Tretja domača naloga pri algoritmih

Lucija Fekonja

8. maj 2024

Naloga 1: Konveksna ovojnica

Konveksna ovojnica unije točk konveksnih ovojnic

Znan algoritem za iskanje konveksne ovojnice unije točk konveksnih ovojnic S_1 in S_2, ki deluje v linearnem času, uporablja Rotating Calipers. Algoritem sprejme točke na konveksnih ovojnicah P in Q urejene v nasprotni smeri urinega kazalca.

ZDRUŽEVANJE KONVEKSNIH OVOJNIC (P, Q)

$mostovi = [\,]$
$p_0 \leftarrow$ Točka z najvišjo y koordinato v P.
$q_0 \leftarrow$ Točka z najvišjo y koordinato v Q.
$v = [-1, 0]$ (Predstavlja vodoravnico).
$rotacija = 0$
$i, j = 0, 0$
while $rotacija < 2\pi$:
    $\theta_p \leftarrow$ Kot med v in $\overline{p_i p_{i+1}}$
    $\theta_q \leftarrow$ Kot med v in $\overline{q_j q_{j+1}}$
    $\theta = \min(\theta_p, \theta_q)$
    Zarotiraj v za θ v nasprotni smeri urinega kazalca.
    $rotacija = rotacija + \theta$
    if $\theta_p < \theta_q$
        $i++$
    elif $\theta_p > \theta_q$
        $j++$
    else (* $\theta_p = \theta_q$ *)
        $i++$
        $j++$
    end if

if $\theta_p \neq \theta_q$

$l \leftarrow$ Premica skozi p_i in q_j

Če točke p_{i-1}, p_{i+1}, q_{i-1} in q_{i+1} ležijo na isti strani premice l,

dodaj par (p_i, q_j) v *mostovi*.

else (* $\theta_p = \theta_q$ imamo vzporedne robove *)

Isto stvar preveri za premice skozi (p_i, q_j), (p_{i-1}, q_j), (p_i, q_{j-1}) in (p_{i-1}, q_{j-1}.)

end if

end while

ovojnica = [Točka z višjo y-koordinato med p_0 in q_0]

if Prva točka v ovojnici je p_0

Dodajamo točke iz P v smeri urinega kazalca, dokler ne pridemo to mosta.

Dodajamo točke iz Q v smeri urinega kazalca, dokler ne pridemo do mosta.

Dodajamo točke iz P v smeri urinega kazalca, dokler ne pridemo do začetne točke.

else

Dodajamo točke iz Q v smeri urinega kazalca, dokler ne pridemo to mosta.

Dodajamo točke iz P v smeri urinega kazalca, dokler ne pridemo do mosta.

Dodajamo točke iz Q v smeri urinega kazalca, dokler ne pridemo do začetne točke.

Preverjanje ali se dve konveksni ovojnici sekata

Algoritem, ki preveri ali se dva konveksna lika sekata temelji na Separating Axis Theorem, ki pravi, da se dva lika ne sekata, če lahko med njima začrtamo premico, ki se nobenega ne dotika. Algoritem deluje na sledeč način:

CHECK INTERSECTION (P, Q)

robovi ← Vsi robovi P in Q

normale ← Normale na robove

for n **in** *normale*

Izračunaj projekcijo vseh vozlišč P in Q na n.

$p_{\min}, p_{\max} \leftarrow$ Prva in zadnja projekcija točk it P na n.

