  southNeighbor->NE && southNeighbor->NE->NE))
    {
        return true;
```

```
    }

    if ((eastNeighbor && eastNeighbor->NW && eastNeighbor->NW->NE)
    ↪  || (eastNeighbor && eastNeighbor->SW &&
    ↪  eastNeighbor->SW->NE))
    {
        return true;
    }

    if ((westNeighbor && westNeighbor->NE && westNeighbor->NE->NE)
    ↪  || (westNeighbor && westNeighbor->SE &&
    ↪  westNeighbor->SE->NE))
    {
        return true;
    }

    return false;
}
```

Složenost `isSubdivisionNeeded` ovisi samo o metodama za traženje susjeda jer sve ostalo su provjere uvjeta. Znamo da složenosti metoda za traženje susjeda iznose $O(d)$, stoga je složenost metode `isSubdivisionNeeded` jednaka $O(d)$.

Posljednja potrebna metoda za kreiranje balansiranog stabla koju ćemo opisati je `hasLargerSquare`. Metodu pozivamo na čvoru koji predstavlja korijen quadtree-a te mu u argumentu proslijeđujemo drugi quadtree s čijim kvadratom želimo uspoređivati. Ukoliko je duljina stranice početnog kvadrata veća od duljine stranice kvadrata stabla dobivenog u argumentu, metoda vraća `true`. U suprotnom, `hasLargerSquare` vraća `false`. Metoda samo uspoređuje duljine stranica kvadrata stoga je njena složenost konstantna.

```
bool QuadTree::hasLargerSquare(Range *square)
{
    if ((this->getData()->square->x2 - this->getData()->square->x1)
    ↪  > (square->x2 - square->x1))
    {
        return true;
    }

    return false;
}
```

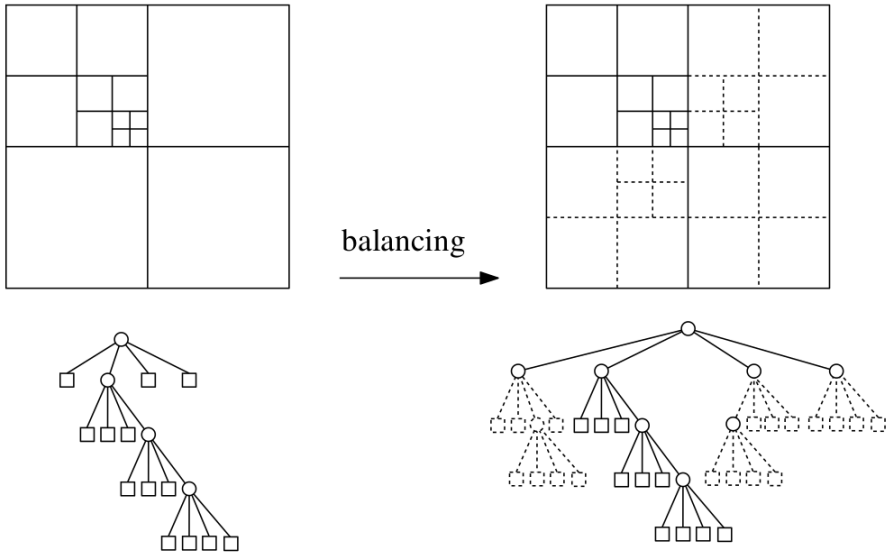


Slika 1.16: Primjer subdivizije kvadrata

Primjenom svega navedenog, pozivom metode `balanceQuadtree` dobivamo balansirani quadtree, a primjer možemo vidjeti na slici 1.17. Originalna subdivizija prikazana je punom linijom, a profinjenje potrebno za balansiranost točkastom linijom.

Analizirajmo sada složenost izgradnje balansiranog stabla i ukupan broj čvorova. Dokazujemo da balansirana verzija ima ukupno $O(m)$ čvorova te je složenost izgradnje jednaka $O(d \cdot m)$, pri čemu je d dubina stabla, a m ukupan broj čvorova inicijalnog quadtree-a. Označimo sa $\mathcal{T}$ quadtree, a njegovu balansiranu verziju sa $\mathcal{T}_B$. Balansirano stablo dobijemo iz početnog quadtree-a tako da zamijenimo određene listove s unutarnjim čvorovima koji sadrže četiri lista. To ćemo ostvariti na način da podijelimo određeni list na četiri dijela, odnosno kreiramo četiri nova podstabla koji predstavljaju listove. Dokazat ćemo da je za dobivanje balansirane verzije potrebno maksimalno $8m$ takvih podjela. Kako jedna podjela povećava ukupan broj čvorova (unutarnjih i listova) za četiri, slijedi da balansirano stablo ima $O(m)$ čvorova.

Nazovimo kvadrate u $\mathcal{T}$ starim kvadratima, a kvadrate koji se nalaze u $\mathcal{T}_B$, ali ne i u $\mathcal{T}$ novim kvadratima. Pretpostavimo da moramo podijeliti kvadrat σ (stari ili novi) u procesu dobivanja balansiranog stabla. Kvadranti od σ možda trebaju dalje biti podijeljeni, ali to gledamo zasebno. Ovdje samo pratimo porast broja čvorova za četiri u slučaju podjele kvadrata σ. Uskoro ćemo dokazati da je barem jedan od 8 kvadrata iste veličine koji



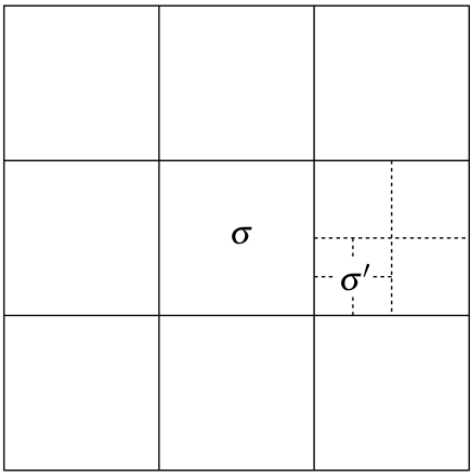
Izvor: Preuzeto iz cjeline 14.2 iz [2]

Slika 1.17: Quadtree i njegova balansirana verzija

okružuju σ stari kvadrat. Podjelu kvadrata σ povezujemo s jednim od tih starih kvadrata. Svaki takav stari kvadrat, odnosno svaki čvor $\mathcal{T}$ s kojim smo ga povezali, sudjeluje u najviše osam podjela, stoga je ukupan broj podjela najviše $8m$ kako smo i tvrdili.

Sada dokazujemo da je svaki kvadrat, koji se dijeli u procesu balansiranja, okružen barem jednim starim kvadratom od 8 kvadrata iste veličine. Pretpostavimo suprotno, odnosno neka je σ najmanji kvadrat koji nije okružen niti jednim starim kvadratom od osam kvadrata iste veličine. Kako σ mora biti podijeljen, on mora biti susjedan kvadratu čija je duljina stranice manja od polovine duljne stranice kvadrata σ. Neka je σ' kvadrat čija je duljina točno jednaka polovini duljine stranice od σ te σ' sadrži mali kvadrat koji uzrokuje podjelu kvadrata σ (slika 1.18). Po pretpostavci je σ' sadržan u novom kvadratu, stoga je i on sam novi kvadrat iz čega slijedi da je morao biti podijeljen u procesu balansiranja. Primijetimo da su svi kvadrati koji okružuju σ' novi jer su ili sadržani u kvadratima koji okružuju σ (a takvi kvadrati su novi), ili su sadržani u σ (koji se dijeli u procesu balansiranja). Iz ovoga slijedi da je σ' manji kvadrat od σ, također se dijeli u procesu balansiranja te su svi kvadrati koji ga okružuju novi. Time smo dobili kontradikciju s našom pretpostavkom, stoga iz svega navedenog dobivamo da je svaki kvadrat koji se dijeli okružen barem s jednim starim kvadratom. Iz svega navedenog dobivamo da je maksimalan broj podjela listova u procesu balansiranja manji ili jednak $8m$.

Preostaje nam dokazati da je složenost izgradnje balansiranog stabla jednaka $O(d \cdot m)$.



Izvor: Preuzeto iz cjeline 14.2 iz [2]

Slika 1.18: Inicijalni kvadrat σ i osam kvadrata iste veličine koji ga okružuju

Složenost koju ima svaki čvor u slučaju dijeljenja na kvadrante jednaka je $O(d)$ što dobivamo iz složenosti metode `isSubdivisionNeeded`, `getAllLeaves` i metoda za traženje susjeda koje traju $O(d)$ te metoda `divideNode` (metoda samo podijeli kvadrat na četiri manja, ne ulazi dublje u rekurziju) i `hasLargerSquare` čije su složenosti konstantne. Kako je svaki čvor tretiran u algoritmu samo jednom te je ukupan broj čvorova $O(m)$, slijedi da je složenost izgradnje balansiranog stabla jednaka $O(d \cdot m)$.

Poglavlje 2

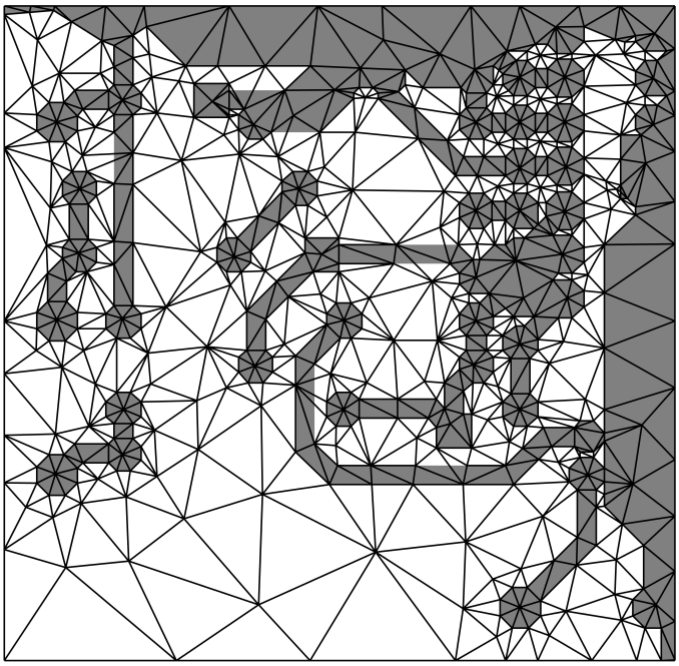
Generiranje triangulacija

Dosad smo opisali dvije različite strukture: kd-stablo i quadtree. Te strukture imaju različite primjene, a mi ćemo se posvetiti primjeni quadtree-a. Jedna od primjena quadtree-a je pohranjivanje slika. Slike često imaju dijelove određene veličine koje sadrže istu boju. Quadtree može pohraniti informaciju o boji dijela slike u jednom čvoru jer svaki čvor sadrži dimenzije kvadrata (u ovom slučaju dimenzije dijela slike koji sadrži istu boju). To je jednostavnije i brže nego da se informacija o svakom pikselu slike pohranjuje u dvodimenzionalnom nizu, odnosno matrici.

Iduća primjena je u brzom dobivanju unije/presjeka dviju slika. Imamo dvije slike u binarnom obliku te želimo dobiti njihovu uniju (eng. *overlay*). Binarne slike sadrže samo dvije boje: crnu i bijelu. Unijom dobivamo sliku u kojoj je piksel crne boje ukoliko barem jedna slika ima crni piksel na lokaciji koju proučavamo. Ako preformuliramo, piksel u rezultantnoj slici je bijele boje ako i samo ako je odgovarajući piksel u objema ulaznim binarnim slikama bijele boje, inače je piksel obojan u crno. Umjesto da obavljamo operaciju piksel po piksel, uniju/presjek možemo efikasnije izračunati upotrebom quadtree-a čiji čvor može pohranjivati više piksela odjednom. Za dvije ulazne slike napravimo odgovarajuće quadtree-eve te paralelnim prolaskom kroz njih gradimo rezultantni quadtree koji odgovara rezultantnoj slici, odnosno uniji dviju ulaznih binarnih slika. Na sličan način dobivamo i presjek binarnih slika.

Više o ostalim primjenama quadtree-a i popratnim algoritmima možemo pronaći u [3], a mi se ovdje specijaliziramo za korištenje quadtree-a u generiranju triangulacija, odnosno mreža koje se sastoje od trokuta. Općenito, **mreža** (eng. *mesh*) geometrijskih oblika dobiva se podjelom geometrijskog prostora na diskretne oblike, a primjer možemo vidjeti na slici 2.1. Postoji više različitih vrsta mreža, a one ovise o načinu na koji dijelimo prostor.

Gotovo svi električni uređaji sadrže elektroničke sklopove koji kontroliraju njihovo funkcioniranje. Ti sklopovi (otpornici, VLSI sklopovi i slične komponente) smješteni su na tiskanoj pločici (eng. *printed circuit board*). Kako bismo dizajnirali tiskanu pločicu,



Izvor: Preuzeto iz cjeline 14.3 iz [2]

Slika 2.1: Primjer triangulacije na tiskanoj pločici s komponentama

moramo odlučiti kako ćemo rasporediti komponente i kako ćemo ih međusobno povezati. Jedno od rješenja za upravo opisani problem koristi kreiranje mreža.

Mnoge se komponente na tiskanoj pločici zagrijavaju tijekom rada te zrače toplinu. Kako bi pločica ispravno radila, zračenje topline mora biti unutar dopuštene granice. Poprilično je teško unaprijed predvidjeti hoće li zračenje topline uzrokovati probleme jer to ovisi o relativnom rasporedu komponenti i njihovoj povezanosti. U prošlosti, trebao se dizajnirati jedan raspored komponenti, zatim se eksperimentom provjeravalo je li zračenje topline u normalnim granicama te ukoliko nije, radio se alternativni dizajn pločice. Danas, ovakvi se eksperimenti mogu simulirati. Kreiranje dizajna sada je automatizirano, stoga je računalni model tiskane pločice brzo napravljen i spreman za simulaciju. Sve zajedno traje puno kraće od stvarnog kreiranja prototipa pločice i provođenja eksperimenata na njemu.

Prijenos topline između različitih materijala na tiskanoj pločici je poprilično kompliciran proces. Kako bismo simulirali procese prijenosa topline, moramo koristiti aproksimacije koje koriste metodu konačnih elemenata (eng. *finite element method*). Takva metoda prvo podijeli pločicu na više manjih regija, odnosno na više manjih elemenata. Ti elementi su obično trokuti ili četverokuti, a toplina koju emitira svaki element je unaprijed poznata. Također, poznato je i kako susjedni elementi utječu jedni na druge. Sve opisano dovodi do

velikog sustava jednadžbi koji se rješava numerički.

Preciznost metode konačnih elemenata uvelike ovisi o mreži: ukoliko je mreža detaljnija, rješenje će biti preciznije. Negativna strana predetaljne mreže jest da se vrijeme računanja numeričkih procesa drastično povećava s povećanjem broja elemenata. To trebamo imati na umu te trebamo koristiti detaljnu mrežu samo u situacijama gdje je to zaista potrebno (na primjer, na granicama regija različitih materijala).

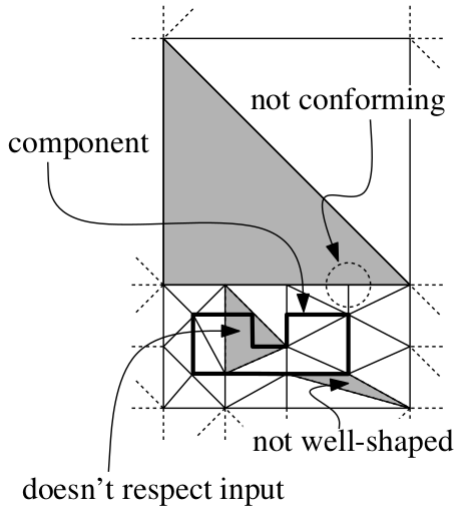
2.1 Uniformne i neuniformne triangulacije

Mi ćemo promatrati sljedeću varijantu problema kreiranja mreža: kao ulaz dobivamo kvadrat (koji reprezentira tiskanu pločicu) u kojem se nalazi različiti broj disjunktnih poligona (komponenti). Ponekad se kvadrat zajedno s komponentama naziva **domena** (eng. *domain*) mreže. Vrhovi kvadrata su točke $(0, 0)$, $(0, U)$, $(U, 0)$ i (U, U) gdje je $U = 2^k$ pri čemu je k prirodni broj. Pretpostavljamo da su vrhovi komponenti točke sa cjelobrojnim koordinatama čije su vrijednosti između 0 i U (uključivo). Također pretpostavljamo da bridovi komponenti imaju samo četiri različite orijentacije, tj. kut koji zatvara brid komponente sa x-osi može imati samo vrijednosti 0°, 45°, 90° ili 135°.

Cilj nam je izračunati **triangulaciju** (eng. *triangular mesh*) kvadrata, tj. subdiviziju kvadrata na trokute s određenim svojstvima koja ćemo sada navesti, a njihovu vizualizaciju možemo vidjeti na slici 2.2. Zahtijevamo da triangulacija ima sljedeća svojstva:

- Triangulacija mora biti **konformna** (eng. *conforming*): vrh trokuta ne smije biti unutrašnja točka stranice nekog drugog trokuta.
- Triangulacija mora **poštovati ulazne podatke** (eng. *respect the input*): bridovi komponenti moraju biti sadržani u uniji svih bridova trokuta.
- Triangulacija mora biti **dobro oblikovana** (eng. *well-shaped*): kut bilo kojeg trokuta triangulacije ne smije biti niti prevelik niti premalen. Zahtijevamo da vrijednost kuta bude između 45° i 90°.
- Triangulacija mora biti **neuniformna** (eng. *non-uniformn*): zahtijevamo da je mreža detaljnija blizu rubova komponenti, a grublja kako se udaljavamo od komponente. Time zadovoljavamo svojstvo da profinjujemo mrežu samo ukoliko situacija to zahtijeva.

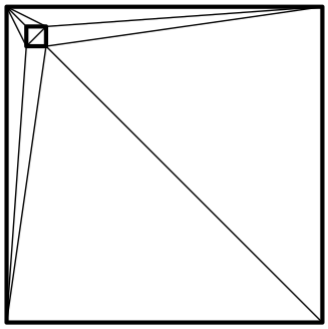
Postoji više načina kako možemo podijeliti kvadrat i komponente unutar njega na trokute. Jedan način je primjenom Delaunayjeve triangulacije čiji se detalji mogu pronaći u [2]. Primjenom Delaunayjeve trinagulacije možemo dobiti kuteve trokuta koji su premaleni. Pogledajmo primjer na slici 2.3: Na početku definiramo kvadrat sa stranicom duljine 16. Unutar kvadrata nalazi se samo jedna komponenta koja je zapravo mali kvadrat sa



Izvor: Preuzeto iz cjeline 14.3 iz [2]

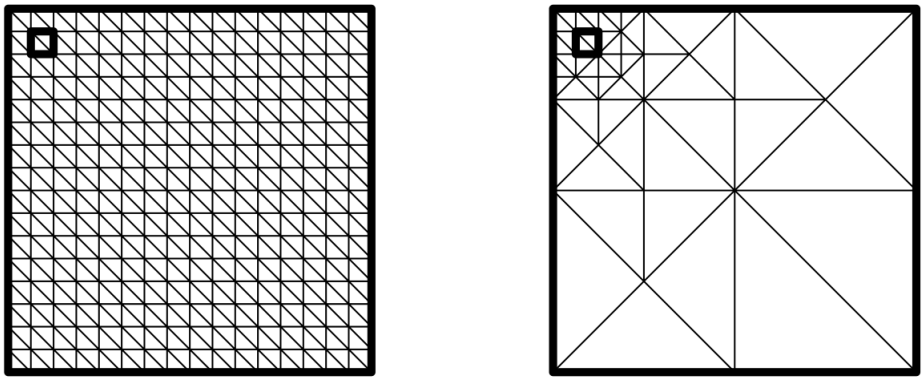
Slika 2.2: Svojstva triangulacije

stranicom duljine 1. Mali kvadrat nalazi se u gornjem lijevom dijelu zadanog kvadrata, na udaljenosti 1 od lijevog i gornjeg ruba zadanog kvadrata. Delaunayjevom triangulacijom dobivamo triangulaciju koji nije dobro oblikovana, tj. sadrži trokute čiji su kutevi manji od 5° kao što možemo vidjeti na slici 2.3. Može nam se činiti kako nikako ne možemo dobiti rezultat koji će biti dobro oblikovan. No, za razliku od trokuta dobivenih Delaunayjevom triangulacijom, trokuti dobiveni nekim drugim postupkom mogu imati vrh koji nije vrh zadanog kvadrata ili vrh komponente. Dozvoljeno je dodavati točke kako bismo dobili dobro oblikovanu triangulaciju. Takve dodatne točke nazivamo **Steinerove točke** (eng. *Steiner points*). U prethodno spomenutom primjeru, ako dodamo Steinerovu točku na svaku poziciju s cjelobrojnim koordinatama unutar zadanog kvadrata, dobit ćemo triangulaciju koji se sastoji samo od trokuta s kutevima od 45° i 90°, tj. dobit ćemo dobro oblikovnu mrežu koji možemo vidjeti na lijevom dijelu slike 2.4. Nažalost, ovakva triangulacija ne zadovoljava četvrto svojstvo, odnosno uniformna je (trokuti iste veličine nalaze se svugdje u mreži, ne postoji profinjenija mreža blizu komponente, a grublja udaljavanjem od te iste komponente), a mi želimo neuniformnu triangulaciju. Neuniformnu triangulaciju možemo dobiti ukoliko postupno povećavamo veličinu trokuta kako se udaljavamo od komponente, a rezultat možemo vidjeti na desnom dijelu slike 2.4. Ovim postupkom dobivamo značajno manji broj trokuta: na slici 2.4 uniformna mreža ima 512 trokuta, a neuniformna samo 52.



Source: Preuzeto iz cjeline 14.3 iz [2]

Slika 2.3: Mesh dobiven Delaunayjevom triangulacijom



Izvor: Preuzeto iz cjeline 14.3 iz [2]

Slika 2.4: Uniformna i neuniformna triangulacija

2.2 Primjena quadtree-a u triangulacijama

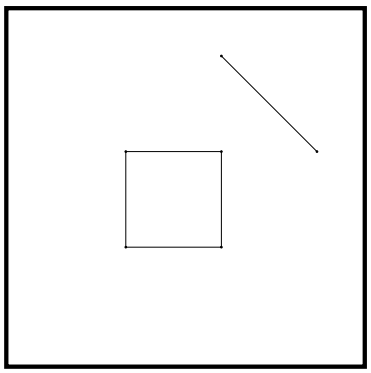
Kako bismo generirali triangulaciju koji zadovoljava sva četiri svojstva, koristit ćemo quadtree koji smo već implementirali. Prvi korak u dobivanju triangulacije bit će konstrukcija quadtree-a na skupu poligona. Naša dosadašnja implementacija gradila je quadtree na skupu točaka, a ne na skupu poligona, stoga moramo prilagoditi algoritam za izgradnju quadtree-a tako da podržava skup poligona. Poligone ćemo reprezentirati bridovima, odnosno unijom bridova dobit ćemo sve poligone koji predstavljaju komponente tiskane pločice. Uvodimo novu strukturu `Edge` koja će reprezentirati brid poligona. `Edge` se sastoji od dvije varijable tipa `Point`, a to su dvije točke koje označavaju početnu točku brida (`startPoint`), odnosno završnu točku (`endPoint`) brida. Orijentacija brida važna nam

je samo kod kreiranja brida iz poligona, ali to će biti opisano naknadno. Inače, njegova orijentacija nije bitna, odnosno zamjenom varijabli `startPoint` i `endPoint` dobivamo identičan brid. Struktura se sastoji od jednog konstruktora koji inicijalizira vrijednosti varijabli `startpoint` i `endPoint` te nekoliko metoda koje opisujemo u nastavku.

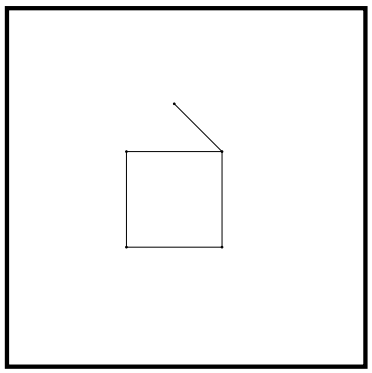
```
struct Edge
{
    Point startPoint;
    Point endPoint;