$q_{\min}, q_{\max} \leftarrow$ Prva in zadnja projekcija točk it Q na n.

if ($p_{\min} < q_{\max}$ and $p_{\max} < q_{\min}$) or ($q_{\min} < p_{\max}$ and $q_{\max} < p_{\min}$) :

return $False$

return $True$

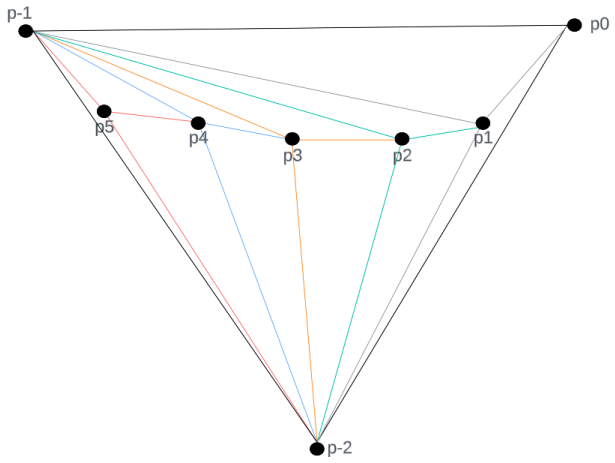
Oba algoritma sta implementirana v `rotating_calipers_and_SAT.py`. Program sprejme datoteko s točkami `points.txt` in izhod napiše v `output.txt`.

Naloga 2: Delaunay triangualizacija

Delaunay triangualizacija je algoritem, s katerim dane točke poveže s trikotniki. V danem članku je opisan naključnostni algoritem, saj na začetku poljubno permutira točke. Recimo, da vhodne točke ležijo blizu neke premice ali krivulje in jih algoritem permutira tako, da so urejene padajoče glede na x-koordinato.

Slika 1: Permutirane vhodne točke.

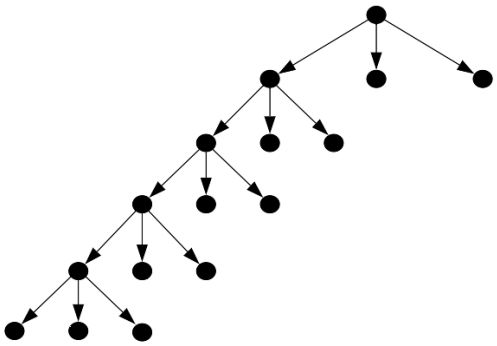
Na vsakem koraku zanke, bo algoritem povezal najbolj desno točko, ki še ni bila povezana z ogljišči trikotnika v katerem je.



Slika 2: Delaunay triangualizacija.

Istočasno pa gradi usmerjen acikličen graf $\mathcal{D}$, ki nosi informacijo o trenutnih in uničenih trikotnikih. Ker točke dodaja vsakič znotraj najbolj levega trikotnika, bodo novi listi na grafu dodani vsakih pod istimi vozlišči, tako da bo končen graf neuravnovešen.

Na vsakem koraku mora algoritem najti kam postaviti novo točko. Ker je graf $\mathcal{D}$ vse bolj neuravnovešen, bo za iskanje porabil vse več časa. Za prvo točko mora pregledati le tri vozlišča (liste pod korenom). Za drugo jih mora pregledati šest. Za k-to točko bo moral pregledati $3k$ vozlišč. To mora narediti



Slika 3: Usmerjen acikličen graf.

za vsako točko, torej bo za vsako od n točk porabil $\mathcal{O}(n)$ časa, kar nam da časovno zahtevnost $\mathcal{O}(n^2)$.

Naloga 3: Drevesne strukture

Prednosti drevesnih struktur

Četrtinska drevesa lahko uporabimo za kompresijo slik. Kompresija najbolje deluje za slike, ki imajo velike dele iste barve, saj namesto $k \times k$ kvadratov shrani samo enega.

Kd-drevesa so uravnotežena binarna drevesa, zato imajo logaritemsko globino in so najprimernejša za hitro iskanje najbližjih sosedov. Dobro delujejo tudi za sparse točke, saj se logaritemska globina ohrani, medtem ko je pri četrtinskih drevesih veliko praznega prostora.???

Intervalna drevesa so primerna za spreminjajča se podatke, na primer premikajoče točke. Namreč drevesna struktura se pri dodajanju in odstranjevanju točk ohrani, medtem ko se struktura četrtinskih dreves in kd-dreves spremeni toliko, da je lažje izdelati novo drevo.