    Edge(Point, Point);
    virtual ~Edge();

    bool hasEqualVertex(Range *);
    bool intersectsSquare(Range *);
    bool isInInterior(Range *);
};
```

Metoda `hasEqualVertex` provjerava ima li brid barem jedan isti vrh kao i kvadrat tipa `Range` kojeg dobivamo u argumentu. Ukoliko postoji barem jedna zajednička točka, metoda vraća `true`, a u suprotnom vraća `false`. Implementaciju metode izostavljamo, a sastoji se od provjeravanja vrijednosti koordinata bridova s koordinatama točka vrhova kvadrata.

Pomoću metode `intersectsSquare` utvrđujemo postoji li presjek između brida i kvadrata tipa `Range` zadanog putem argumenta metode. Smatramo da presjek postoji ukoliko brid dodiruje zatvarač zadanog kvadrata. Na početku, nalazimo minimalnu i maksimalnu vrijednost x-koordinate brida te ih redom pospremamo u varijable `xMin` i `xMax`. Isti postupak radimo za y-koordinatu, odnosno `yMin` sadrži najmanju vrijednost y-koordinate brida, a `yMax` najveću vrijednost. Ukoliko je `xMax` strogo manja od varijable `square->x1` (koja predstavlja najmanju vrijednost x-koordinate kvadrata), tada se brid nalazi s lijeve strane kvadrata te ga ne dodiruje, stoga vraćamo `false`. Isto tako, ako je `xMin` strogo veća od desne granice kvadrata po x-koordinati, tada se brid nalazi s desne strane zadanog kvadrata te ponovo vraćamo `false`. Analogno provjeravamo nalazi li se brid ispod, odnosno iznad kvadrata te i u tim slučajevima vraćamo `false`. Ovdje se može dogoditi slučaj da brid ima jednu istu koordinatu točke kao i kvadrat, ali ipak ne dodiruje kvadrat, a primjer takvog slučaja možemo vidjeti na slici 2.5. Zasad bi napisani algoritam vratio vrijednost `true`, odnosno vratio bi da presjek postoji. Zbog toga, prvo provjeravamo jesu li x-koordinate bridova jednake. Ukoliko jesu, gledamo nalaze li se obje y-koordinate brida iznad ili ispod kvadrata. Ukoliko nisu jednake, računamo jednadžbu pravca kroz dvije točke koristeći



Slika 2.5: Primjer nepostojanja presjeka brida i kvadrata



Slika 2.6: Primjer brida koji se ne nalazi u unutrašnjosti kvadrata

točke brida. Jednadžba pravca koja sadrži brid glasi $y = a \cdot x + b$, pri čemu a i b izračunamo pomoću unaprijed poznatih formula iz matematike. Zatim računamo y-vrijednosti za x-koordinate vrhova zadanog kvadrata. Ukoliko su obje dobivene vrijednosti strogo manje od `square->y1`, tada se kvadrat nalazi sasvim iznad brida te presjek ne postoji. Također, ukoliko su obje dobivene vrijednosti strogo veće od `square->y2`, tada se kvadrat nalazi ispod brida (slika 2.5) te presjek ne postoji. U svakom drugom slučaju koji nije naveden, presjek postoji te `intersectsSquare` vraća `true`.

```
bool Edge::intersectsSquare(Range *square)
{
    int xMin, xMax;
    int yMin, yMax;
    if (this->startPoint.x >= this->endPoint.x)
    {
        xMax = this->startPoint.x;
        xMin = this->endPoint.x;
    }
    else
    {
        xMax = this->endPoint.x;
        xMin = this->startPoint.x;
    }

    if (this->startPoint.y >= this->endPoint.y)
    {
        yMax = this->startPoint.y;
```

```
        yMin = this->endPoint.y;
    }
    else
    {
        yMax = this->endPoint.y;
        yMin = this->startPoint.y;
    }

    if (square->x1 > xMax || square->x2 < xMin)
    {
        return false;
    }

    if (square->y1 > yMax || square->y2 < yMin)
    {
        return false;
    }

    double yRectLeft;
    double yRectRight;

    if (this->endPoint.x == this->startPoint.x)
    {
        yRectLeft = yMin;
        yRectRight = yMax;
    }
    else
    {
        double a = (this->endPoint.y - this->startPoint.y) /
        ↪  (this->endPoint.x - this->startPoint.x);
        double b = a * this->startPoint.x * (-1) +
        ↪  this->startPoint.y;

        yRectLeft = a * square->x1 + b;
        yRectRight = a * square->x2 + b;
    }

    if (square->y1 > yRectLeft && square->y1 > yRectRight)
    {
```

```
        return false;
    }

    if (square->y2 < yRectLeft && square->y2 < yRectRight)
    {
        return false;
    }

    return true;
}
```

Zadnja metoda strukture `Edge` koju opisujemo je metoda `isInInterior` koja provjerava nalazi li se brid u unutrašnjosti kvadrata tipa `Range` zadanog argumentom. Algoritam je sličan upravo opisanom za metodu `intersectsSquare`, ali uvjeti su drugačiji. Na početku, ponovo nalazimo `xMin`, `xMax`, `yMin` te `yMax` brida. Ukoliko je `xMax` manji ili jednak od `square->x1`, tada se brid nalazi s lijeve strane kvadrata te ili uopće ne dodiruje kvadrat, ili ga dodiruje samo na rubu. Nas zanima samo je li brid sadržan u unutrašnjosti kvadrata, stoga u tom slučaju vraćamo `false`. Isto tako, ako je `xMin` veća ili jednaka `square->x2`, tada se brid nalazi s desne strane kvadrata te ga možda dodiruje samo u rubu, stoga ponovo vraćamo `false`. Analogno radimo za varijable `yMin` i `yMax`. Isto kao i u prethodnoj metodi, može se dogoditi slučaj koji zasad nije pokriven u algoritmu te bi vratio netočnu vrijednost. Na primjer, taj slučaj se dogodi kada brid dodiruje samo jedan vrh kvadrata i ništa drugo (slika 2.6). Ponovo gledamo jesu li x-koordinate brida jednake te, u slučaju da jesu, gledamo je li brid iznad ili ispod kvadrata. Ukoliko su x-koordinate različite, određujemo jednadžbu pravca kroz dvije točke te u nju uvrštavamo x-koordinate kvadrata. Ukoliko su dobivene vrijednosti manje ili jednake od `square->y1`, brid dodiruje kvadrat u nekom od donjih vrhova kvadrata, stoga se ne nalazi u unutrašnjosti. Također, ako su dobivene vrijednosti veće ili jednake od `square->y2`, tada brid dodiruje jedan od gornjih vrhova kvadrata te nije sadržan u unutrašnjosti, stoga `isInInterior` vraća `false`. U svakom drugom slučaju, metoda vraća `true`.

```
bool Edge::isInInterior(Range *square)
{
    int xMin, xMax;
    int yMin, yMax;
    if (this->startPoint.x >= this->endPoint.x)
    {
        xMax = this->startPoint.x;
        xMin = this->endPoint.x;
    }
```

```
else
{
    xMax = this->endPoint.x;
    xMin = this->startPoint.x;
}

if (this->startPoint.y >= this->endPoint.y)
{
    yMax = this->startPoint.y;
    yMin = this->endPoint.y;
}
else
{
    yMax = this->endPoint.y;
    yMin = this->startPoint.y;
}

if (xMax <= square->x1 || xMin >= square->x2)
{
    return false;
}

if (yMax <= square->y1 || yMin >= square->y2)
{
    return false;
}

double yRectLeft;
double yRectRight;

if (this->endPoint.x == this->startPoint.x)
{
    yRectLeft = yMin;
    yRectRight = yMax;
}
else
{
    double a = (this->endPoint.y - this->startPoint.y) /
    ↪  (this->endPoint.x - this->startPoint.x);
```

```
        double b = a * this->startPoint.x * (-1) +
        ↪  this->startPoint.y;

        yRectLeft = a * square->x1 + b;
        yRectRight = a * square->x2 + b;
    }

    if (square->y1 >= yRectLeft && square->y1 >= yRectRight)
    {
        return false;
    }

    if (square->y2 <= yRectLeft && square->y2 <= yRectRight)
    {
        return false;
    }

    return true;
}
```

Opisali smo strukturu koja predstavlja bridove, a sada ćemo objasniti kako dobivamo poligone te kako iz njih kreiramo bridove.

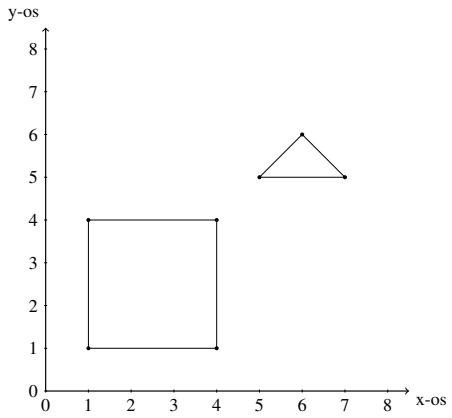
Informacije o poligonima i samim dimenzijama kvadrata dobivamo iz datoteke (primjer 2.1). U prvom redu datoteke zadana je dimenzija kvadrata, te pretpostavljamo da je taj broj oblika 2^k, gdje je k pozitivni cijeli broj. Nadalje, čitamo vrhove poligona koji su zapisani u obliku (x, y). Vrhovi su popisani na način da opisuju poligon u smjeru suprotnom od kazaljke na satu. Kada smo došli do kraja poligona, čitamo znak # koji označava početak drugog poligona, odnosno označava da su u datoteci nadalje zapisani vrhovi drugog poligona.

Gledajući primjer 2.1, u prvom redu čitamo dimenzije kvadrata ($[0, 8] \times [0, 8]$). Zatim čitamo vrhove prvog poligona (koji je zapravo kvadrat), dolazimo do simbola # te čitamo vrhove drugog poligona (poligon je trokut). Navedene poligone možemo vidjeti na slici 2.7.

Kako bismo iz navedene datoteke dobili bridove poligona, koristimo pomoćnu metodu pod nazivom `prepareEdges`. Ta metoda kao argument prima datoteku u upravo opisanom obliku, te referencu na cijeli broj u koji bude pohranjena gornja/desna granica kvadrata, a vraća skup bridova svih zadanih poligona. Na početku, definiramo skup pod nazivom `edges` koji će sadržavati bridove poligona. Prolazimo datotekom i čitamo retke dok ne dođemo do kraja datoteke. Ukoliko redak ne sadrži simbol #, znači da čitamo dimenziju kvadrata ili vrh poligona. U slučaju da ne postoji znak zareza u retku, tada se

Listing 2.1: Datoteka s poligonima

```
8
(1,1)
(4,1)
(4,4)
(1,4)
#
(5,5)
(7,5)
(6,6)
```



Slika 2.7: Poligoni

u retku nalazi dimenzija kvadrata čiju vrijednost pohranimo u varijablu `boundary` dobivenu u argumentu metode. Također, naznačimo da čitamo novi poligon, odnosno varijablu `isNewPolygon` postavimo na `true` te čitamo idući redak datoteke. U slučaju da se u retku nalazi vrh poligona, tada čitamo x i y koordinate vrha te ih redom pospremamo u varijable `firstCoordinate` i `secondCoordinate`. Od tih koordinata kreiramo točku `point` tipa `Point`. Ukoliko je upravo kreirana točka prvi pročitani vrh poligona, tada pohranimo tu točku u varijablu `startPolygonPoint` (jer će nam ta točka biti potrebna za kreiranje zadnjeg brida, odnosno kada pročitamo zadnji vrh poligona, moramo ga spojiti s početnom točkom tog istog poligona), stavimo da je `point` prvi vrh brida, odnosno `startPoint = point` te označimo da smo pročitali prvi vrh poligona, stoga će idući vrh pripadati istom poligonu (`isNewPolygon = false`). Ako kreirana točka nije prvi pročitani vrh poligona, tada kreiramo brid `edge` koji ima jedan vrh u upravo kreiranoj točki `point`, a drugi vrh je točka koja je kreirana u prethodnom retku i pohranjena u varijablu `startPoint`. No-

vokreirani brid dodamo u skup svih bridova `edges` te ažuriramo zadnju pročitanu točku, odnosno pripremimo da je vrh novog brida koji će se kreirani nakon čitanja idućeg retka jednak zadnjoj kreiranoj točki (`startPoint = point`).

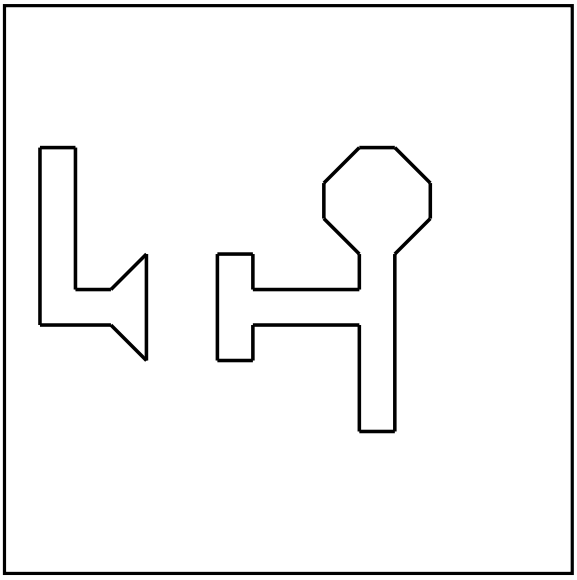
Ukoliko redak sadrži simbol #, znači da smo došli do kraja prvog poligona i iduća točka će biti vrh novog poligona. Tada kreiramo zadnji brid koji ima vrhove u prvom pročitanom vrhu poligona (`startPolygonPoint`) te zadnjoj kreiranoj točki pohranjenoj u varijabli `startPoint`. Dodamo kreirani brid u `edges` te označimo početak čitanja novog poligona (`isNewPolygon = true`).

Ako smo došli do kraja datoteke, tada nemamo više novih poligona za pročitati, no moramo kreirati i pohraniti zadnji brid poligona. U prethodnom retku pročitali smo zadnji vrh poligona te ga sad moramo spojiti s prvim pročitanom vrhom tog istog poligona. Kreirani brid dodamo u `edges` čime algoritam završava, a skup svih kreiranih bridova `edges` predstavlja povratnu vrijednost metode `prepareEdges`.

```
std::vector<Edge> prepareEdges(std::ifstream &file, int &boundary)
{
    std::string line;
    std::vector<Edge> edges;
    int firstCoordinate, secondCoordinate;
    bool isNewPolygon = false;
    Point startPolygonPoint;
    Point startPoint;

    while (std::getline(file, line))
    {
        if (line.compare("#") != 0)
        {
            std::wstring::size_type position =
            ↪  line.find_first_of(',');

            if (position == std::string::npos)
            {
                boundary = std::atoi(line.c_str());
                isNewPolygon = true;
                continue;
            }

            firstCoordinate = std::atoi(line.substr(1, position -
            ↪  1).c_str());
```

```
            secondCoordinate = std::atoi(line.substr(position + 1,
            ↪  line.length() - 1).c_str());
            Point point(firstCoordinate, secondCoordinate);

            if (isNewPolygon)
            {
                startPolygonPoint = point;
                startPoint = point;
                isNewPolygon = false;
            }
            else
            {
                Edge edge(startPoint, point);
                edges.push_back(edge);
                startPoint = point;
            }
        }
        else
        {
            Edge edge(startPoint, startPolygonPoint);
            edges.push_back(edge);
            isNewPolygon = true;
        }
    }

    if (!file)
    {
        Edge edge(startPoint, startPolygonPoint);
        edges.push_back(edge);
    }

    return edges;
}
```

Kako bi naša triangulacija bila zanimljivija, kreirali smo primjer s malo kompliciranijim komponentama čija se motivacija i dodatni primjeri mogu naći u [1]. Ovaj primjer često će biti spomenut u nastavku pa ga možemo nazvati *finalni primjer*. Čitanjem datoteke koja sadrži spomenute kompliciranije komponente i korištenjem metode `prepareEdges` dobivamo rezultat koji možemo vidjeti na slici 2.8.



Slika 2.8: Zadane komponente

Sada možemo iz zadanih komponenti dobiti sve bridove, no i dalje nismo prilagodili dosadašnju implementaciju quadtree-a kako bi ona podržavala skup bridova umjesto skup točaka. U čvoru stabla smo u varijabli `data` tipa `QuadData` pohranjivali kvadrat i točku ukoliko postoji točka koja pripada spomenutom kvadratu. Sada mijenjamo strukturu `QuadData` tako da podržava bridove. Umjesto varijable `point` tipa `Point`, sada pohranjujemo skup bridova koji sijeku kvadrat naveden u varijabli `square`. Konstruktor strukture također je prilagođen skupu bridova, dok su svi ostali dijelovi strukture ostali isti.

```
struct QuadData
{
    std::vector<Edge> edges;
    Range *square;

    QuadData(std::vector<Edge>, Range *);
    QuadData(Range *);
    virtual ~QuadData();

    Point findMiddlePoint();
};
```

Dosadašnja implementacija quadtree-a imala je konstruktor čiji je prvi argument bio skup točaka te je bio definiran na idući način: `QuadTree(std::vector<Point>, Range`

`*, QuadTree *)`. Kriterij za zaustavljanje rekurzije bio je kardinalitet skupa točaka, tj. ako je skup sadržavao strogo manje od dvije točke, rekurzija nije ulazila u novu dubinu. Naš željeni konstruktor za bridove ima oblik: `QuadTree(std::vector<Edge>, Range *, QuadTree *)`, a kriterij za zaustavljanje je također potrebno prilagoditi. Rekurzija se zaustavlja ukoliko kvadrat ne siječe niti jedan brid komponente (u argumentu tada dobijemo prazan skup bridova jer niti jedan brid ne siječe kvadrat), ili ako stranica kvadrata ima jediničnu duljinu. Podrazumijevamo da komponenta siječe kvadrat i u slučaju ako stranica kvadrata sadrži rub komponente, tj. ne gledamo samo presjek brida i unutrašnjosti kvadrata nego dozvoljavamo i dijeljenje rubova. Novi kriterij zaustavljanja garantira nam da će mreža biti neuniformna, odnosno bridovi komponenti bit će okruženi kvadratima sa stranicama jedinične duljine, a veličina kvadrata će se postupno povećavati udaljavanjem od bridova komponenti.