Osmiška drevesa

Podane imamo točke P v 3-dimenzionalnem prostoru. Prostor razdelimo na osem kock. Nadaljne delitve prostora potekajo rekurzivno. Če je v kocki kvečemu ena točka, kocke ne delimo naprej. Če je v kocki več kot ena točka, jo ponovno razdelimo na osem manjših kock in rekurzivno ponovimo na vseh novih otrocih. Podrobneje je algoritem sledeč.

(* Naj bodo točke predstavljene kot podatkovna struktura POINT(x, y, z). *)
struct POINT (x, y, z)

(* Naj bodo vozlišča predstavljena kot par (position, data), kjer prvi element predstavlja pozicijo v prostoru (kot točka), drugi pa vrednost, ki jo vozlišče nosi. *)
struct NODE (position, data)

(* Definirajmo novo podatkovno strukturo Box, ki bo rekurzivno gradila Osmiško drevo. *)
struct BOX (leftTopBack, rightBottomFront, maxCapacity):
    (* leftTopBack in rightBottomFront sta ustrezni
    nasprotni točki na diagonali kocke. *)

    (* Otroci (manjše kocke) so na začetku prazni. *)
    leftTopBackBox = None
    leftTopFrontBox = None
    leftBottomBackBox = None
    leftBottomFrontBox = None
    rightTopBackBox = None
    rightTopFrontBox = None
    rightBottomBackBox = None
    rightBottomFrontBox = None

```
(*Točke shranjene v vozlišču.*)
points = [ ]

(* Funkcija za vstavljanje novih vozlišč. *)
define insert (node):

    (*Če je pozicija izven mej kocke, ne naredimo nič.*)
    if node.position.x ¡ leftTopBack.x or
        node.position.x ¿ rightBottomFront.x or
        node.position.y ¿ leftTopBack.y or
        node.position.y ¡ rightBottomFront.y or
        node.position.z ¿ leftTopBack.z or
        node.position.z ¿ leftTopBack.z :
            do nothing

    (*Če smo dosegli najmanjšo možno velikost kocke,
    shranimo vozlišče, če je v kocki še prostor.*)
    if points.size ¡ maxCapacity:
            points.append(node)

    (*Sicer pa preverimo v katero manjšo kocko spada.*)

    (*Točka je na levem delu.*)

    if (leftTopBack.x + rightBottomFront.x)/2 ≥ node.position.x:

            (*Točka je na sprednjem delu.*)

            if (leftTopBack.y + rightBottomFront.y)/2 ≥ node.position.y:

                    (*Točka je na spodnjem delu.*)

                    if (leftTopBack.z + rightBottomFront.z)/2 ≥ node.position.z:
                            leftBottomFrontBox.insert(node)

                    (*Točka je na zgornjem delu.*)

                    else:
                            leftTopFrontBox.insert(node)

            (*Točka je na zadnjem delu.*)

            else:

                    (*Točka je na spodnjem delu.*)

                    if (leftTopBack.z + rightBottomFront.z)/2 ≥ node.position.z:
                            leftBottomBackBox.insert(node)

                    (*Točka je na zgornjem delu.*)

                    else:
                            leftTopBackBox.insert(node)

    (*Točka je na desnem delu.*)
```

else:

(*Točka je na sprednjem delu.*)

if $\frac{\text{leftTopBack.y + rightBottomFront.y}}{2} \geq$ node.position.y:

(*Točka je na spodnjem delu.*)

if $\frac{\text{leftTopBack.z + rightBottomFront.z}}{2} \geq$ node.position.z:

rightBottomFrontBox.insert(node)

(*Točka je na zgornjem delu.*)

else:

rightTopFrontBox.insert(node)

(*Točka je na zadnjem delu.*)

else:

(*Točka je na spodnjem delu.*)

if $\frac{\text{leftTopBack.z + rightBottomFront.z}}{2} \geq$ node.position.z:

rightBottomBackBox.insert(node)

(*Točka je na zgornjem delu.*)

else:

rightTopBackBox.insert(node)