```
QuadTree::QuadTree(std::vector<Edge> edges, Range *square, QuadTree
↪  *parent)
{
    this->parent = parent;

    if ((square->x2 - square->x1) == 1)
    {
        this->data = new QuadData(edges, square);
        this->NE = NULL;
        this->NW = NULL;
        this->SW = NULL;
        this->SE = NULL;

        return;
    }

    if (edges.size() == 0)
    {
        this->data = new QuadData(square);
        this->NE = NULL;
        this->NW = NULL;
        this->SW = NULL;
        this->SE = NULL;

        return;
    }
```

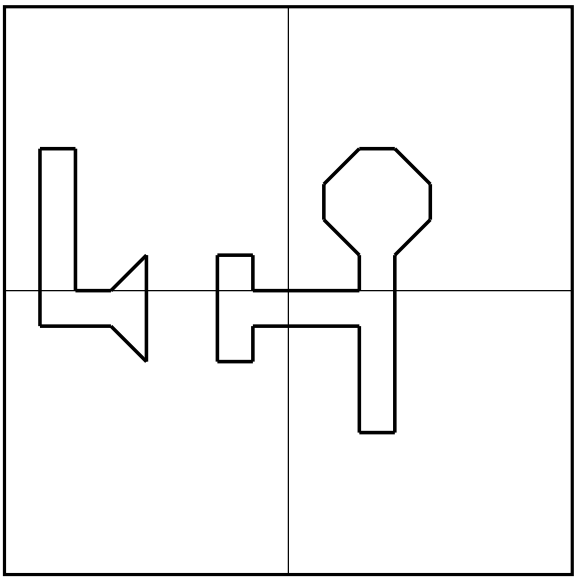
```
    this->data = new QuadData(edges, square);
    this->divideNode(edges);
}
```

Također je potrebno prilagoditi metodu `divideNode`. Ta metoda je u dosadašnjoj implementaciji dijelila zadani skup točaka na skupove točaka za svaki kvadrat pomoću metode `partitionPoints`. Sada definiramo novu metodu `partitionEdges` koja će dijeliti zadani skup bridova na skupove bridova za svaki kvadrat. Nađemo središnju točku kvadrata čvora kojeg dijelimo te pomoću nje kreiramo kvadrate za svako dijete čvora. Za svaki brid provjerimo siječe li kvadrat djeteta, te ukoliko presjek postoji, dodamo brid u skup bridova za određeno dijete. Uočimo kako u ovoj metodi jedan brid može presijecati kvadrate različite djece dok se u metodi `preparePoints` točka mogla nalaziti samo u jednom kvadratu djeteta. Razlog pripadanja jednog brida u više različitih kvadrata leži u činjenici da stranica kvadrata može sadržavati brid, stoga dva susjedna kvadrata mogu sadržavati isti brid.

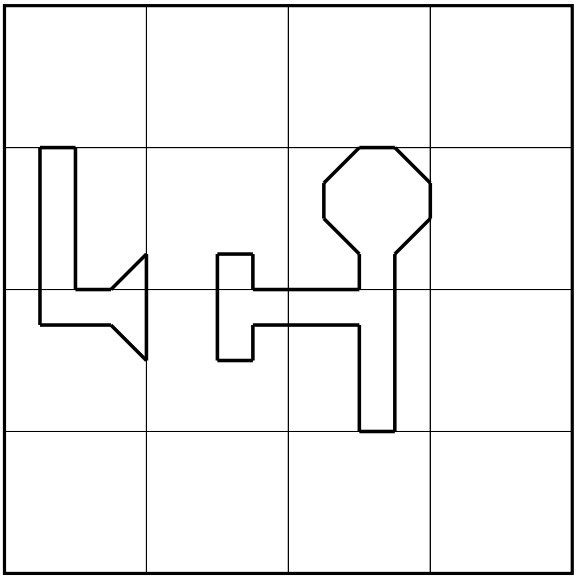
```
void partitionEdges(QuadData *data, std::vector<Edge> edges,
↪  std::vector<Edge> &edgesNE, std::vector<Edge> &edgesNW,
↪  std::vector<Edge> &edgesSW, std::vector<Edge> &edgesSE)
{
    Point middlePoint = data->findMiddlePoint();
    Range *squareNE = new Range(middlePoint.x, data->square->x2,
    ↪  middlePoint.y, data->square->y2);
    Range *squareNW = new Range(data->square->x1, middlePoint.x,
    ↪  middlePoint.y, data->square->y2);
    Range *squareSW = new Range(data->square->x1, middlePoint.x,
    ↪  data->square->y1, middlePoint.y);
    Range *squareSE = new Range(middlePoint.x, data->square->x2,
    ↪  data->square->y1, middlePoint.y);

    for (auto edge : edges)
    {
        if (edge.intersectsSquare(squareNE))
        {
            edgesNE.push_back(edge);
        }

        if (edge.intersectsSquare(squareNW))
        {
            edgesNW.push_back(edge);
```

```
        }

        if (edge.intersectsSquare(squareSW))
        {
            edgesSW.push_back(edge);
        }

        if (edge.intersectsSquare(squareSE))
        {
            edgesSE.push_back(edge);
        }
    }
}
```

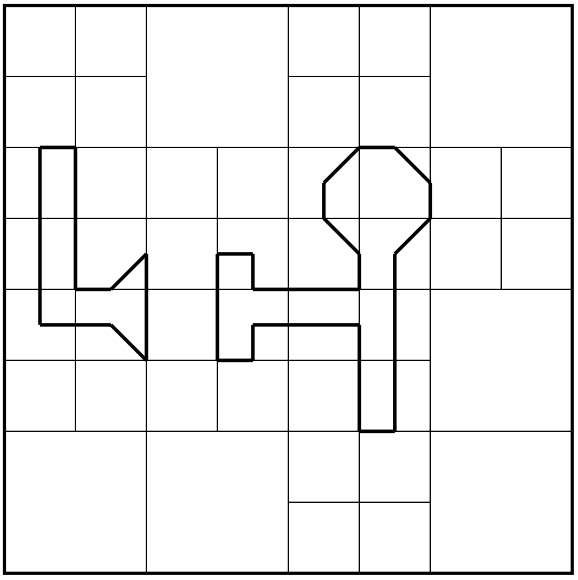
Sada je nova implementacija quadtree-a spremna za poligone. U našem *finalnom primjeru*, kvadrat ima dimenzije $[0, 32] \times [0, 32]$ te pozivom konstruktora dobivamo sljedeće stablo: početak procesa prikazuje slika 2.8 koja predstavlja komponente za koje gradimo quadtree. Na slici 2.9 možemo vidjeti prvu podjelu kvadrata, dok slike 2.10 - 2.12 predstavljaju daljnje podjele kvadrata. Slika 2.13 predstavlja konačan rezultat, odnosno quadtree kreiran na temelju zadanih poligona.



Slika 2.9: Prva podjela

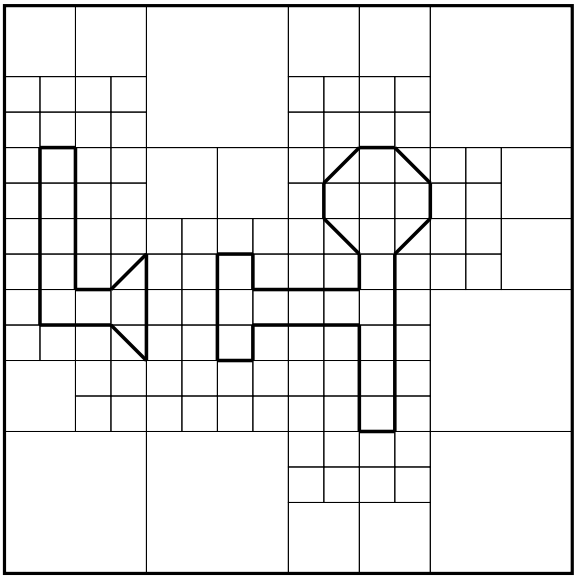


Slika 2.10: Druga podjela

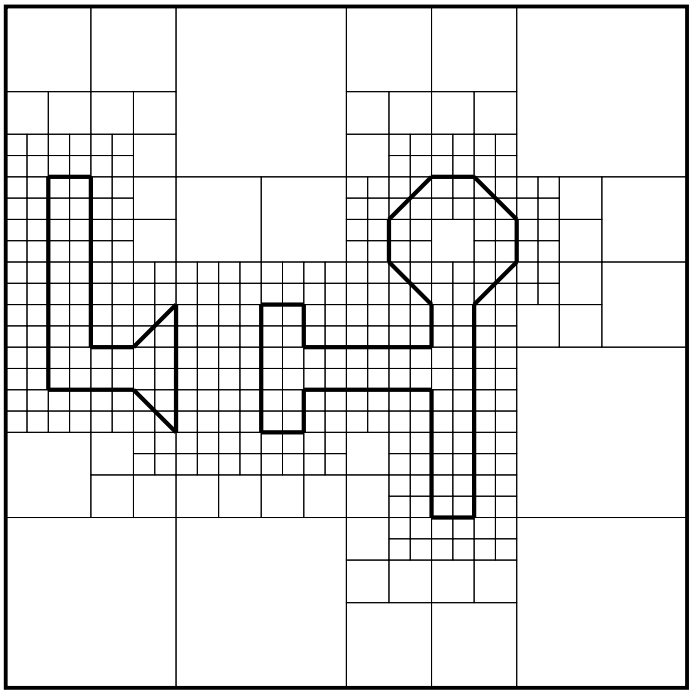


Slika 2.11: Treća podjela

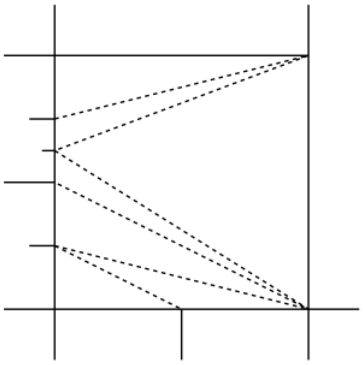
Tvrdimo da unutrašnjost svakog kvadrata u rezultantnoj quadtree subdiviziji može imati presjek s bridom komponente samo na jedan način: presjek je dijagonala kvadrata. Uistinu, kvadrati koji imaju presjek (u unutrašnjosti ili rubu) s bridovima komponente



Slika 2.12: Četvrta podjela

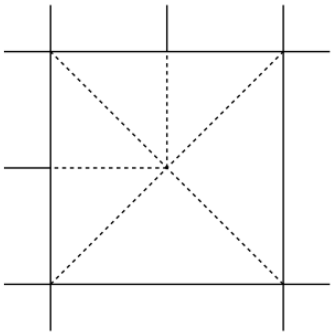


Slika 2.13: Završna podjela



Izvor: Preuzeto iz cjeline 14.3 iz [2]

Slika 2.14: Kvadrat koji uzrokuje loše oblikovanu triangulaciju



Source: Preuzeto iz cjeline 14.3 iz [2]

Slika 2.15: Povezivanje Steinerove točke s ostalim vrhovima

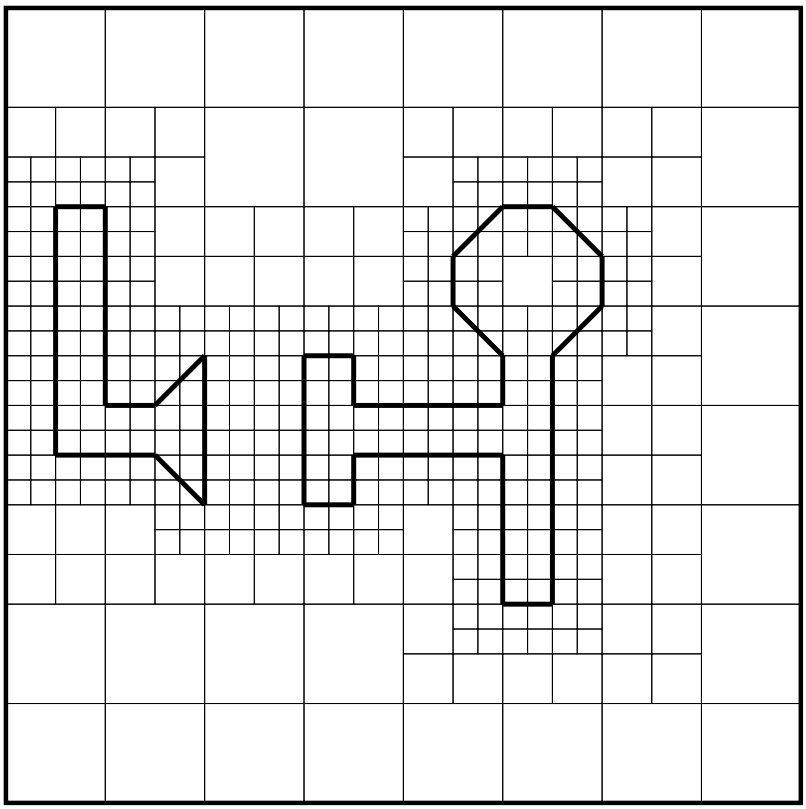
imaju stranice jedinične duljine te vrhovi komponenti imaju cjelobrojne koordinate. Zbog toga, unutrašnjost kvadrata ne može biti presječena vertikalnom ili horizontalnom linijom te je presjek s bridom koji ima kut vrijednosti 45° ili 135° upravo dijagonala. Možemo pomisliti kako trebamo dodati dijagonale onim kvadratima čija unutrašnjost nema presjek s bridom komponente kako bismo dobili validnu triangulaciju. Takva triangulacija će poštovati ulazne podatke, bit će dobro oblikovana te neuniformna. Nažalost, takav triangulacija neće biti konformna. Ovome možemo doskočiti na način da uzmemo u obzir i vrhove na stranicama kvadrata koji nastaju quadtree subdivizijom kada radimo trokute triangulacije, no time dolazimo do drugog problema: ukoliko kvadrat ima više vrhova na stranici, odnosno ako stranica sadrži više rubova drugih kvadrata, tada triangulacija više neće biti dobro oblikovana, odnosno trokuti neće imati dozvoljene kuteve kao što možemo vidjeti na slici 2.14.

Kako bismo izbjegli upravo navedene probleme u dobivanju validne triangulacije, balansirat ćemo quadtree prije negoli napravimo triangulaciju. Tada možemo na jednostavan način generirati triangulaciju koja je dobro oblikovana jer će se susjedne stranice kvadrata razlikovati u duljini najviše za faktor 2. Kvadratima koji nemaju vrhove drugih kvadrata u unutrašnjosti stranica te koji nemaju presjek niti s jednim bridom komponente rastavimo na trokute na način da kvadratu dodamo dijagonalu. Dijagonalu dodajemo i kvadratima koji imaju presjek s bridom komponente. Preostali su kvadrati koji imaju vrhove drugih kvadrata u unutrašnjosti stranice (točnije, imaju vrh točno na sredini stranice) te ne sadrže brid komponente. Njima dodajemo Steinerovu točku u sredinu kvadrata i povežemo ju sa svim ostalim vrhovima, odnosno s vrhovima kvadrata i vrhovima na sredini stranica (slika 2.15). Na ovaj način smo dobili trokute koji imaju samo kuteve od 45° i 90°.

Metoda za balansiranje stabla (`balanceQuadtree`) ostaje ista kao u prvotnoj imple-

mentaciji, samo mijenjamo jedan mali dio implementacije koji se tiče pozivanja metode `divideNode`. Metodu `divideNode` sada zovemo s argumentom koji sadrži skup bridova pa cijeli novi redak izgleda: `leaf->divideNode(leaf->getData()->edges)`, a izbacimo sve vezano uz varijable koje su uključivale tip `Point`.

Pozivanjem metode `balanceQuadtree` na izgrađenom quadtree-u iz *finalnog primjera* dobivamo balansirani quadtree koji možemo vidjeti na slici 2.16.



Slika 2.16: Balansirani quadtree s poligonima

2.3 Implementacija generiranja triangulacija

Riješili smo sve probleme koji su uzrokovali dobivanje nevalidne triangulacije, stoga smo sada spremni za implementaciju generiranja validne triangulacije. Kako triangulaciju dobivamo rastavljajući kvadrate na trokute, kreiramo novu strukturu `Triangle` koja će predstavljati trokute u triangulaciji. Struktura se sastoji od tri točke koje predstavljaju vrhove trokuta dobivenog podjelom kvadrata te nam njihov redoslijed nije bitan. `Triangle` je

jednostavna struktura koja se sastoji od destruktora i samo jednog konstruktora koji inicijalizira vrhove trokuta.

```
struct Triangle
{
    Point firstPoint;
    Point secondPoint;
    Point thirdPoint;

    Triangle(Point, Point, Point);
    virtual ~Triangle();
};
```

Konačno dolazimo do strukture pod nazivom `Mesh` koja će predstavljati generiranu triangulaciju. Struktura se sastoji od niza trokuta, odnosno niza varijabli tipa `Triangle`. Također, struktura sadrži nekoliko metoda koje će nam olakšati samo generiranje. Obje metode `addDiagonal` dodaju dva trokuta u niz `triangles`, ali se razlikuju u načinu na koji se dijagonala kreira. `addDiagonal(Edge, Range*)` kreira dijagonalu u kvadratu čija je unutrašnjost presječena nekim bridom te uzima u obzir orijentaciju brida, dok `addDiagonal(Range*)` dodaje dijagonalu uvijek na isti način. Preostale dvije metode ćemo opisati kasnije.

```
struct Mesh
{
    std::vector<Triangle *> triangles;

    void addDiagonal(Edge, Range *);
    void addDiagonal(Range *);
    void connectSteiner(Point, Range *, Neighbor);
    void addTriangle(Point, Range *, Neighbor);

    virtual ~Mesh();
};
```

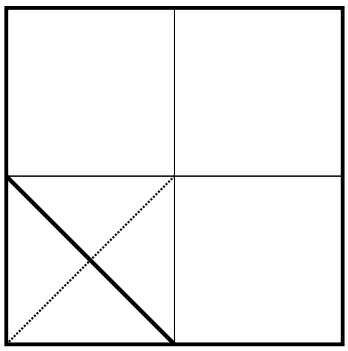
Prvo opisujemo metodu koja dodaje dijagonalu kvadratu čija je unutrašnjost presječena bridom. Metoda uzima u obzir način na koji je brid orijentiran tako da kvadrat ne bi bio pogrešno trianguliran kao na slici 2.17. Na slici, brid siječe kvadrat iz gornjeg lijevog kuta prema donjem desnom kutu (puna linija), dok je kvadrat trianguliran u obrnutom smjeru, odnosno napravljena je dijagonala čiji su vrhovi u donjem lijevom i gornjem desnom kutu (točkasta linija). Kako se to ne bi dogodilo, provjeravamo orijentaciju brida. Na početku,

napravimo četiri točke koje predstavljaju vrhove kvadrata. Tada, u ovisnosti o orijentaciji brida, kreiramo dva trokuta te ih na kraju dodajemo u niz `triangles`.

```
void Mesh::addDiagonal(Edge edge, Range *square)
{
    Point firstPoint(square->x2, square->y2);
    Point secondPoint(square->x1, square->y2);
    Point thirdPoint(square->x1, square->y1);
    Point fourthPoint(square->x2, square->y1);

    if ((edge.startPoint.y >= edge.endPoint.y && edge.startPoint.x
    ↪  >= edge.endPoint.x) || (edge.startPoint.y <= edge.endPoint.y
    ↪  && edge.startPoint.x <= edge.endPoint.x))
    {
        Triangle *firstTriangle = new Triangle(firstPoint,
        ↪  secondPoint, thirdPoint);
        Triangle *secondTriangle = new Triangle(firstPoint,
        ↪  thirdPoint, fourthPoint);
        this->triangles.push_back(firstTriangle);
        this->triangles.push_back(secondTriangle);
    }
    else
    {
        Triangle *firstTriangle = new Triangle(firstPoint,
        ↪  secondPoint, fourthPoint);
        Triangle *secondTriangle = new Triangle(secondPoint,
        ↪  thirdPoint, fourthPoint);
        this->triangles.push_back(firstTriangle);
        this->triangles.push_back(secondTriangle);
    }
}
```

Druga `addDiagonal` metoda dodaje dijagonalu kvadrata uvijek na isti način, tj. vrhovi dijagonale nalaze se u donjem lijevom i gornjem desnom kutu. Isto kao i u prethodnoj istoimenoj metodi, na početku kreiramo točke koje predstavljaju vrhove kvadrata. Zatim kreiramo dva trokuta u skladu s dijagonalom te ih dodamo u niz `triangles`.

```
void Mesh::addDiagonal(Range *square)
{
    Point firstPoint(square->x2, square->y2);
```



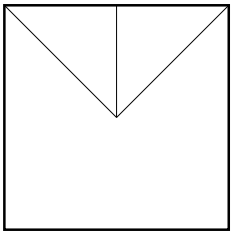
Slika 2.17: Pogrešno trianguliran kvadrat

```
    Point secondPoint(square->x1, square->y2);
    Point thirdPoint(square->x1, square->y1);
    Point fourthPoint(square->x2, square->y1);

    Triangle *firstTriangle = new Triangle(firstPoint, secondPoint,
    ↪  thirdPoint);
    Triangle *secondTriangle = new Triangle(firstPoint, thirdPoint,
    ↪  fourthPoint);
    this->triangles.push_back(firstTriangle);
    this->triangles.push_back(secondTriangle);
}
```

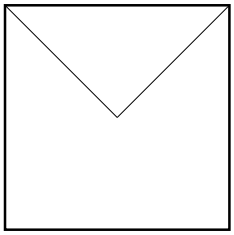
Zadnje dvije metode strukture uzimaju u obzir Steinerovu točku, a to je točka koja se nalazi u sredini kvadrata. Te dvije metode razlikuju se u broju trokuta koje kreiraju. Naime, `connectSteiner` kreira dva trokuta, dok `addTriangle` dodaje samo jedan.

Metoda `connectSteiner` prima tri argumenta. Prvi predstavlja Steinerovu točku, drugi označava kvadrat koji trianguliramo, a treći naznačava susjeda čiji rub uzimamo u obzir. Opisat ćemo slučaj kada treći argument predstavlja sjevernog susjeda. U tom slučaju gledamo gornju stranicu kvadrata, odnosno stranicu koju dijelimo sa sjevernim susjedom. Kreiramo tri točke: `leftPoint` označava gornji lijevi vrh kvadrata, `rightPoint` označava gornji desni vrh kvadrata, dok `middlePoint` predstavlja točku koja se nalazi na sredini gornje stranice kvadrata. Zatim kreiramo dva trokuta od kojih svaki ima dva vrha u točkama `steinerPoint` i `middlePoint`, dok je treći vrh različit (jedan trokut ima treći vrh u točki `leftPoint`, dok drugi trokut u `rightPoint`). Primjer navedenog možemo vidjeti na slici 2.18. Analogno radimo trokute kada treći argument predstavlja nekog drugog susjeda.

```
void Mesh::connectSteiner(Point steinerPoint, Range *square,
↪  Neighbor neighbor)
{
    if (neighbor == NORTHNEIGHBOR)
    {
        Point leftPoint(square->x1, square->y2);
        Point middlePoint(steinerPoint.x, square->y2);
        Point rightPoint(square->x2, square->y2);
        this->triangles.push_back(new Triangle(leftPoint,
        ↪  steinerPoint, middlePoint));
        this->triangles.push_back(new Triangle(steinerPoint,
        ↪  rightPoint, middlePoint));
    }

    if (neighbor == SOUTHNEIGHBOR)
    {
        Point leftPoint(square->x1, square->y1);
        Point middlePoint(steinerPoint.x, square->y1);
        Point rightPoint(square->x2, square->y1);
        this->triangles.push_back(new Triangle(leftPoint,
        ↪  steinerPoint, middlePoint));
        this->triangles.push_back(new Triangle(steinerPoint,
        ↪  rightPoint, middlePoint));
    }

    if (neighbor == EASTNEIGHBOR)
    {
        Point upperPoint(square->x2, square->y2);
        Point middlePoint(square->x2, steinerPoint.y);
        Point lowerPoint(square->x2, square->y1);
        this->triangles.push_back(new Triangle(upperPoint,
        ↪  steinerPoint, middlePoint));
        this->triangles.push_back(new Triangle(lowerPoint,
        ↪  middlePoint, steinerPoint));
    }

    if (neighbor == WESTNEIGHBOR)
    {
        Point upperPoint(square->x1, square->y2);
```



Slika 2.18: Trokuti nastali metodom `connectSteiner` u slučaju sjevernog susjeda



Slika 2.19: Trokuti nastali metodom `addTriangle` u slučaju sjevernog susjeda

```
        Point middlePoint(square->x1, steinerPoint.y);
        Point lowerPoint(square->x1, square->y1);
        this->triangles.push_back(new Triangle(upperPoint,
        ↪  middlePoint, steinerPoint));
        this->triangles.push_back(new Triangle(lowerPoint,
        ↪  steinerPoint, middlePoint));
    }
}
```

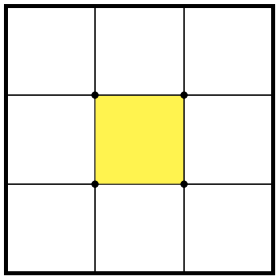
Zadnja metoda strukture `Mesh` je metoda `addTriangle`. Ona također prima tri argumenta te oni imaju iste uloge kao i u metodi `connectSteiner`: prvi argument predstavlja Steinerovu točku, drugi predstavlja kvadrat koji trianguliramo, a treći susjeda čiji rub uzimamo u obzir. Metodu ćemo opet objasniti na primjeru sjevernog susjeda. Naime, kreiramo dvije točke, odnosno `leftPoint` koja predstavlja gornji lijevi vrh kvadrata te `rightPoint` koja označava gornji desni vrh kvadrata. Zatim kreiramo trokut koji ima vrhove u upravo napravljenim točkama i Steinerovoj točki dobivenoj putem prvog argumenta, a primjer možemo vidjeti na slici 2.19. Takav trokut dodamo u niz svih trokuta `triangles`. Na sličan način kreiramo i dodajemo trokut u slučaju drugih susjeda.

```
void Mesh::addTriangle(Point middlePoint, Range *square, Neighbor
↪  neighbor)
{
    if (neighbor == NORTHNEIGHBOR)
    {
        Point leftPoint(square->x1, square->y2);
        Point rightPoint(square->x2, square->y2);
        this->triangles.push_back(new Triangle(leftPoint,
        ↪  middlePoint, rightPoint));
```

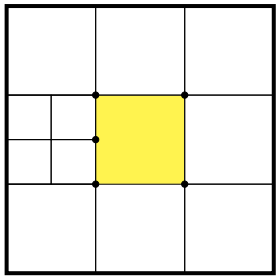
```
    }

    if (neighbor == SOUTHNEIGHBOR)
    {
        Point leftPoint(square->x1, square->y1);
        Point rightPoint(square->x2, square->y1);
        this->triangles.push_back(new Triangle(leftPoint,
        ↪  middlePoint, rightPoint));
    }

    if (neighbor == EASTNEIGHBOR)
    {
        Point upperPoint(square->x2, square->y2);
        Point lowerPoint(square->x2, square->y1);
        this->triangles.push_back(new Triangle(upperPoint,
        ↪  middlePoint, lowerPoint));
    }

    if (neighbor == WESTNEIGHBOR)
    {
        Point upperPoint(square->x1, square->y2);
        Point lowerPoint(square->x1, square->y1);
        this->triangles.push_back(new Triangle(upperPoint,
        ↪  lowerPoint, middlePoint));
    }
}
```

Završili smo s implementacijom strukture `Mesh` te nam je preostalo implementirati algoritam za generiranje triangulacija. Kako bismo to ostvarili, dodajemo dvije metode u već implementiranu klasu `QuadTree`.

```
class QuadTree
{
    ...
public:
    ...
    Mesh generateMesh();
    bool hasOnlyVertices(QuadTree *);
}
```



Slika 2.20: Kvadrat obojan žutom bojom sadrži samo vrhove na svojim stranicama



Slika 2.21: Kvadrat obojan žutom bojom sadrži i susjedove vrhove na svojim stranicama

Metoda `hasOnlyVertices` provjerava ima li čvor u stablu kvadrat koji sadrži samo vrhove na svojim stranicama. Preciznije, metoda vraća `true` samo ako susjedi zadanog čvora nemaju djecu (slika 2.20). U slučaju da susjedi imaju djecu, tada susjed sadrži manje kvadrate, stoga zadani kvadrat sadrži vrh manjeg kvadrata na svojoj stranici. Na slici 2.21 možemo vidjeti zadani kvadrat obojan žutom bojom. On ima zapadnog susjeda koji ima djecu, odnosno na stranici koja povezuje gornji lijevi i donji lijevi vrh nalazi se vrh kvadrata istočne djece zapadnog susjeda. U ovom slučaju, metoda vraća `false`. Implementacijske detalje metode izostavljamo te se posvećujemo algoritmu za generiranje triangulacija, odnosno metodi `generateMesh`.

Metoda `generateMesh` prvo definira triangulaciju (koji se sastoji od prazne liste trokuta) te iz postojećeg stabla (stabla na kojem je pozvana metoda `generateMesh`) kreira balansirano stablo. Zatim u listu `leaves` pohranimo sve listove stabla. Dokle god je lista `leaves` neprazna, radimo sljedeći postupak: uzmemo list `leaf` na početku liste te ga izbacimo iz `leaves`. Inicijaliziramo varijablu `addedDiagonal` na `false` kako bismo naznačili da zasad kvadratu lista `leaf` nismo dodali dijagonalu, tj. da još nismo triangulirali kvadrat. Za svaki brid koji siječe kvadrat lista `leaf` provjerimo da li brid siječe dijagonalno kvadrat, odnosno da li je brid sadržan u unutrašnjosti kvadrata. Ukoliko je brid stvarno sadržan u unutrašnjosti, brid predstavlja dijagonalu kvadrata stoga triangluliramo kvadrat uzimajući u obzir orijentaciju brida. To radimo pomoću već opisane metode `addDiagonal` s dva parametra. Time smo triangulirali kvadrat lista `leaf`, stoga prelazimo na idući list liste `leaves`. Ukoliko niti jedan brid lista `leaf` ne siječe unutrašnjost kvadrata, provjeramo ima li kvadrat samo vrhove na svojim stranicama. U slučaju da ima, dodajemo dijagonalu kvadratu, odnosno trianguliramo kvadrat na dva trokuta pomoću metode `addDiagonal` s jednim parametrom. Ako kvadrat ima više od četiri vrha na svojim stranicama, tada definiramo varijablu `steinerPoint` koja predstavlja Steinerovu točku te u varijablu `square` pohranjujemo kvadrat lista `leaf`. Nalazimo sve susjede čvora (lista `leaf`) te se pitamo: ako određeni susjed ima djecu, odnosno ako se na stranici kvadrata koju kvadrat dijeli

sa susjedom nalazi više od dva vrha, tada kreiramo trokute na način da ih povežemo sa Steinerovom točkom i točkom na sredini dijeljene stranice kvadrata. Ukoliko se na dijeljenoj stranici nalaze samo dva vrha, tada kreiramo samo jedan trokut pomoću metode `addTriangle`. Na kraju, kada je lista `leaves` prazna, vratimo varijablu `mesh`, odnosno sve trokute koje smo kreirali iz kvadrata listova stabla.

```
Mesh QuadTree::generateMesh()
{
    Mesh mesh;
    this->balanceQuadtree();

    std::list<QuadTree *> leaves = this->getAllLeaves();

    while (leaves.size())
    {
        QuadTree *leaf = leaves.front();
        leaves.pop_front();
        bool addedDiagonal = false;

        for (auto edge : leaf->getData()->edges)
        {
            if (edge.isInInterior(leaf->getData()->square))
            {
                mesh.addDiagonal(edge, leaf->getData()->square);
                addedDiagonal = true;
                break;
            }
        }

        if (!addedDiagonal)
        {
            if (this->hasOnlyVertices(leaf))
            {
                mesh.addDiagonal(leaf->getData()->square);
            }
            else
            {
                QuadTree *northNeighbor =
                ↪  this->findNorthNeighbor(leaf);
```

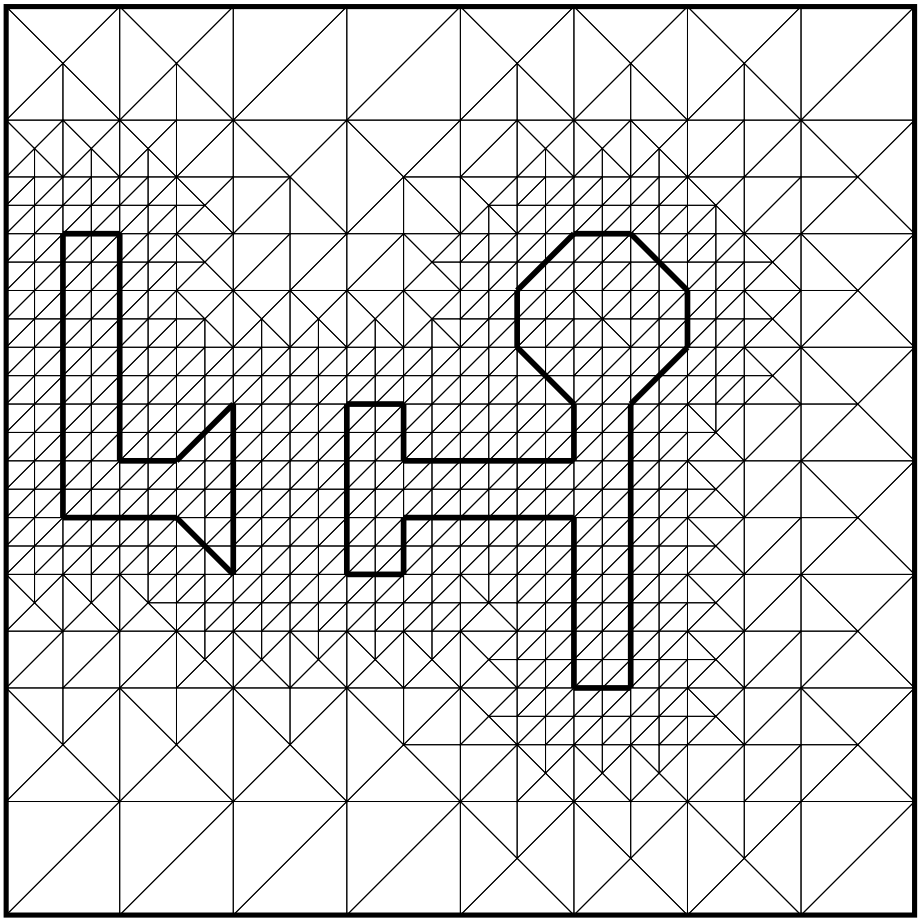
```
QuadTree *southNeighbor =
↪  this->findSouthNeighbor(leaf);
QuadTree *eastNeighbor =
↪  this->findEastNeighbor(leaf);
QuadTree *westNeighbor =
↪  this->findWestNeighbor(leaf);

Point steinerPoint =
↪  leaf->getData()->findMiddlePoint();
Range *square = leaf->getData()->square;

if (northNeighbor && northNeighbor->getNE())
{
    mesh.connectSteiner(steinerPoint, square,
    ↪  NORTHNEIGHBOR);
}
else
{
    mesh.addTriangle(steinerPoint, square,
    ↪  NORTHNEIGHBOR);
}

if (southNeighbor && southNeighbor->getNE())
{
    mesh.connectSteiner(steinerPoint, square,
    ↪  SOUTHNEIGHBOR);
}
else
{
    mesh.addTriangle(steinerPoint, square,
    ↪  SOUTHNEIGHBOR);
}

if (eastNeighbor && eastNeighbor->getNE())
{
    mesh.connectSteiner(steinerPoint, square,
    ↪  EASTNEIGHBOR);
}
else
```

```
                {
                    mesh.addTriangle(steinerPoint, square,
                    ↪  EASTNEIGHBOR);
                }

                if (westNeighbor && westNeighbor->getNE())
                {
                    mesh.connectSteiner(steinerPoint, square,
                    ↪  WESTNEIGHBOR);
                }
                else
                {
                    mesh.addTriangle(steinerPoint, square,
                    ↪  WESTNEIGHBOR);
                }
            }
        }
    }

    return mesh;
}
```

Pozivanjem metode `generateMesh` na dobivenom stablu iz našeg *finalnog primjera*, dobivamo triangulaciju koji možemo vidjeti na slici 2.22.

Ovime smo dovršili proces generiranja triangulacija. U našem primjeru, triangulaciju smo generirali na komponentama koje su bile zadane nizom bridova. Naravno, komponente mogu biti zadane i na drugačiji način te se tada implementacija generiranja prilagođava. Također, kreirali smo triangulaciju na dvodimenzionalnim komponentama, no postoje i algoritmi za generiranje na trodimenzionalnim.

Generiranje triangulacija, ili mreža sa različitim geometrijskim oblicima, koristi se u raznim područjima stoga postoje specifični zahtjevi u samom procesu generiranja. Možemo razlikovati strukturirane i nestrukturirane mreže. Strukturirane obično izgledaju poput deformiranih mreža, odnosno oblici koji čine jednu mrežu mogu biti različiti, dok se nestrukturirane sastoje od trokuta ili četverokuta (*finalni primjer* sastojao se od nestrukturirane mreže). Također, neke mreže ne moraju zadovoljavati sva četiri svojstva koja smo naveli na početku poglavlja. Ovisno o problemu koji se rješava, zahtijevaju se određena svojstva. Isto tako, kutevi trokuta mogu biti različite veličine. Postoje problemi u kojima su dozvoljene vrijednosti kuteva trokuta samo između 45° i 90° (kao u našem primjeru), ali ima i problema gdje mogu imati drugačije veličine, ili samo postoji zahtjev da im veličina ne smije biti veća od 90°.



Slika 2.22: Generirana triangulacija

Proučavaju se i triangulacije gdje se želi minimizirati broj trokuta, odnosno pokušava se kontrolirati gustoća triangulacije. Glavni zahtjev na dobivenu triangulaciju jest svojstvo da je mreža gušća na interesantnim područjima, a rijeđa na mjestima koja nisu važna za proučavanje. Zbog postojanja takvog zahtjeva, nastao je algoritam u kojem korisnik definira funkciju koja odlučuje je li mreža dovoljno profinjena. Također, trokuti u spomenutom algoritmu moraju imati kuteve između 30° i 120°.

Postoje još brojne primjene mreža, odnosno triangulacija, i razni algoritmi za njihovo generiranje, no mi smo ovdje opisali jednu: raspodjela komponenti i njihovo povezivanje na tiskanoj pločici. U tom rješavanju, baza nam je bila quadtree koji smo putem balansirali. Sve strukture koje smo koristili, klase raznih stabala, implementacije algoritama te pomoćne metode, mogu se pronaći u repozitoriju na linku: `https://github.com/koines/Geometric-data`.

Bibliografija

[1] Kevin Buchin, *Quadtrees*, https://www.win.tue.nl/~kbuchin/teaching/2IMA15/slides/11-quadtrees.pdf, Pristupljeno: 5-30-2022.

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld i Mark Overmars, *Computational Geometry*, Springer, 2008.

[3] Anthony D'Angelo, *A Brief Introduction to Quadtrees and Their Applications*, http://people.scs.carleton.ca/~maheshwa/courses/5703COMP/16Fall/quadtrees-paper.pdf, Pristupljeno: 5-29-2022.

[4] Saša Singer, *Uvod u složenost*, http://degiorgi.math.hr/oaa/materijali/scans/pog_1.pdf, Pristupljeno: 4-10-2022.

Sažetak

Podatke koji sadrže informacije o geometriji, poput koordinata točka, dužina, poligona i slično, možemo u računalu pohraniti na više načina. U ovom radu proučavali smo strukture podataka koje nam, uz kompaktno pohranjivanje odgovarajućih geometrijskih objekata, omogućavaju efikasno izvođenje raznih upita nad pohranjenim skupom. Jedna od takvih struktura je kd-stablo, odnosno binarno stablo čiji listovi reprezentiraju k-dimenzionalne točke. To je struktura čiji unutarnji čvor dijeli zadani skup točaka na dva podskupa približno jednake veličine te spomenute podskupove pohranjuje u djecu čvora sve dok skup ne sadrži samo jednu točku. Proučavali smo pohranjivanje 2-dimenzionalnih točaka te pretraživanje kd-stabla, odnosno kako možemo za zadani 2-dimenzionalni interval efikasno pronaći sve točke koje pripadaju spomenutom intervalu.

Druga struktura koju smo istraživali u radu je quadtree. To je struktura slična kd-stablu, no svaki njen unutarnji čvor ima točno četiri djeteta. Također, svaki čvor predstavlja kvadrat, te su kvadrati djece zapravo particionirani roditeljski kvadrat na četiri dijela, stoga quadtree ima veliku primjenu u pohranjivanju geometrijskih podataka. Skup početnih geometrijskih podataka (točaka, bridova, ...) može biti nepovoljan, odnosno dva susjedna kvadrata mogu imati značajne razlike u veličini stranice kvadrata. To neželjeno svojstvo riješili smo balansiranjem quadtree-a. U balansiranom quadtree-u, duljine dužina stranica susjednih kvadrata razlikuju se najviše za faktor 2, odnosno svaka dva susjedna čvora razlikuju se najviše za jedan u dubini.

Balansirani quadtree koristili smo u generiranju triangulacija, odnosno podjeli geometrijskog prostora na trokute. Kreirali smo triangulaciju na dvodimenzionalnim poligonima čija je mreža trokuta gušća oko poligona, a rijeđa udaljavanjem od njih. Triangulaciju smo generirali podjelom kvadrata balansiranog quadtree-a na trokute uzevši u obzir blizinu zadanih poligona.

Svu implementaciju spomenutih struktura podataka i pripadnih algoritama, napisali smo u programskom jeziku C++.

Summary

Data which contains information about geometry, such as coordinates of points, line segments, polygons and others, can be stored in a computer in several ways. In this thesis, we studied data structures that, along with the compact storage of appropriate geometric objects, enable us to efficiently perform various queries over the stored set. One of the structures with mentioned properties is kd-tree which is a binary tree which contains leaves that represent k-dimensional points. It is a structure whose internal node divides a given set of points into two subsets of approximately equal sizes. Those subsets are stored as left and right child of the node which performs the division. Set of points is divided as long as it contains more than one point. We studied storing 2-dimensional points and kd-tree search, that is, how can we efficiently find all points that belong to the given 2-dimensional interval.

Second structure that we studied was quadtree. Quadtree is a structure similar to a kd-tree, but each of its internal nodes has exactly four children. Also, each node represents a square and the children's squares are actually partitioned parent square in four parts, so the quadtree has great application in storing geometric data. A set of initial geometric data (points, edges, ...) can be unfavorable. For example, two adjacent squares can have significant differences in their size. We solved this unwanted feature by balancing the quadtree. In a balanced quadtree, the lengths of the line segment representing one side of the adjacent squares differ at the most by factor 2, that is, every two adjacent nodes differ by at most one in depth.

We used balanced quadtree to compute triangular mesh, that is, we divided geometric space into triangles. We performed computation on two-dimensional polygons and the constructed mesh was fine near the edges of the polygons and coarse further away from them. Mesh was generated by square triangulation of the nodes in the balanced quadtree.

Implementation of the mentioned data structures and algorithms was done using C++ programming language.

Životopis

Rođena sam 19.05.1996. u Zaboku gdje sam pohađala Osnovnu školu Ksavera Šandora Gjalskog. U istom gradu polazila sam Gimnaziju Antuna Gustava Matoša, prirodoslovno-matematički smjer nakon čijeg završetka upisujem Preddiplomski sveučilišni studij na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu. Nakon završetka, 2019. godine upisujem Diplomski sveučilišni studij Računarstva i matematike na istom fakultetu.

Tijekom diplomskog studija sudjelovala sam na ljetnoj školi za programski jezik Go pod nazivom *Axilis 2021 Golang Course* te ljetnoj školi za razvoj web aplikacija koristeći Spring pod organizacijom tvrtke Agency04. Također, završila sam smjer *JS & Angular* polazeći Infinum Academy nakon čega se zapošljavam u tvrtci Porsche Digital Croatia gdje radim i danas.